Een introductie in

programmeren in Emacs Lisp

Een introductie in programmeren in Emacs Lisp

door by Robert J. Chassell

Dit is *Een introductie in Programmeren in Emacs Lisp*, voor mensen die geen programmeur zijn.

Gedistribueerd met Emacs versie [No value for “EMACSVER”].

Gepubliceerd door de:

GNU Press,
a division of the
Free Software Foundation, Inc.
31 Milk Street, # 960789
Boston, MA 02196 USA

`https://www.fsf.org/licensing/gnu-press/`
email: `sales@fsf.org`
Tel: +1 (617) 542-5942
Fax: +1 (617) 542-2652

ISBN 1-882114-43-4

Korte inhoudsopgave

Inhoudsopgave

Voorwoord

Het grootste deel van de GNU Emacs geïntegreerde omgeving is geschreven in de programmeertaal met de naam Emacs Lisp. De code in deze taal is de software, de verzameling instructies, die de computer vertellen wat te doen wanneer je die commando's geeft. Emacs is zo ontworpen dat je nieuwe code in Emacs Lisp kunt schrijven en die makkelijk als uitbreiding op de editor installeert.

(GNU Emacs wordt soms een 'uitbreidbare editor" genoemd, maar biedt veel meer dan mogelijkheden voor tekstbewerking. Het is beter om Emacs als een "uitbreidbare computer omgeving" te beschrijven. Dat is echter nogal een mond vol. Het is eenvoudiger om Emacs een editor te noemen. Bovendien is alles wat je in Emacs doet, zoals het vinden van de Maya-datum, fasen van de maan, het vereenvoudigen van polynomen, code debuggen, bestanden beheren, brieven lezen, boeken schrijven, al deze activiteiten komen neer op tekst bewerken in de breedste zin van het woord.)

Alhoewel men Emacs Lisp meestal alleen in samenhang met Emacs ziet, is het een volwaardige computer programmeertaal. Je kunt Emacs Lisp net als elke andere programmeertaal gebruiken.

Misschien wil je programmeren begrijpen, misschien wil je Emacs uitbreiden, of misschien wil je een programmeur worden. Deze introductie in Emacs Lisp is bedoeld om je op weg te helpen, je te begeleiden bij het leren van de beginselen van programmeren en, belangrijker, te laten zien hoe je jezelf kunt leren verder te gaan.

Over het lezen van deze tekst

Verspreid over dit document kom je kleine voorbeeldprogramma's tegen die je in Emacs kunt draaien. Wanneer je dit document in Info binnen GNU Emacs leest, dan draai je de programma's zoals ze verschijnen. (Dit is eenvoudig te doen en wordt uitgelegd bij de presentatie van de voorbeelden.) Daarnaast kan je deze introductie lezen in een gedrukt boek zittend bij een computer waarop Emacs draait. (Dit is wat ik graag doe, ik hou van gedrukte boeken.) Wanneer je geen draaiende Emacs bij de hand hebt dan kan je nog steeds dit boek lezen. Beschouw het in dat geval als een roman of een reisgids naar een land waar je niet geweest bent, interessant, maar niet hetzelfde als er daadwerkelijk zijn.

Een groot deel van deze introductie leidt je rond door de code van GNU Emacs. Deze rondleidingen dienen twee doeleinden. Ten eerste om je vertrouwd te maken met de echte, werkende code (code die je dagelijks gebruikt) en ten tweede om je vertrouwd te maken met de manier waarop Emacs werkt. Het is interessant te zien hoe een werkomgeving is geïmplementeerd. Ook hoop ik dat je de gewoonte overneemt om de source code door te bladeren. Je kan er van leren en je kunt er ideeën uit halen. GNU Emacs hebben is zoals een drakengrot vol schatten hebben.

Naast het leren van Emacs als editor en Emacs Lisp als programmeertaal geven de voorbeelden en de rondleiding je de gelegenheid vertrouwd te raken met Emacs als een Lisp programmeeromgeving. GNU Emacs ondersteunt het programmeren en verschaft gereedschappen waar je comfortabel gebruik van wilt maken, zoals `M-.` (de toetsaanslag waarmee je het `xref-find-definitions` commando start). Je leert

ook over buffers en andere objecten die onderdeel van de omgeving uitmaken. Deze functionaliteiten van Emacs ontdekken is als nieuwe routes door je geboorteplaats ontdekken.

Tenslotte hoop ik hiermee vaardigheden over te brengen over het gebruik van Emacs om aspecten van het programmeren te leren die je nog niet weet. Vaak kan Emacs je helpen om dingen te begrijpen of te ontdekken hoe je iets nieuws doet. Deze zelfredzaamheid is niet alleen prettig, maar ook een voordeel.

Voor wie dit is geschreven

De tekst is bedoeld als een elementaire introductie voor mensen die geen programmeur zijn. Wanneer je een programmeur bent, zal je niet tevreden zijn met deze introductie. De reden daarvoor is dat je waarschijnlijk een expert bent geworden in het lezen van referentiehandleidingen en niet blij wordt van de manier waarop deze tekst is opgezet.

Een expert-programmeur die deze tekst heeft gereviewed zei me:

> *Ik geeft de voorkeur aan het bestuderen van referentiehandleidingen. Ik "duik" in elke paragraaf en "kom boven voor lucht" tussen de paragrafen. Wanneer ik aan het einde van een paragraaf ben verwacht ik dat het onderwerp klaar is, dat ik alles weet wat ik moet weten (met als mogelijke uitzondering het geval de volgende paragraaf dieper op de inhoud ingaat). Ik verwacht dat een goed geschreven referentiehandleiding weinig herhaling bevat en dat het uitstekende verwijzingen heeft naar de (enige) plek waar de informatie is die ik zoek.*

Deze introductie in niet voor die persoon geschreven!

Ten eerste probeer ik alles tenminste drie keer te behandelen: eerst om het te introduceren, dan om het in context te laten zien, en vervolgens in een andere context, of om het te herhalen.

Ten tweede stop ik vrijwel nooit alle informatie over een onderwerp op een enkele plek, laat staat in een paragraaf. In mijn manier van denken legt dit een te zware last op de lezer. In plaats daarvan probeer ik alleen uit te leggen wat de lezer op dat moment moet weten. (Soms voeg ik wat extra informatie toe zodat je later niet verrast wordt wanneer de aanvullende informatie formeel wordt geïntroduceerd.)

Wanneer je deze test leest wordt niet van je verwacht dat je alles meteen de eerste keer leert. Je hoeft slechts vluchtig kennis te maken met enkele van de genoemde onderdelen. Ik hoop dat ik de tekst zo gestructureerd heb en je genoeg hints gegeven heb dat je bewust bent van wat belangrijk is en je daarop concentreert.

Je zult in sommige paragrafen moeten duiken, er is geen andere manier om die te lezen. Maar ik heb geprobeerd hun aantal zo klein mogelijk te houden. Dit boek is bedoeld als een toegankelijke heuvel in plaats van een ontmoedigende berg.

Dit boek, *Een introductie in programmeren in Emacs Lisp*, heeft een bijpassend document, *The GNU Emacs Lisp Reference Manual*. Deze referentiehandleiding bevat meer details dan deze introductie. In de referentiehandleiding is alle informatie met betrekking tot een onderwerp op een plek geconcentreerd. Je zou die moeten gebruiken wanneer je bent zoals de hierboven genoemde programmeur. En nadat je

deze *Introductie* gelezen hebt is de *Reference Manual* natuurlijk bruikbaar tijdens het schrijven van je eigen programma's.

Geschiedenis van Lisp

Lisp is oorspronkelijk eind jaren '50 ontwikkeld in het Massachusetts Institute of Technology voor onderzoek op het gebied van kunstmatige intelligentie. De grote kracht van de Lisp taal maakt het ook voor andere toepassingen superieur, zoals het schrijven van editor-commando's en geïntegreerde omgevingen.

GNU Emacs Lisp is grotendeels geïnspireerd door Maclisp, dat in de jaren '60 in het MIT is geschreven. Het is ook enigszins geïnspireerd door Common Lisp, dat in de jaren '80 een standaard werd. Emacs Lisp is echter een stuk eenvoudiger dan Common Lisp. (De standaard Emacs distributie bevat een optioneel uitbreidingsbestand `cl-lib.el`, dat veel Common Lisp functionaliteiten aan Emacs Lisp toevoegt.)

Een opmerking voor beginners

Ook als je GNU Emacs niet kent heeft het lezen van dit document veel nut. Ik raad je echter aan om Emacs te leren, al is het alleen maar te leren hoe je over het scherm beweegt. Je kunt zelf het gebruik van Emacs leren met de ingebouwde tutorial. Om die te gebruiken, type je *C-h t*. (Dit betekent dat je de toetsen CTRL *h* tegelijk indrukt en loslaat, en daarna de toets *t* indrukt en loslaat.)

Ook verwijs ik vaak naar Emacs's standaard commando's door de toetscombinaties te noemen die je gebruikt om het commando aan te roepen en vervolgens de naam van het commando tussen haakjes, zoals dit:x *C-M-* (`indent-region`). Wat dit betekent is dat het `indent-region` commando gebruikelijk wordt aangeroepen door *C-M-* te typen. (Je kunt, als je dat wilt, de toetscombinaties om commando's aan te roepen wijzigen, dit heet *rebinding*. Zie Sectie 16.8 "Keymaps", pagina 208.) De afkorting *C-M-* betekent dat je de CTRL-toets, de META-toets en de \-toets allemaal tegelijk indrukt. Soms wordt een toetscombinatie als dit een key-chord genoemd, omdat het vergelijkbaar is met de manier waarop je een akkoord op een piano speelt. Wanneer je toetsenbord geen META-toets heeft, dan gebruik je de ESC-toets als prefix. In dat geval betekent *C-M-* dat je eerst de ESC indrukt en loslaat en daarna tegelijkertijd de CTRL-toets en de \-toets indrukt. Maar meestal betekent *C-M-* dat je de CTRL-toets samen met de toets die als ALT is gelabeld indrukt en tegelijkertijd de \-toets indrukt.

In aanvulling op het typen van een toetscombinatie, kan je wat je typt uitbreiden met de prefix *C-u*, wat het *universal argument* heet. De *C-u*-toetscombinatie geeft een argument door aan het eerstvolgende commando. Dus, om een gebied met gewone tekst 6 spaties te indenten, markeer je het gebied en vervolgens type je *C-u 6 C-M-*. (Wanneer je geen getal opgeeft, dan geeft Emacs hetzij het getal 4 door aan het commando, of voert het commando anders uit). Zie Sectie "Numeric Arguments" in *The GNU Emacs Manual*.

Wanneer je dit in Info met GNU Emacs leest, dan kun je het hele document lezen slechts door het indrukken van de spatiebalk, `SPC`. (Om meer over Info te leren, type `C-h i` en selecteer daarna Info.)

Een opmerking over terminologie: wanneer ik het woord Lisp alleen gebruik, dan refereer ik meestal naar de verschillende dialecten van Lisp in het algemeen, maar wanneer ik het over Emacs Lisp heb, dan refereer ik naar GNU Emacs Lisp in het bijzonder.

Dank

Mijn dank aan iedereen die met met dit boek hielpen. Mijn speciale dank gaat uit naar Jim Blandy, Noah Friedman, Jim Kingdon, Roland McGrath, Frank Ritter, Randy Smith, Richard M. Stallman en Melissa Weisshaus. Mijn dank gaat ook uit naar Philip Johnson en David Stampe voor hun geduldige aanmoediging. Mijn vergissingen zijn van mij zelf.

Robert J. Chassell

bob@gnu.org

1 Lijsten verwerken

Voor het ongetrainde oog is Lisp een vreemde programmeertaal. In Lisp code staan overal haakjes. Sommige mensen stellen zelfs dat de naam voor "Lots of Isolated Silly Parentheses" staat. Maar die stelling is ongegrond. Lisp staat voor LISt Processing en de programmeertaal behandelt *lijsten* (en lijsten van lijsten) door ze tussen haakjes te zetten. De haakjes markeren de grenzen van de lijst. Soms wordt een lijst voorafgegaan door een apostrof '`'`', in Lips heet dat een *single-quote*.[1] Lijsten zijn de basis van Lisp.

1.1 Lisp lijsten

In Lisp ziet een lijst er zo uit: `'(roos viool madelief boterbloem)`. Deze lijst wordt voorafgegaan door een enkele apostrof. Het kon net zo goed als volgt geschreven worden, wat meer lijkt op de soort van lijsten waar je waarschijnlijk bekend mee bent:

```
'(roos
  viool
  madelief
  boterbloem)
```

De elementen van deze lijst zijn de namen van vier verschillende bloemen, onderling gescheiden door witte ruimte en omgeven door haakjes, net als bloemen in een veld met een stenen muur er omheen.

Lijsten kunnen ook getallen bevatten, zoals deze lijst: `(+ 2 2)`. Deze lijst heeft een plusteken, '`+`', gevolgd door twee '`2`'-en, onderling gescheiden met witte ruimte.

In Lisp worden zowel data als programma's op dezelfde manier gerepresenteerd. Het zijn beiden lijsten van woorden, getallen, andere lijsten, gescheiden door witte ruimte en omgeven door haakjes. (Omdat een programma er uit ziet als data, kan het ene programma makkelijk dienen als data voor een ander; dit is een erg krachtige eigenschap van Lisp.) (Overigens, deze twee opmerkingen tussen haakjes zijn *geen* Lisp lijsten, omdat zij '`;`' en '`.`' als leesteken bevatten.)

Hier is een andere lijst, deze keer met daarin een lijst.

```
'(deze lijst heeft (van binnen een lijst))
```

De componenten van deze lijst zijn de woorden '`deze`', '`lijst`', '`heeft`' en de lijst '`(van binnen een lijst)`'. De binnenste lijst bestaat uit de woorden '`van`', '`binnen`', '`een`' en '`lijst`'.

1.1.1 Lisp atomen

Wat we woorden genoemd hebben, heten in Lisp *atomen*. Deze term komt van de historische betekenis van het woord atoom, wat "onscheidbaar" betekent. Wat Lisp betreft kunnen de woorden die we in de lijst gebruikt hebben niet in kleinere delen opgedeeld worden en nog steeds dezelfde betekenis als onderdeel van een programma hebben, net als getallen en enkele-karakter symbolen zoals '`+`'. Aan de andere kant,

[1] Een single-quote is een afkorting voor de speciale vorm `quote`. Je hoeft nu nog niet over speciale vormen na te denken. Zie Sectie 1.5 "Lisp interpreter", pagina 6.

in tegenstelling tot het oude atoom, kan een lijst in delen worden opgesplitst. (Zie Hoofdstuk 7 "`car` `cdr` en `cons` Fundamentele functies", pagina 78.)

In een lijst worden atomen onderling gescheiden door witte ruimte. Zij kunnen meteen naast een haakje staan.

Technisch gesproken bestaat een lijst in Lisp uit haakjes rondom atomen gescheiden door witte ruimte of rondom andere lijsten of rondom zowel atomen als andere lijsten. Een list kan slechts een atoom bevatten, of helemaal niets bevatten. Een lijst met niets er in ziet er zo uit: `()`, en heet een *lege lijst*. In tegenstelling tot alles anders wordt een lege lijst tegelijk zowel als een atoom als een lijst beschouwd.

De afgedrukte representatie van zowel atomen als lijsten heten *symbolische expressies* of, korter, *s-expressies*. Het woord *expressie* op zichzelf kan hetzij naar de afgedrukte representatie hetzij naar het atoom of lijst zoals die intern in de computer staat verwijzen. Vaak gebruiken mensen de term *expressie* willekeurig. (Ook wordt in veel teksten het woord *vorm* gebruikt als synoniem voor expressie.)

Toevallig werden de atomen waaruit ons universum bestaat zo genoemd omdat men dacht dat ze ondeelbaar zijn, maar er is vastgesteld dat fysieke atomen niet ondeelbaar zijn. Deeltjes kunnen van een atoom afsplitsen of het kan zich in twee ongeveer even grote delen opsplitsen. Fysieke atomen werden voortijdig zo genoemd, voordat hun ware aard was ontdekt. In Lisp kunnen sommige soorten atomen, zoals een array, in delen worden gescheiden, maar het mechanisme om dit te doen wijkt af van het mechanisme om een lijst op te splitsen. Voor zover het lijst-operaties betreft, zijn de atomen van een lijst ondeelbaar.

Net als in het Engels, is de betekenis van de samenstellende letters van een Lisp atoom verschillend van de betekenis die de letters als woord maken. Bijvoorbeeld het Engelse woord voor de Zuid-Amerikaanse luiaard, '`ai`', is volledig anders dan de twee Engelse woorden '`a`' en '`i`'.

In de natuur bestaan veel verschillende soorten atomen, maar slechts een paar in Lisp: bijvoorbeeld *getallen*, zoals 37, 511 of 1729, en *symbolen*, zoals '`+`', '`foo`', of '`forward-line`'. In het dagelijks Lisp taalgebruik wordt het woord "atoom" maar weinig gebruikt, omdat programmeurs meestal specifieker zijn over het soort atoom waar ze mee te maken hebben. Lisp programmeren betreft hoofdzakelijk symbolen (en soms getallen) binnen lijsten. (Overigens, de voorgaande opmerking met drie woorden tussen haakjes is een geldige lijst in Lisp, omdat het uit atomen bestaat, welke in dit geval symbolen zijn, gescheiden door witte ruimte en omgeven door haakjes, zonder enige non-Lisp leestekens.)

Tekst tussen dubbele aanhalingstekens —zelfs zinnen en paragrafen— is ook een atoom. Hier is een voorbeeld:

```
'(deze lijst bevat "tekst tussen aanhalingstekens.")
```

In Lisp is alle gequote tekst inclusief leestekens en witte ruimte een enkele atoom. Deze atoom-soort heet een *string* (wegens "string of characters") en is het soort dingen dat gebruikt wordt voor boodschappen die de computer toont om door mensen gelezen te worden. Strings zijn een andere atoom-soort dan getallen of symbolen en worden anders gebruikt.

1.1.2 Whitespace in lijsten

De hoeveelheid witte ruimte in een lijst maakt niet uit. Vanuit het gezichtspunt van de Lisp taal, is

```
'(deze lijst
    lijkt op dit)
```

exact hetzelfde als dit:

```
'(deze lijst lijkt op dit)
```

Beide voorbeelden tonen wat Lisp als dezelfde lijst ziet, de lijst samengesteld uit de symbolen '`deze`', '`lijst`', '`lijkt`', '`op`' en '`dit`' in die volgorde.

Extra witte ruimte en regels zijn bedoeld om de lijst meer voor mensen leesbaar te maken. Wanneer Lisp de expressie leest, verwijdert het alle extra witte ruimte (maar heeft tenminste een spatie tussen de atomen nodig om ze uit elkaar te houden).

Hoe vreemd het ook lijkt, de voorbeelden die we gezien hebben dekken alles hoe lijsten er uit zien! Elke andere lijst in Lisp ziet er min of meer hetzelfde uit als een van deze voorbeelden, behalve dat de lijst langer of meer complex kan zijn. In het kort, een lijst is tussen haakjes, een string tussen aanhalingstekens, een symbool ziet er uit als een woord en een getal als een getal. (Voor bepaalde situaties worden vierkante haken, punten en enkele andere speciale karakters gebruikt, maar we gaan vrij ver zonder hen.)

1.1.3 GNU Emacs helpt je om lijsten te typen

Wanneer je een Lisp expressie in GNU Emacs typt, met hetzij Lisp Interaction mode of Emacs Lisp mode, dan zijn verschillende commando's beschikbaar om de Lisp expressie zo te formatteren dat deze makkelijk te lezen is. Bijvoorbeeld het indrukken van de `TAB`-toets laat de regel waar de cursor is automatisch met de juiste hoeveelheid inspringen. Het commando om de code in een region juist te laten inspringen is normaal gesproken gebonden aan `C-M-\`. De inspringingen zijn ontworpen om te laten zien welke elementen van een lijst bij welke lijst horen—elementen van een sub-lijst springen meer in dan elementen van de insluitende lijst.

Bovendien, wanneer je een haakje sluiten typt, dan laat Emacs tijdelijk de cursor terugspringen naar het bijbehorende haakje openen, zodat je kunt zien welke dat is. Dit is erg zinvol, omdat elke lijst die je typt in Lisp een haakje sluiten horend bij zijn haakje openen moet hebben. (Zie Sectie "Major Modes" in *The GNU Emacs Manual*, voor meer informatie over Emacs's modes.)

1.2 Draai een programma

Een lijst in Lisp—elke lijst—is een programma om te draaien. Wanneer je het draait (wat in Lisp-jargon *evalueren* is), doet de computer een van deze drie dingen: doe niets behalve de lijst zelf teruggeven; een foutmelding geven; of het eerste symbool in de lijst behandelen als commando om iets te doen. (Meestal is dat laatste natuurlijk wat je echt wilt!)

De enkele apostrof, `'`, die ik aan de voorkant van enkele voorbeeld lijsten in de voorgaande hoofdstukken gebruikte, heten een *quote*. Wanneer die aan een lijst voorafgaan, dan vertelt dat Lisp om niets met de lijst te doen, anders dan te ac-

cepteren precies zoals die is geschreven. Maar wanneer er geen quote aan de lijst voorafgaat, dan is het eerste element in de lijst speciaal: het is een commando voor de computer om te gehoorzamen. (In Lisp, heten deze commando's *functies*.) De eerder genoemde lijst `(+ 2 2)` is een instructie om iets met de rest van de lijst te doen: tel de getallen die volgen op.

Wanneer je dit binnen GNU Emacs in Info leest, dan kun je de lijst als volgt evalueren: ga met de cursor direct achter het rechtse haakje sluiten staan van de volgende lijst en typ vervolgens *C-x C-e*:

```
(+ 2 2)
```

Je ziet het getal `4` verschijnen in het echogebied[2]. (Wat je zojuist gedaan hebt is het evalueren van de lijst. Het echogebied is de regel aan de onderkant van het scherm dat de echotekst toont.) Probeer nu hetzelfde met een gequote lijst: plaats de cursor meteen achter de volgende lijst en type *C-x C-e*:

```
'(dit is een gequote lijst)
```

Je ziet `(dit is een gequote lijst)` in het echogebied verschijnen.

Wat je in beide gevallen deed is het geven van een commando aan een programma in GNU Emacs met de naam *Lisp interpreter*—de interpreter een commando geven om de expressie te evalueren. De naam van de Lisp interpreter komt van het woord voor de taak die een mens uitvoert die de betekenis van de expressie bedenkt—die het interpreteert.

Je kunt ook een atoom evalueren dat geen deel van een lijst is—een die niet door haakjes ingesloten is. Weer vertaalt de Lisp interpreter van de voor mensen leesbare expressie naar de taal van de computer. Maar voor we dit bespreken (zie Sectie 1.7 "Variabelen", pagina 9), bespreken we eerste wat de Lisp interpreter doet wanneer je een fout maakt.

1.3 Een foutmelding genereren

Deels zodat je je geen zorgen hoeft te maken wanneer je het per ongeluk doet, geven we nu een commando aan de Lisp interpreter dat een foutmelding genereert. Dit is een onschuldige activiteit, en wij zullen inderdaad vaker proberen expres een foutmelding te genereren. Wanneer je eenmaal het jargon begrijpt, kunnen foutmeldingen informatief zijn. In plaats van "fout"-meldingen zouden ze "hulp"-meldingen moeten heten. Zij zijn als wegwijzers voor een reiziger in een vreemd land. Ze ontcijferen kan moeilijk zijn, maar wanneer je ze begrijpt wijzen ze je de weg.

De foutmelding wordt gegenereerd door een ingebouwde GNU Emacs debugger. Wij gaan de debugger binnen. Je verlaat de debugger met het typen van `q`.

Wat we gaan doen is het evalueren van een lijst die niet gequote is en niet een betekenisvol commando als eerste element heeft. Hier is een lijst die bijna gelijk is aan die we niet gebruikt hebben, maar zonder de voorafgaande enkele quote. Positioneer de cursor direct er achter en type *C-x C-e*:

```
(dit is een ongequote lijst)
```

[2] Emacs toont integer waardes in decimaal, in octaal en in hex en ook als een karakter, maar laten we deze gemaksfunctie nu even negeren.

Een `*Backtrace*` window opent waarin je het volgende zou moeten zien:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-function dit)
  (dit is een ongequote lijst)
  eval((dit is een ongequote lijst) nil)
  elisp--eval-last-sexp(nil)
  eval-last-sexp(nil)
  funcall-interactively(eval-last-sexp nil)
  call-interactively(eval-last-sexp nil nil)
  command-execute(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

Je cursor staat in dit venster (mogelijk moet je een paar seconden wachten voordat het zichtbaar wordt). Om de debugger af te sluiten en het venster te laten verdwijnen, typ je:

```
q
```

Gelieve `q` nu te typen, zodat je zelfverzekerd de debugger weet te verlaten. Type vervolgens weer `C-x C-e` om er opnieuw in te gaan.

Gebaseerd op wat we al weten, kunnen we bijna deze foutmelding lezen.

Je leest de `*Backtrace*` buffer van beneden naar boven; het vertelt je wat Emacs deed. Toen je `C-x C-e` typte, maakte je een interactieve aanroep naar het commando `eval-last-sexp`. `eval` is een afkorting van "evalueer" en `sexp` is een afkorting voor "symbolische expressie", de expressie vlak voor de cursor.

Elke regel er boven vertelt je wat de Lisp interpreter er na evalueerde. De meest recente actie staat bovenaan. Het buffer heet het `*Backtrace*` buffer omdat het je in staat stelt Emacs achterwaarts te volgen.

Bovenaan het `*Backtrace*` buffer zie je de regel:

```
Debugger entered--Lisp error: (void-function dit)
```

De Lisp interpreter probeerde het eerste atoom van de lijst te evalueren, het woord '`dit`'. Het is deze actie die de foutmelding '`(void-function dit)`' genereerde.

De melding bevat de woorden '`void-function`' en '`dit`'.

Het woord '`functie`' is een keer eerder genoemd. Het is een erg belangrijk woord. Voor onze doeleinden kunnen we definiëren door te zeggen dat een '`functie`' een verzameling instructies is die de computer vertellen iets te doen.

Nu beginnen we de foutmelding te begrijpen: '`void-function dit`'. De functie (het woord '`dit`') heeft geen definitie van enige verzameling instructies voor de computer om uit te voeren.

Het ietwat vreemde woord '`void-function`' is ontworpen voor de manier waarop Emacs Lisp is geïmplementeerd, dat is wat het doet wanneer een symbool geen functiedefinitie heeft waaraan het gekoppeld is, de plaats die de instructies zou moeten bevatten is leeg.

Aan de andere kant, omdat we in staat waren 2 plus 2 succesvol op te tellen door `(+ 2 2)` te evalueren, kunnen we afleiden dat het symbool `+` een verzameling instructies moet hebben voor de computer om te gehoorzamen en dat deze instructies het optellen van de getallen achter de `+` moeten zijn.

Het is mogelijk te voorkomen dat Emacs de debugger in gaat in situaties zoals deze. Hier leggen we niet uit hoe je dat doet, maar we vertellen hoe het resultaat

er uit ziet, omdat je een vergelijkbare situatie kunt tegenkomen wanneer er een bug zit in de Emacs code die je gebruikt. In zulke gevallen zie je een foutmelding met maar een regel, het verschijnt in het echogebied en ziet er als volgt uit:

```
Symbol's function definition is void: dit
```

De melding gaat weg zodra je een toets aanslaat of zelfs de cursor verplaatst.

We kennen de betekenis van het woord '`Symbool`'. Het refereert naar de eerste atoom van de lijst, het woord '`dit`'. Het woord '`functie`' refereert naar de instructies die de computer vertellen wat te doen. (Technisch gezien, vertelt het symbool de computer waar de instructies zijn te vinden, maar dit is een complicatie dit we op dit moment kunnen negeren.)

De foutmelding kan begrepen worden: '`Symbol's function definition is void: dit`'. Het symbool (het woord '`dit`') mist instructies die de computer kan uitvoeren.

1.4 Symboolnamen en functiedefinities

We kunnen een ander kenmerk van Lisp verwoorden op basis van wat we tot nu toe besproken hebben—en belangrijk kenmerk: een symbool, zoals `+`, is zelf geen verzameling van instructies die de computer kan uitvoeren. In plaats daarvan wordt het symbool gebruikt, misschien tijdelijk, als een manier om de instructies te lokaliseren. Namen van mensen werken op de zelfde manier. Ik kan worden gerefereerd als '`Bob`'. Ik ben echter niet de letters '`B`', '`o`' en '`b`' maar ik ben of was het bewustzijn dat consequent met een bepaalde levensvorm wordt geassocieerd. De naam is niet mij, maar het kan worden gebruikt om naar mij te refereren.

In Lisp kan een enkele verzameling instructies meerdere namen hebben. De computer instructies om getallen op te tellen kan bijvoorbeeld zijn gekoppeld aan zowel het symbool `plus` als het symbool `+` (en zijn dat in sommige Lisp-dialecten). Onder mensen kan ik gerefereerd met zowel '`Robert`' als met '`Bob`' en ook nog andere woorden.

Aan de andere kant kan maar aan een functiedefinitie tegelijk zijn gekoppeld. Anders zou de computer in verwarring raken over welke definitie te gebruiken. Wanneer dit onder mensen het geval zou zijn, dan kan er in de hele wereld maar een persoon de naam '`Bob`' hebben. Echter de functiedefinitie naar de naam naar refereert kan makkelijk worden aangepast. (Zie Sectie 3.2 "Installeer een functiedefinitie", pagina 28.)

Omdat Emacs erg groot is, is het gebruikelijk om de symbolen zo te benoemen dat zij het deel van Emacs waar zij thuishoren identificeren. Alle namen voor functies die met Texinfo te maken hebben beginnen met '`texinfo-`' en die voor de functies die met e-mail lezen te maken hebben beginnen met '`rmail-`'.

1.5 De Lisp interpreter

Op basis van wat we nu gezien hebben kunnen we beginnen te achterhalen wat de Lisp interpreter doet wanneer we de opdracht geven een lijst te evalueren. Eerst kijkt die of er een quote voor de lijst staat. Als die er is, dan geeft de interpreter ons slechts de lijst. Aan de andere kant, als er geen quote is, dan kijkt de interpreter naar

het eerste element van de lijst om te zien of die een functiedefinitie heeft. Wanneer dat het geval is, dan voert de interpreter de instructies in de functiedefinitie uit. Zo niet, dan drukt de interpreter een foutmelding af.

Dit is hoe Lisp werkt. Simpel. Er zijn aanvullende complicaties waar we zo aan toe komen, maar dit zijn de basisprincipes. Om Lisp-programma's te schrijven moet je natuurlijk weten hoe je functiedefinities schrijft en aan een naam koppelt, en hoe je dit doet zonder zelf of de computer in verwarring te brengen.

Nu komt de eerste complicatie. Naast lijsten kan de Lisp interpreter een symbool evalueren dat niet is gequote en niet tussen haakjes staat. De Lisp interpreter zal proberen de waarde van het symbool als een *variabele* te bepalen. Deze situatie wordt besproken in het hoofdstuk over variabelen. (Zie Sectie 1.7 "Variabelen", pagina 9.)

De tweede complicatie ontstaat doordat sommige functies ongebruikelijk zijn en niet op de normale manier werken. Zij die dat niet doen worden *speciale vormen* genoemd. Zij worden voor speciale taken gebruikt, zoals het definieren van een functie, er er zijn er niet veel van. In enkele van de volgende hoofdstukken maak je kennis met verschillende belangrijkere speciale vormen.

Naast speciale vormen zijn er ook *macros*. Een macro is een in Lisp gedefineerde constructie die afwijkt van een functie omdat het een Lisp expressie vertaalt in een andere expressie, die in plaats van de orginele expressie geëvalueerd gaat worden.

Voor deze introductie hoef je je niet al te veel zorgen te maken of iets een speciale vorm, een macro of een gewone functie is. `if` bijvoorbeeld is een speciale vorm (zie Sectie 3.7 "if", pagina 38), maar `when` is een macro. In eerdere versies van Emacs was `defun` een speciale vorm, maar nu is het een macro (zie Sectie 3.1 "defun", pagina 26). Het gedraagt zich nog steeds op dezelfde manier.

De laatste complicatie is het volgende: wanneer een functie waar de Lisp interpreter naar kijkt geen speciale vorm is, en deel van een lijst uitmaakt, dan kijkt de interpreter of de lijst een lijst in zich bevat. Wanneer er een binnenliggende lijst is, dan zoekt de interpreter eerst uit wat het moet doen met die binnenliggende lijst en daarna gaat die met de buitenste lijst aan de slag. Als er weer een lijst opgenomen is in de binnenliggende lijst, dan werkt die eerst aan die lijst, enzovoorts. Het begint steeds met de meest binnenliggende lijst, om het resultaat van die lijst te evalueren. Het resultaat kan gebruikt worden door de omsluitende expressie.

Zo niet, dan werkt de interpreter van links naar rechts, van de ene expressie naar de ander.

1.5.1 Byte compiling

Een ander aspect van interpreteren: de Lisp interpreter is in staat om twee soorten entiteiten te interpreteren: voor mensen leesbare code, op welke wij ons exclusief richten, en speciaal geproduceerde code, *byte compiled* code genoemd, die niet door mensen te lezen is. Byte compiled code draait sneller dan voor mensen leesbare code.

Je transformeert voor mensen leesbare code in byte compiled code met een van de compile commando's zoals `byte-compile-file`. Byte compiled code is meestal opgeslagen in een bestand dat eindigt met de extensie `.elc`. Je ziet beide soorten

bestanden in de `emacs/lisp` directory. De bestanden om te lezen zijn die met de extensie `.el`.

Praktisch gesproken, voor de meeste dingen die je doet om Emacs aan te passen of uit te breiden is er geen noodzaak voor byte compile, en ik bespreek dit onderwerp hier niet. Zie Sectie "Byte Compilation" in *The GNU Emacs Lisp Reference Manual*, voor een volledige beschrijving van byte compilation.

1.6 Evaluatie

Wanneer de Lisp interpreter een expressie verwerkt, dan is de term voor die activiteit *evaluatie*. We zeggen dat de interpreter "de expressie evalueert". Ik heb deze term verschillende keren eerder gebruikt. Het woord komt van het gebruik in alledaagse taal, "de waarde bepalen", "taxeren", volgens de *Webster's New Collegiate Dictionary*.

Na het evalueren van een expressie zal de Lisp interpreter hoogstwaarschijnlijk de waarde *teruggeven* die de computer produceert bij het uitvoeren van de instructies die die in de functiedefinitie vondt, of misschien geeft die de functie op en produceert een foutmelding. (De interpreter kan bij wijze van spreken ook naar een andere functie geworpen worden, of het kan proberen continue wat het doet voor altijd en eeuwig te herhalen in een oneindige loop. Deze activiteiten zijn minder gebruikelijk en wij kunnen ze negeren.) Meestal geeft de interpreter een waarde terug.

Op hetzelfde moment dat de interpreter een waarde terug geeft kan het ook iets anders doen, zoals de cursor verplaatsen of een bestand kopiëren. Deze andere soort activiteit heet een *zij-effect*. Acties die wij belangrijk vinden, zoals het tonen van resultaten, zijn vaak zij-effecten voor de Lisp interpreter. Het is vrij eenvoudig te leren hoe deze zij-effecten te gebruiken.

Samenvattend, het meest gebruikelijk is dat het evalueren van een symbolische expressie zorgt dat de Lisp interpreter een waarde teruggeeft en misschien ook een zij-effect uitvoert, of anders een fout produceert.

1.6.1 Geneste lijsten evalueren

Wanneer de evaluatie van toepassing is op een lijst die binnen een andere lijst zit, kan de omsluitende lijst de waarde gebruiken van de eerste evaluatie als informatie bij het evalueren van de omsluitende lijst. Dit verklaart waarom de binnenste expressies het eerst worden geëvalueerd: de waarden die zij teruggeven worden gebruikt door de buitenliggende expressies.

Wij kunnen dit proces onderzoeken door het evalueren van een ander optelvoorbeeld. Plaats je cursor achter de volgende expressie en type `C-x C-e`:

```
(+ 2 (+ 3 3))
```

Het getal 8 zal in het echogebied verschijnen.

Wat er gebeurt is dat de Lisp interpreter eerst de waarde van de binnenste expressie, `(+ 3 3)` evalueert, wat de waarde 6 teruggeeft. Vervolgens evalueert het de buitenste expressie alsof die als `(+ 2 6)` geschreven was, wat de waarde 8 teruggeeft. Omdat er geen omsluitende expressies meer te evalueren zijn, toont de interpreter de waarde in het echogebied.

Nu is de naam van het commando aangeroepen met de toetsaanslagen `C-x C-e` te begrijpen: de naam is `eval-last-sexp`. De letters `sexp` zijn een afkorting van "symbolische expressie", en `eval` is een afkorting van "evalueer". Het commando evalueert de laatste symbolische expressie.

Als een experiment kan je proberen de expressie te evalueren door de cursor aan het begin van de regel direct onder de expressie te plaatsen, of binnen de expressie.

Hier is een andere kopie van de expressie:

```
(+ 2 (+ 3 3))
```

Als je de cursor aan het begin van de lege regel die onmiddellijk op de expressie volgt plaatst en je `C-x C-e` typt, krijg je nog steeds de waarde 8 getoond in het echogebied. Probeer nu de cursor binnen de expressie te plaatsen. Als je die rechts achter het een-na-laatste haakje plaatst (zodat het lijkt of die bovenop het laatste haakje zit), dan krijg je een 6 getoond het echogebied! Dit is omdat het commando de expressie `(+ 3 3)` evalueert.

Plaats nu de cursor direct achter een getal. Typ `C-x C-e` en je krijgt het getal zelf. Wanneer je in Lisp een getal evalueert, dan krijg je het getal zelf—dit is hoe getallen verschillen van symbolen. Wanneer je een lijst evalueert die begint met een symbool zoals `+`, dan krijg je de waarde terug die het resultaat is van de computer die de instructies uitvoert die zijn gekoppeld aan die naam. Wanneer je het symbool op zichzelf evalueert, dan gebeurt er iets anders, zoals we in het volgende hoofdstuk zien.

1.7 Variabelen

In Emacs Lisp kan een symbool een waarde aan zich gekoppeld hebben, net zoals het een functiedefinitie aan zich gekoppeld kan hebben. Die twee zijn verschillend. De functiedefinitie is een verzameling instructies die de computer zal gehoorzamen. Daartegenover is een waarde iets, zoals een getal of een naam die kan variëren (daarom wordt zo'n symbool een variabele genoemd). De waarde van een symbool kan elke expressie in Lisp zijn, zoals een symbool, getal, lijst of string. Een symbool die een waarde heeft wordt vaak een *variabele* genoemd.

Een symbool kan zowel een functiedefinitie als een waarde tegelijkertijd aan zich gekoppeld hebben. Of het kan slechts een van die tee hebben. De twee staan los van elkaar. Dit is ongeveer vergelijkbaar me de manier waarop Cambridge kan refereren naar een stad in Massachusetts en informatie aan de naam gekoppeld kan hebben, zoals "geweldig programmeercentrum".

Een andere manier om hierover te denken is een symbool voor te stellen als een ladekast. De functiedefinitie wordt in de ene lade gestopt, de waarde in een andere, enzovoorts. Wat in de lade die de waarde bevat gestopt is kan worden veranderd zonder effect te hebben op de lade die de functiedefinitie bevat, en omgekeerd.

De variabele `fill-column` illustreert een symbool waar een waarde aan gekoppeld is. In elk GNU Emacs buffer is dit symbool op een bepaalde waarde ingesteld, meetal 72 of 70, maar soms een andere waarde. Om de waarde van dit symbool te vinden evalueer je het op zichzelf. Wanneer je dit in Info binnen GNU Emacs leest,

dan doe je dit door de cursor achter het symbool te zetten en vervolgens `C-x C-e` te typen:

```
fill-column
```

Nadat ik `C-x C-e` typte, toonde Emacs het getal 72 in mijn echogebied. Dit is waarde waar voor mij `fill-column` op is ingesteld terwijl ik dit schrijf. Het kan verschillend zijn in jouw Infobuffer. Merk op dat de waarde die de variabele teruggeeft op precies dezelfde manier wordt getoond als de waarde die teruggeven wordt door een functie die zijn instructies uitvoert. Vanuit het gezichtspunt van de Lisp interpreter is een teruggeven waarde een teruggegeven waarde. Van wat voor soort expresse het vandaan kwam doet er niet meer toe zodra de waarde bekend is.

Een symbool kan elke waarde aan zich gekoppeld hebben, of om het jargon te gebruiken, we kunnen de variabele aan een waarde *binden*: aan een getal, zoals 72, aan een string `"zoals deze"`, aan een lijst, zoals `(spar den eik)`; we kunnen zelfs een variabele aan een functiedefinitie binden.

Een symbool kan op verschillende manieren aan een waarde worden gebonden. Zie Sectie 1.9 “De waarde van een variabele zetten”, pagina 16, voor informatie over een manier om dit te doen.

1.7.1 Foutmelding voor een symbool zonder functie

Bij het evalueren van `fill-column` om de waarde daarvan als een variabele te ontdekken, plaatsten we geen haakjes rondom het woord. Dit is omdat we het niet als functienaam wilden gebruiken.

Wanneer `fill-column` het eerste of enige element van een lijst was, zou de Lisp interpreter proberen de er aangekoppelde functiedefinitie te vinden. Maar `fill-column` heeft geen functiedefinitie. Probeer dit te evalueren:

```
(fill-column)
```

Je maakt een `*Backtrace*` buffer dat het volgende toont:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-function fill-column)
  (fill-column)
  eval((fill-column) nil)
  elisp--eval-last-sexp(nil)
  eval-last-sexp(nil)
  funcall-interactively(eval-last-sexp nil)
  call-interactively(eval-last-sexp nil nil)
  command-execute(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(Onthoud, om de debugger te sluiten en het debugger window te laten verdwijnen typ je `q` in het `*Backtrace*` buffer.)

1.7.2 Foutmelding voor een symbool zonder waarde

Wanneer je een symbool dat geen waarde aan zich gekoppeld heeft probeert te evalueren, dan krijg je een foutmelding. Dit kan je zien door te experimenteren met onze 2 plus 2 optelling. Zet de cursor meteen achter de `+`, voor het eerste getal 2 in de volgende expressie, en typ `C-x C-e`:

```
(+ 2 2)
```

In GNU Emacs 22 maak je een `*Backtrace*` buffer dat het volgende toont:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-variable +)
  eval(+)
  elisp--eval-last-sexp(nil)
  eval-last-sexp(nil)
  funcall-interactively(eval-last-sexp nil)
  call-interactively(eval-last-sexp nil nil)
  command-execute(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(Je sluit de debugger opnieuw door `q` in het `*Backtrace*` buffer te typen.)

Deze backtrace verschilt van de allereerste foutmelding die we zagen, die toonde '`Debugger entered--Lisp error: (void-function dit)`'. In dit geval heeft de functie geen waarde als variabele, terwijl in de andere foutmelding, de functie (het woord '`dit`') geen definitie had.

Wat we in het experiment met de `+` deden liet de Lisp interpreter de `+` evaluaren en zoeken naar de waarde van de variabele in plaats van de functiedefinitie. We deden dit door de cursor meteen achter het symbool te plaatsen in plaats van achter het haakje van de omsluitende lijst, zoals we eerder deden. Als gevolg hiervan evalueerde de Lisp interpreter de voorafgaande s-expressie, in dit geval `+` op zichzelf.

Omdat `+` geen waarde aan zich gekoppeld heeft en alleen een functiedefinitie, rapporteerde de foutmelding dat de waarde van het symbool als variabele leeg was.

1.8 Argumenten

Laten we nog eens naar ons oude voorbeeld kijken om te zien hoe informatie tussen functies doorgegeven wordt, de optelling van twee plus twee. In Lisp schrijven we dit als volgt:

```
(+ 2 2)
```

Wanneer je deze expressie evalueert verschijnt het getal in je echogebied. Wat de Lisp interpreter doet is het optellen van de getallen die volgen na de `+`.

De getallen toegevoegd aan `+` heten de *argumenten* van de functie `+`. Deze getallen vormen de informatie die *doorgegeven* wordt aan de de functie.

Het woord "argument" komt van de manier waarop het gebruikt wordt in wiskunde en refereert niet naar een dispuut tussen twee mensen. In plaats daarvan refereert het naar de informatie gegeven aan een functie, in dit geval aan de `+`. In Lisp zijn de argumenten van een functie de atomen of lijsten die volgen op de functie. De waarde die de evaluatie van deze atomen of lijsten teruggeeft worden doorgegeven aan de functie. Verschillende functies vereisen een verschillend aantal argumenten, sommige functies vereisen er geen.[3]

[3] Het is interessant om het pad te volgen dat voor het woord "argument" tot twee verschillende betekenissen leidde, eentje in wiskunde en eentje in dagelijks Engels. Volgens de *Oxford English Dictionary* is het woord afgeleid van het Latijn voor '`duidelijk maken, bewijzen`'. Via één tak van de afleiding kreeg het de betekenis "het bewijsmateriaal aangereikt als bewijs", met andere woorden "de aangeboden informatie", wat tot de betekenis van het woord in Lisp leidde. In de andere tak van de afleiding kreeg

1.8.1 Data typen van argumenten

Het type data dat aan een functie moet worden doorgegeven hangt af van het soort informatie dat die gebruikt. De argumenten voor een functie zoals `+` moeten waarden hebben die getallen zijn, omdat `+` getallen optelt. Andere functies gebruiken andere vormen van data voor hun argumenten.

De functie `concat` bijvoorbeeld koppelt of verbindt twee of meer tekstreeksen om een string te produceren. De argumenten zijn strings. Het concatenaten van de twee tekenreeksen `abc` en `def` produceert de enkele string `abcdef`. Dit is te zien door het volgende te evalueren:

```
(concat "abc" "def")
```

De waarde die het evalueren van deze expressie produceert is `"abcdef"`.

Een functie zoals `substring` gebruikt zowel een string en getallen als argumenten. De functie geeft een van de string terug, een *substring* van zijn eerste argument. Deze functie accepteert drie argumenten. Het eerste argument is de tekenreeks, het tweede en derde argument zijn de getallen die het begin (inclusief) en het einde (exclusief) van de substring aanduiden. De getallen zijn een telling van het aantal karakters (inclusief spaties en leestekens) vanaf het begin van de string. Merk op dat de karakters vanaf nul zijn genummerd, niet vanaf één.

Bijvoorbeeld, als je het volgende evalueert:

```
(substring "The quick brown fox jumped." 16 19)
```

dan zie je `"fox"` in het echogebied verschijnen. De argumenten zijn de string en twee getallen.

Merk op dat de aan `substring` doorgegeven string een enkelvoudig atoom is, alhoewel het is opgebouwd uit meerdere woorden gescheiden door spaties. Lisp rekent alles tussen de twee aanhalingstekens als onderdeel van de string, inclusief de spaties. Je kunt de `substring` functie als een soort van atoombreker zien, omdat het een anders ondeelbaar atoom neemt en daar een onderdeel uit haalt. `substring` is echter alleen in staat een substring uit een argument te halen dat een string is, niet van ander soort atoom zoals een getal of een symbool.

1.8.2 Een argument als de waarde van een variabele of lijst

Een argument kan een symbool zijn die een waarde teruggeeft wanneer het wordt geëvalueerd. Wanneer bijvoorbeeld het symbool `fill-column` op zichzelf wordt geëvalueerd, geeft het een getal terug. Dit getal kan in een optelling worden gebruikt.

---

het de betekenis “iets beweren op een manier waarop anderen een tegenbewering maken”, wat leidde tot de betekenis van het woord als dispuut. (Merk op dat het Engelse woord hier tegelijk aan twee betekenissen zich gekoppeld heeft. In tegenstelling kan in Emacs Lisp een symbool geen twee verschillende functiedefinities tegelijk kan hebben.)

Plaats de cursor achter de volgende expressie en typ *C-x C-e*:

```
(+ 2 fill-column)
```

De waarde is een getal, twee groter dan wat je krijgt door het evalueren van alleen `fill-column`. Voor mij is dit 74, omdat mijn waarde van `fill-column` 72 is.

Zoals we zojuist zagen kan een argument een symbool zijn dat een waarde teruggeeft wanneer het geëvalueerd wordt. Daarnaast kan een argument een lijst zijn die een waarde teruggeeft wanneer die geëvalueerd wordt. Bijvoorbeeld in de volgende expressie zijn de argumenten voor de functie `concat` de strings `"De "` en `" rode vossen."` en de lijst `(number-to-string (+ 2 fill-column))`.

```
(concat "De " (number-to-string (+ 2 fill-column)) " rode vossen.")
```

Wanneer je deze expressie evalueert—en als, zoals met mijn Emacs, `fill-column` naar 72 evalueert—verschijnt `"De 74 rode vossen."` in het echogebied. (Merk op dat je spaties achter het woord '`De`' en voor het voord '`rode`' zet zodat deze in de uiteindelijke string verschijnen. De functie `number-to-string` converteert de integer die de optelling teruggeeft naar een string. `number-to-string` is ook bekend als `int-to-string`.)

1.8.3 Variabel aantal argumenten

Sommige functies, zoals `concat`, `+` of `*` accepteren elk aantal argumenten. (De `*` is het symbool voor vermenigvuldiging.) Dit is te zien door het op de gebruikelijke manier evalueren van elk van de volgende expressies. Wat je in het echogebied ziet verschijnen is deze tekst achter '⇒' wat je kunt lezen als "evalueert naar".

In de eerste groep hebben de functies geen argumenten:

```
(+)       ⇒ 0

(*)       ⇒ 1
```

In deze groep hebben de functies elk één argument:

```
(+ 3)     ⇒ 3

(* 3)     ⇒ 3
```

In deze groep hebben de functies elk drie argumenten:

```
(+ 3 4 5) ⇒ 12

(* 3 4 5) ⇒ 60
```

1.8.4 Het verkeerde object type als argument gebruiken

Wanneer aan een functie een argument van een verkeerde type wordt doorgegeven, produceert de Lisp interpreter een foutmelding. Bijvoorbeeld verwacht de `+` functie dat de waarden van zijn argumenten getallen zijn. Als experiment kunnen we het gequote symbool `hello` in plaats van een getal doorgeven. Plaats de cursor achter de volgende expressie en typ *C-x C-e*:

```
(+ 2 'hello)
```

Wanneer je dit doet dan genereer je een foutmelding. Wat gebeurde is dat `+` geprobeerd het 2 op te tellen bij de waarde teruggegeven door `'hello`, maar de waarde teruggegeven door `'hello` is het symbool `hello`, geen getal. Alleen getallen kunnen worden opgeteld. Daardoor kon `+` zijn optelling niet uitvoeren.

Je maakt en betreedt een `*Backtrace*` buffer die toont:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error:
         (wrong-type-argument number-or-marker-p hello)
  +(2 hello)
  eval((+ 2 'hello) nil)
  elisp--eval-last-sexp(t)
  eval-last-sexp(nil)
  funcall-interactively(eval-print-last-sexp nil)
  call-interactively(eval-print-last-sexp nil nil)
  command-execute(eval-print-last-sexp)
---------- Buffer: *Backtrace* ----------
```

Zoals gebruikelijk probeert de foutmelding behulpzaam te zijn en klinkt logisch nadat je geleerd hebt hoe die te lezen.[4]

Het eerste deel van de foutmelding is eenvoudig, het toont '`wrong type argument`'. Daarna komt in mysterieus jargon het woord '`number-or-marker-p`'. Dit woord probeert je aan te geven welk type argument de `+` verwachtte.

Het symbool `number-or-marker-p` geeft aan dat de Lisp interpreter probeert vast te stellen of de informatie die het kreeg een getal is of een marker (een speciaal object dat een bufferpositie vertegenwoordigt). Wat het doet is testen of de `+` getallen kreeg om op te tellen. Ook test of het argument iets is dat een marker genoemd wordt, wat een specifieke eigenschap van Emacs Lisp is. (In Emacs worden lokaties in een buffer geregistreerd als markers. Wanneer de mark is gezet met het `C-@` of `C-SPC` commando, wordt zijn positie bewaard als een marker. De mark kan worden beschouwd als een getal–het aantal karakters van de lokatie ten opzichte van het begin van de buffer.) In Emacs Lisp kan `+` gebruikt worden om de numerieke waarde van de marker-posities als getallen op te tellen.

De '`p`' in `number-or-marker-p` is de belichaming van een praktijk die startte in de beginjaren van Lisp programmeren. De '`p`' staat voor *predicate*. In het jargon dat de eerste Lisp onderzoekers gebruikten, refereert een predicate aan een functie die bepaalt of een bepaalde eigenschap waar of onwaar is. De '`p`' vertelt ons dus dat `number-or-marker-p` de naam van een functie is die bepaalt of het waar of onwaar is dat het doorgegeven argument een getal of een marker is. Andere Lisp symbolen die eindigen in '`p`' zijn onder andere `zerop`, een functie die test of zijn argument de waarde nul heeft, en `listp`, een functie die test of zijn argument een lijst is.

Het laatste deel van de foutmelding tenslotte is het symbool `hello`. Dit is de waarde van het argument dat was doorgegeven aan `+`. Wanneer het juiste type object aan de optelling was doorgegeven, dan was de waarde die was doorgegeven een getal, zoals 37, in plaats van een symbool zoals `hello`. Maar dan had je geen foutmelding gekregen.

1.8.5 De `message` functie

Net als `+` accepteert de `message` functie een variabel aantal argumenten. Het wordt gebruikt om berichten naar de gebruiker te sturen en is zo nuttig dat we het hier beschrijven.

[4] `(quote hello)` is een expansie van de afkorting `'hello`.

Een bericht wordt getoond in het echogebied. Je kunt bijvoorbeeld een bericht in je echogebied tonen door het evalueren van de volgende lijst:

```
(message "Dit bericht verschijnt in het echogebied!")
```

De hele string tussen de dubbele aanhalingstekens in een enkelvoudig argument en wordt volledig getoond. (Merk op dat in dit voorbeeld het bericht zelf in het echogebied wordt getoond tussen dubbele aanhalingstekens. Dat is omdat je de waarde ziet die de `message` functie teruggeeft. In de meeste toepassingen van `message` in programma's die je schrijft wordt de tekst getoond in het echogebied als zij-effect, zonder aanhalingstekens. Zie Sectie 3.3.1 "`vermenigvuldig-met-zeven` in detail", pagina 30, voor een voorbeeld hiervan).

Wanneer er echter een '`%s`' in de string tussen aanhalingstekens staat, toont de `message`-functie de '`%s`' niet als zodanig, maar kijkt naar het argument dat op de string volgt. Het evalueert het tweede argument en toont de waarde op de lokatie in de string waar de '`%s`' staat.

De kun je zien door de cursor achter de volgende expressie te plaatsen en *`C-x C-e`* te typen:

```
(message "De naam van dit buffer is: %s." (buffer-name))
```

In Info zal `"De naam van dit buffer is: *info*."` in het echogebied verschijnen. De functie `buffer-name` geeft de naam van het buffer terug als string, die de `message` functie invoegt op de plek van de `%s`.

Om de waarde als een integer te tonen gebruik je '`%d`' op dezelfde manier als '`%s`'. Bijvoorbeeld om een bericht in het echogebied te tonen dat de waarde van `fill-column` opgeeft, evalueer je het volgende:

```
(message "De waarde van fill-column is %d." fill-column)
```

Wanneer ik op mijn systeem de lijst evalueer, verschijnt `"De waarde van fill-column is 72."` in mijn echogebied[5]

Wanneer er meer dan een '`%s`' in de string tussen aanhalingstekens staat, wordt de waarde van het eerste argument dat volgt op de string tussen aanhalingstekens op de plaats van de eerste '`%s`' getoond, de waarde van het tweede argument op de plaats van de tweede '`%s`', enzovoorts.

[5] Je kunt trouwens `%s` gebruiken om een getal te tonen. Het is niet-specifiek. `%d` toont alleen het deel van het getal dat links van het decimale teken staan, en niets dat geen getal is.

Wanneer je bijvoorbeeld het volgende evalueert,

```
(message "Er zijn %d %s in het kantoor!"
         (- fill-column 14) "paarse olifanten")
```

zal een nogal eigenaardig bericht in je echogebied verschijnen. Op mijn systeem zegt het `"Er zijn 58 paarse olifanten in het kantoor!"`.

De expressie `(- fill-column 14)` wordt geëvalueerd and het resulterende getal wordt ingevoegd op de plaats van '`%d`' en de string in dubbele aanhalingstekens `"paarse olifanten"` wordt behandeld als een enkelvoudig argument ingevoegd o de plaats van '`%s`'. (Dat wil zeggen dat een string tussen dubbele aanhalingstekens naar zichzelf evalueert, net als een getal.

Tenslotte is hier een enigszins complex voorbeeld dat niet alleen de berekening van een getal illustreert, maar ook hoe je een expressie binnen een expressie gebruikt om een tekst te genereren die '`%s`' vervangt:

```
(message "He saw %d %s"
         (- fill-column 32)
         (concat "red "
                 (substring
                  "The quick brown foxes jumped." 16 21)
                 " leaping."))
```

In dit voorbeeld heeft `message` drie argumenten: de string `"He saw %d %s"`, de expressie `(- fill-column 32)` en de expressie beginnend met de functie `concat`. De waarde resulterend van de evaluatie van `(- fill-column 32)` wordt ingevoegd op de plaats van de '`%d`' en de waarde teruggeven door de expressie beginnend met `concat` wordt ingevoegd op de plaats van de '`%s`'.

Wanneer jouw fill column 70 is en je de expressie evalueert, verschijnt `"He saw 38 red foxes leaping."` in je echogebied.

1.9 De waarde van een variabele zetten

Er zijn verschillende manieren waarop een variabele een waarde kan krijgen. Een van de manieren is het gebruik van de speciale vorm `setq`. Een andere manier is het gebruik van `let` (zie Sectie 3.6 "let", pagina 33). (Het jargon voor dit proces is *bind* een variabele aan een waarde.)

De volgende paragrafen beschrijven niet alleen hoe `setq` werkt maar illustreren tevens hoe argumenten worden doorgegeven.

1.9.1 Gebruik van `setq`

Om de waarde van het symbool `bloemen` op de lijst `roos viool madelief boterbloem` te zetten evalueer je de volgende expressie door de cursor achter de expressie te zetten en *C-x C-e* te typen:

```
(setq bloemen '(roos viool madelief boterbloem))
```

De lijst `(roos viool madelief boterbloem)` verschijnt in het echogebied. Dat is wat is *teruggegeven* door de speciale vorm `setq`. Als zij-effect is het symbool `bloemen` aan de lijst gebonden. Dat wil zeggen dat het symbool `bloemen`, welke als een variabel gezien kan worden, de lijst als waarde krijgt. (Overigens illustreert dit proces hoe een zij-effect voor de Lisp interpreter, de waarde zetten, het primaire

effect kan zijn waar wij mensen in geïnteresseerd zijn.. Dit is omdat elke Lisp functie een waarde moet teruggeven wanneer het geen fout krijgt, maar alleen een zij-effect heeft wanneer die is ontworpen om er een te hebben.

Nadat je de `setq` expressie hebt geëvalueerd, kun je het symbool `bloemen` evalueren, dit geeft je de waarde terug die je zojuist gezet hebt. Hier is het symbool. Plaats je cursor er achter en typ *C-x C-e*.

```
bloemen
```

Wanneer je `bloemen` evalueert, verschijnt de lijst `(roos viool madelief boterbloem)` in het echogebied.

Overigens, wanneer je `'bloemen` evalueert, de variabele met een enkele quote er voor, dat zie je in het echogebied het symbool zelf, `bloemen`. Hier is het gequote symbool, zodat je het kunt proberen:

```
'bloemen
```

Als extra gemak staat `setq` je toe verschillende variablen naar verschillende waarden te zetten, in een enkele expressie.

Om de waarde van de variabele `carnivoren` met `setq` naar de lijst `'(leeuw tijger luipaard)` te zetten is de volgende expressie gebruikt:

```
(setq carnivoren '(leeuw tijger luipaard)
```

`setq` kan ook gebruikt worden om verschillende waarden naar verschillende variabelen toe te kennen. Het eerste argument wordt gebonden aan de waarde van het tweede argument, het derde argument aan de waarde van het vierde argument, enzovoorts. Bijvoorbeeld kan je het volgende gebruiken om een lijst van bomen aan het symbool `bomen` en een lijst van herbivoren aan het symbool `herbivoren` toe te kennen:

```
(setq bomen '(den spar eik esdoorn)
      herbivoren '(gazelle antiloop zebra)
```

(De expressie had net zo goed op een enkele regel gekund, maar het zou misschien niet op een bladzijde passen, en mensen vinden het makkelijker om mooi opemaakte lijsten te lezen).

Alhoewel ik de term “toekennen” heb gebruikt, is er een andere manier over de werking van `setq` te denken, en dat zo te zeggen dat `setq` het symbool laat *wijzen* naar de lijst. Deze laatste manier van denken is erg gebruikelijk en in de volgende hoofdstukken zullen we een symbool tegenkomen dat “pointer” als onderdeel van zijn naam heeft. De naam is gekozen omdat het symbool een waarde, in het bijzonder een lijst aan zich gebonden heeft, of, op een andere manier uitgedrukt, het symbool is ingesteld om naar de lijst te wijzen.

1.9.2 Tellen

Hier volgt een voorbeeld dat laat zien hoe je `setq` in een teller gebruikt. Je kunt dit gebruiken om te tellen hoe vaak een deel van je programma zichzelf herhaalt. Stel eerst een variabele op nul. Vervolgens tel je iedere keer dat het programma zich herhaalt een bij het getal op. Om dit te doen heb je een variabele nodig die dienst doet als teller, en twee expressies: een initiële `setq` expressie die de variabele instelt op nul, en een tweede `setq` expressie die elke keer dat die wordt geëvalueerd de teller verhoogt.

```
(setq counter 0)                ; Laten wij dit de initialisator noemen.

(setq counter (+ counter 1))    ; Dit is de incrementer.

counter                         ; Dit is de teller.
```

(De tekst die volgt na de ‘;’ zijn comments. Zie Sectie 3.2.1 “Wijzig een functiedefinitie”, pagina 29.)

Wanneer je de eerste van deze drie expressies evalueert, de initialisator, `(setq counter 0)`, en daarna evalueer je de derde expressie, `counter`. dan verschijnt het getal `0` in het echogebied. Wanneer je de tweede expressie, de incrementer, `(setq counter (+ counter 1))`, evalueert, dan verschijnt het getal `1` in het echogebied. Elke keer dat je de tweede expressie evalueert, wordt de waarde van de counter verhoogd.

Wanneer je de incrementer, `(setq counter (+ counter 1))` evalueert, evalueert de Lisp interpreter eerst de meest binnenste lijst, dit is de optelling. Om deze lijst te evalueren, moet het de variabele `counter` en het getal `1` evalueren. Wanneer het de variabele `counter` evalueert, krijgt het diens huidige waarde. Het geeft deze waarde en het getal `1` door aan de `+`, die ze bij elkaar optelt. De som wordt teruggegeven als de waarde van de binnenste lijst en doorgegeven aan de `setq`, die de variabele `counter` op deze nieuwe waarde instelt. Zo is de waarde van de variabele `counter` gewijzigd.

1.10 Samenvatting

Lisp leren is net zo iets als een heuvel beklimmen, waar het eerste deel het steilste is. Je hebt inmiddels het moeilijkste deel beklommen, wat overblijft wordt makkelijker naarmate je verder komt.

Samenvattend,

- Lisp programma’s zijn gemaakt van expressies, elke lijsten of enkelvoudige atomen zijn.
- Lijsten zijn gemaakt van nul of meer atomen of binnenliggende lijsten, gescheiden door witte ruimte en omgeven door haakjes. Een lijst kan leeg zijn.
- Atomen zijn multi-karakter symbolen, zoals `forward-paragraph`, enkele-karakter symboleen zoals `+`, strings of karakters tussen dubbele aanhalingstekens, of getallen.
- Een getal evalueert naar zichzelf.
- Een string tussen dubbele aanhalingstekens evalueert ook naar zichzelf.
- Wanneer je een symbool op zichzelf evalueert, wordt zijn waarde teruggegeven.
- Wanneer je een lijst evalueert, kijkt de Lisp interpreter naar het eerste symbool in de lijst en vervolgens naar de functiedefinitie die aan dat symbool gebonden is. Daarvan worden de instructies in de functiedefinitie uitgevoerd.
- Een enkele quote ‘`'`’ vertelt de Lisp interpreter dat het de daaropvolgende expressie moet teruggeven zoals die is geschreven, en niet evalueren alsof de quote er niet was.

- Argumenten vormen de informatie die aan een functie wordt doorgegeven. De argumenten voor een functie worden berekend door de rest van de elementen van de lijst te evalueren, waarvan de functie het eerste element is.
- Een functie geeft altijd een waarde terug wanneer die wordt geëvalueerd (tenzij het een fout krijgt). Ook kan die een actie uitvoeren dat een zij-effect is. In veel gevallen is het creëren van zij-effecten het primaire doel.

1.11 Oefeningen

Enkele simpele oefeningen:

- Genereer een foutmelding door een geschikt symbool te evalueren dat niet tussen haakjes staat.
- Genereer een foutmelding door een geschikt symbool te evalueren dat tussen haakjes staat.
- Maak een teller die met twee verhoogt in plaats van een.
- Schrijf een expressie die een boodschap in het echogebied toont wanneer die wordt geëvalueerd.

2 Oefenen van evaluatie

Voordat je leert hoe je een functiedefinitie in Emacs Lisp schrijft, is het zinnig om wat tijd te besteden aan het evalueren van diverse expressies die al zijn geschreven. Deze expressies zijn lijsten met de functies als hun eerste (en vaak enige) element. Omdat sommige buffer-geassocieerde functies zowel eenvoudig als interessant zijn, beginnen we met die. In deze sectie evalueren we enkele daarvan. In een andere sectie bestuderen we de code van verscheidene andere buffer-gerelateerde functies om te zien hoe die zijn geschreven.

Telkens wanneer je een edit-commando aan Emacs Lisp geeft, zoals het commando om de cursor te verplaatsen of het scherm te scrollen, *evalueer je een expressie*, waarvan het eerste element een functie is. *Dit is hoe Emacs werkt.*

Wanneer je toetsen aanslaat laat je de Lisp interpreter een expressie evalueren en zo krijg je je resultaten. Zelfs het typen van platte tekst houdt het evalueren van een Emacs Lisp functie in, in dit geval eentje dat het `self-insert-command` gebruikt, die eenvoudig het karakter dat je typte invoegt. De functies die je evalueert met het typen van toetsaanslagen worden *interactief* genoemd, of *commando's*. Hoe je een functie interactief maakt wordt in het volgende hoofdstuk geïllustreerd. Zie Sectie 3.3 "Een functie interactief maken", pagina 29.

Naast het typen van toetsenbord commando's, hebben we een tweede manier gezien om een expressie te evalueren: door de cursor achter de lijst te plaatsen en `C-x C-e` te typen. Dit is wat we in rest van deze sectie zullen doen. Er zijn nog meer manieren om een expressie te evalueren, deze zullen we beschrijven wanneer die aan de orde komen.

Naast dat we ze voor evaluatie-oefeningen gebruiken, zijn de functies die in volgende secties behandeld worden op zichzelf belangrijk. Het bestuderen van deze functies verheldert het verschil tussen buffers en bestanden, hoe tussen buffer te schakelen, en hoe een lokatie er binnen vast te stellen.

2.1 Buffer namen

De twee functies `buffer-name` en `buffer-file-name` tonen het verschil tussen een buffer en een bestand. Wanneer je volgende expressie evalueert, `(buffer-name)`, verschijnt de naam van het buffer in het echogebied. Wanneer je `(buffer-file-name)` evalueert, verschijnt de naam van het bestand waar het buffer naar refereert in het echogebied. Meestal is de naam die `(buffer-name)` teruggeeft hetzelfde als de naam van het bestand waarnaar het refereert, en is de naam die `(buffer-file-name)` teruggeeft de volledige padnaam van het bestand.

Een bestand en een buffer zijn twee verschillende entiteiten. Een bestand is informatie permanent opgeslagen op de computer (tenzij je het verwijdert). Een buffer daarentegen is informatie binnen Emacs, die verdwijnt aan het einde van de editing sessie (of wanneer je het buffer killt). Meestal bevat een buffer informatie die je van een bestand gekopieerd hebt, we zeggen dat het buffer het bestand *bezoekt*. Deze kopie is waar je op werkt en wijzigt. Veranderingen in het buffer wijzigen het bestand niet, totdat je het buffer opslaat. Wanneer je het buffer opslaat, wordt het buffer gekopieerd naar het bestand en zo permanent opgeslagen.

Wanneer je dit in Info leest binnen Emacs, kan je elk van de volgende expressies evalueren door de cursor er achter te zetten en `C-x C-e` te typen.

```
(buffer-name)

(buffer-file-name)
```

Wanneer ik dit in Info doe, geeft evaluatie van `(buffer-name)` de waarde `"*info*"` terug, en is de waarde die de evaluatie van `(buffer-file-name)` teruggeeft `nil`.

Anderzijds, terwijl ik dit document schrijf is de waarde die het evalueren van `(buffer-name)` teruggeeft, `"introduction.texinfo"` en de waarde die de evaluatie van `(buffer-file-name)` teruggeeft `"/gnu/work/intro/introduction.texinfo"`.

De eerste is de naam van het buffer en het tweede is de naam van het bestand. In Info is de naam van het buffer `"*info*"`. Info wijst niet naar een bestand, waardoor `nil` het resultaat van de evaluatie `(buffer-file-name)` is. Het symbool `nil` is van het Latijnse woord voor “niets”, in dit geval betekent het dat het buffer met geen enkel bestand geassocieerd is. (In Lisp betekent `nil` ook “onwaar” en is synoniem voor de lege lijst, `()`.)

Wanneer ik schrijf is de naam van mijn buffer `"introduction.texinfo"`. De naam van het bestand waar het naar wijst is `"/gnu/work/intro/introduction.texinfo"`.

(In de expressies vertellen de haakjes aan de Lisp interpreter om `buffer-name` en `buffer-file-name` als functies te behandelen. Zonder de haakjes zou de interpreter proberen de symbolen als variabelen te behandelen. Zie Sectie 1.7 “Variabelen”, pagina 9.)

Ondanks het onderscheid tussen bestanden en buffers zie je vaak dat mensen aan een bestand refereren wanneer ze een buffer bedoelen en andersom. Inderdaad zeggen de meeste mensen “ik edit een bestand” in plaats van “ik edit een buffer dat ik binnenkort naar een bestand ga saven”. Het is vrijwel altijd duidelijk van de context wat mensen bedoelen. Wanneer we met computerprogramma’s werken is het belangrijk het onderscheid in gedachten te houden, want computers zijn niet zo slim als mensen.

Overigens komt het woord “buffer” van de betekenis van het woord als een “kussen dat de kracht van een botsing opvangt”. In de eerste computers verzachtte een buffer de interactie tussen bestanden en de centrale verwerkingseenheid. De drums of tapes die het bestand bevatten en de centrale verwerkingseenheid waren onderdelen van de apparatuur die zeer verschillend van elkaar waren en op hun eigen snelheid werkten, in stoten. Uiteindelijk groeide het buffer van een intermediair, een tijdelijke opslagplaats uit naar de plaats waar het werk wordt gedaan. Deze transformatie is ongeveer zoals een kleine zeehaven uitgroeide naar een grote stad. Eens was het slechts de plaats waar lading tijdelijk in magazijnen werd opgeslagen voordat het op schepen werd verladen en daarna een zakelijk en cultureel centrum op zichzelf werd.

Niet alle buffers zijn geassocieerd met bestanden. Bijvoorbeeld een `*scratch*`-buffer bezoekt geen enkel bestand. Op dezelfde manier is het `*Help*`-buffer met geen enkel bestand geassocieerd.

Wanneer je vroeger geen `~/.emacs` bestand had en een Emacs sessie startte door het typen van alleen het commando `emacs`, zonder enig bestand te noemen, startte Emacs met het `*scratch*` buffer zichtbaar. Tegenwoordig zie je een startscherm. Je kunt een van de commando's volgen die op het startscherm gesuggereerd worden, een bestand bezoeken of op `q` drukken om het startscherm te sluiten en het `*scratch*`-buffer te bereiken.

Schakel over naar het `*scratch*`-buffer en typ `(buffer-name)`, plaats de cursor direct er achter, en typ vervolgens `C-x C-e` om de expressie te evalueren. De naam `"*scratch*"` wordt teruggegeven en verschijnt in het echogebied. `"*scratch*"` is de naam van het buffer. Wanneer je `(buffer-file-name)` in het `*scratch*` buffer typt en dat evalueert, verschijnt `nil` in het echogebied, net zoals dat gebeurt wanneer je `(buffer-file-name)` in Info evalueert.

Overigens, als je in het `*scratch*` buffer bent en door een expressie teruggeven waarde in het `*scratch*` wilt laten verschijnen in plaats van het echogebied, typ je `C-u C-x C-e` in plaats van `C-x C-e`. Dit zorgt dat de teruggegeven waarde achter de expressie verschijnt. Het buffer ziet er zo uit:

```
(buffer-name)"*scratch*"
```

Je kunt dit niet in Info doen, omdat Info read-only is en je niet toestaat de inhoud van het buffer te wijzigen. Maar je kunt dit in elk buffer doen dat je kunt editen. Wanneer je code schrijft of documentatie (zoals dit boek), is deze eigenschap erg nuttig.

2.2 Buffers pakken

De `buffer-name`-functie geeft de *naam* van het buffer. Om het buffer *zelf* te krijgen is een andere functie benodigd, de functie `current-buffer`. Wanneer je deze functie in code gebruikt krijg je het buffer zelf.

Een naam en het object of entiteit waar de naam aan refereert zijn van elkaar verschillend. Jij bent niet je naam. Je bent een persoon die anderen bij naam noemen. Wanneer je vraagt om met George te spreken en iemand geeft je een kaart met de letters '`G`', '`e`', '`o`', '`r`', '`g`' en '`e`', dan kan dat je amuseren, maar je zal er niet tevreden mee zijn. Je wilt niet met de naam spreken, maar met de persoon die met die naam genoemd is. Een buffer is vergelijkbaar: de naam van het scratch-buffer is `*scratch*`, maar de naam is niet het buffer. Om het buffer zelf te krijgen, heb je een functie nodig zoals `current-buffer`.

Er is echter een kleine complicatie: wanneer je `current-buffer` zelf evalueert in een expressie, zoals we hier gaan doen, dan wordt de de representatie van de naam van het buffer getoond zonder de inhoud van het buffer. Emacs werkt op deze manier wegens twee redenen: het buffer kan duizenden regels lang zijn—te lang om handig weer te geven, en een ander buffer kan dezelfde inhoud maar een andere naam hebben en het is belangrijk om onderscheid tussen hen te maken.

Hier is een expressie die de functie bevat:

```
(current-buffer)
```

Wanneer je deze expressie in Info in Emacs op de gebruikelijke manier evalueert, verschijnt `#<buffer *info*>` in het echogebied. Het speciale formaat geeft aan dat het buffer zelf is teruggegeven in plaats van slechts de naam.

Overigens, hoewel je een getal of symbool in een programma kunt typen, kan je dit niet doen met de getoonde representatie van een buffer: de enige manier om een buffer zelf te krijgen is met een functie zoals `current-buffer`.

Een gerelateerde functie is `other-buffer`. Deze geeft de meest recent geselecteerde buffer terug, anders dan degene waar je nu in bent, en niet de getoonde representatie van de naam. Wanneer je recentelijk heen en weer tussen het `*scratch*` buffer geschakeld hebt, dan zal `other-buffer` dat buffer teruggeven.

Dit zie je door de volgende expressie te evalueren:

```
(other-buffer)
```

Je zou `#<buffer *scratch*>` moeten zien verschijnen in het echogebied, of de naam van welk ander buffer je het meest recent van teruggeschakeld hebt.[1]

2.3 Van buffer verwisselen

De `other-buffer`-functie biedt eigenlijk een buffer wanneer het gebruikt wordt als een argument voor een functie die er een vereist. We zien dit door `other-buffer` en `switch-to-buffer` te gebruiken om naar een ander buffer te schakelen.

Maar eerst een korte introductie van de `switch-to-buffer` functie. Toen je heen en weer schakelde tussen Info en het `*scratch*`-buffer om `(buffer-name)` te evalueren, heb je hoogstwaarschijnlijk *`C-x b`* en daarna `*scratch*` getypt[2] toen je in het minibuffer gevraagd werd naar de naam van het buffer waar naar toe te schakelen. De toetsaanslagen *`C-x b`* laten de Lisp interpreter de interactieve `switch-to-buffer`-functie evalueren. Zoals we eerder gezegd hebben, dit is hoe Emacs werkt: verschillende toetsaanslagen roepen verschillende functies aan of draaien die. Bijvoorbeeld *`C-f`* roept `forward-char` aan, *`M-e`* roept `forward-sentence` aan, enzovoorts.

Door `switch-to-buffer` in een expressie te schrijven en het een buffernaam te geven om naar te schakelen, kunnen we buffers schakelen op dezelfde manier zoals *`C-x b`* doet:

```
(switch-to-buffer (other-buffer))
```

Het symbool `switch-to-buffer` is het eerste element in de lijst, dus zal de Lisp interpreter het als een functie behandelen en de instructies uitvoeren die daaraan gekoppeld zijn. Maar voordat die dat doet, merkt de interpreter op dat `other-`

[1] Eigenlijk wanneer het buffer waar je laatst vandaan schakelde voor je zichtbaar is in een ander venster. zal standaard `other-buffer` het meest recente buffer kiezen dat je niet kunt zien. Dit is een subtiliteit die ik meestal vergeet.

[2] Of beter gezegd, om toetsaanslagen te besparen, heb je waarschijnlijk alleen *`RET`* getypt indien het default buffer `*scratch*` was, of wanneer het niet zo was, je slechts een gedeelte van de naam typte, zoals `*sc`, en de *`TAB`*-toets gebruikte zodat het expandeerde naar de volledige naam, en daarna *`RET`* typte.

`buffer` tussen haakjes staat en eerst dat symbool verwerkt. `other-buffer` is het eerste (en in dit geval enige) element in deze lijst, dus de Lisp interpreter roept of runt deze functie. Het geeft een ander buffer terug. Vervolgens runt de interpreter `switch-to-buffer` en geeft het andere buffer als argument door, waar Emacs naar zal omschakelen. Wanneer je dit in Info leest, probeer het dan nu direct. Evalueer de expressie. (Typ *C-x b* `RET` om terug te gaan.)[3]

In de programmeervoorbeelden in volgende secties van dit document zie je de functie `set-buffer` vaker dan `switch-to-buffer`. Dit is wegens een verschil tussen computerprogramma's en mensen: mensen hebben ogen en verwachten het buffer te zien waarop ze werken op hun computerterminals. Dit is zo voor de hand liggend, het spreekt bijna voor zich. Echter, programma's hebben geen ogen. Wanneer een computerprogramma op een buffer werkt, dan hoeft dat buffer niet op het scherm zichtbaar te zijn.

`switch-to-buffer` is ontworpen voor mensen en doet twee verschillende dingen: het schakelt het buffer waar de aandacht van Emacs op is gevestigd en het schakelt het buffer dat getoond wordt in het venster naar een nieuw buffer. `set-buffer` anderzijds doet meer een enkel ding: het schakelt de aandacht van het computerprogramma naar een ander buffer. Het buffer op het scherm blijft onveranderd (uiteraard gebeurt er niets totdat het commando klaar is met lopen).

Ook hebben we net een ander jargon term geïntroduceerd, het woord *aanroepen*. Wanneer je een lijst evalueert waar het eerste symbool een functie is, dan roep je die functie aan. Het gebruik van deze term komt voort uit het begrip van de functie als entiteit dan iets voor je kan doen wanneer je het aanroept—net als een loodgieter een entiteit is die een lek kan repareren wanneer je hem of haar roept.

2.4 Buffergrootte en de lokatie van point

Laten wij tenslotte naar enkele nogal eenvoudige functies kijken, `buffer-size`, `point`, `point-min` en `point-max`. Deze geven informatie over de grootte van een buffer en de lokatie van point daarin.

De functie `buffer-size` vertelt je hoe groot het huidige buffer is. Dit houdt in dat de functie de som van het aantal karakters in het buffer teruggeeft.

```
(buffer-size)
```

Je kunt dit op de gebruikelijke manier evalueren door de cursor achter de expressie te plaatsen en *C-x C-e* te typen.

---

[3] Onthoudt, deze expressie zal je naar het meest recent geselecteerde buffer brengen, die je niet kunt zien. Wanneer je werkelijk naar je meest recent geselecteerde buffer wilt gaan, zelfs wanneer je die kunt zien, evalueer dan de volgende expressie:

```
(switch-to-buffer (other-buffer (current-buffer) t))
```

In dit geval vertelt het eerste argument van `other-buffer` welk buffer het moet overslaan—het huidige—en het tweede argument vertelt `other-buffer` dat het OK is om naar een zichtbaar buffer te schakelen. Bij normaal gebruik neemt `switch-to-buffer` je naar een buffer dat niet in vensters zichtbaar is omdat je hoogst waarschijnlijk *C-x o* (`other-window`) gebruikt om naar een ander zichtbaar buffer te gaan.

In Emacs wordt de huidige positie van de cursor *point* genoemd. De expressie `(point)` geeft een getal terug dat je vertelt waar de cursor is gepositioneerd als een telling van het aantal karakters vanaf het begin van het buffer tot point.

Je kunt de karaktertelling voor point in dit buffer zien door het op de gebruikelijke manier evalueren van de volgende expressie:

```
(point)
```

Terwijl ik dit schrijf is de waarde van point 65724. De `point` functie wordt vaak gebruikt in sommige voorbeelden later in dit boek.

De waarde van point hangt natuurlijk af van zijn lokatie in het buffer. Wanneer je point op deze plek evalueert, is het getal groter:

```
(point)
```

Voor mij is de waarde van point op deze plek 66043, wat betekent dat er 319 karakters (inclusief spaties) tussen de twee expressies zijn. (Ongetwijfeld zie je verschillende getallen, omdat ik dit heb geschreven heb na de eerste evaluatie van point.)

De functie `point-min` lijkt enigszins op `point`, maar het geeft de waarde van de minimum toegestane waarde van point in het huidige buffer. Dit is het getal 1 tenzij *versmallen* van toepassing is. (Versmallen is het mechanisme waarmee je jezelf of een programma beperkt tot operaties op slechts een deel van een buffer. Zie Hoofdstuk 6 “Versmallen en verbreden”, pagina 74.) Evenzo geeft de functie `point-max` de maximum toegestane waarde van point in het huidige buffer.

2.5 Oefening

Vindt een bestand waarmee je werkt en ga naar het midden. Vindt zijn buffernaam, bestandsnaam, lengte en je positie in het bestand.

3 Hoe functiedefinities te schrijven

Wanneer de Lisp interpreter een lijst evalueert, kijkt het of het eerste symbool in de lijst een functiedefinitie aan zich gekoppeld heeft, met andere woorden, of het symbool naar een functiedefinitie wijst. Een symbool met een functiedefinitie wordt eenvoudig een functie genoemd (alhoewel, eigenlijk is de definitie de functie en verwijst het symbool er naar).

Alle functies zijn beschreven in termen van andere functies, behalve enkele *primitieve* functies die in de C programmeertaal geschreven zijn. Wanneer je functiedefinities schrijft, schrijf je ze in Emacs Lisp en je gebruikt andere functies als bouwstenen. Sommige functies die je gebruikt zijn zelf geschreven in Emacs Lisp (misschien door jezelf) en sommige zijn primitieven geschreven in C. De primitieve functies worden precies zo gebruikt als de in Emacs Lisp geschreven functies en gedragen zich net zo. Zij zijn in C geschreven zodat we GNU Emacs makkelijk op elke computer kunnen draaien die voldoende krachtig is en C kan draaien.

Laat me dit nogmaals benadrukken: wanneer je code in Emacs Lisp schrijft, maak je geen onderscheid tussen het gebruik van functies geschreven in C en functies geschreven in Emacs Lisp. Het verschil is irrelevant. Ik noem het onderscheid alleen omdat is interessant is het te weten. Inderdaad, tenzij je het onderzoekt, zou je niet weten of een reeds geschreven functie is geschreven in Emacs Lisp of C.

3.1 De `defun` macro

In Lisp heeft een symbool zoals `mark-whole-buffer` code aan zich gekoppeld die de computer vertelt wat te doen wanneer die functie wordt aangeroepen. Deze code heet de *functiedefinitie* en is gecreëerd door het evalueren van een Lisp expressie die start met het symbool `defun` (wat een afkorting is van *definieer functie*).

In volgende secties kijken we naar functiedefinities van de Emacs broncode, zoals `mark-whole-buffer`. In deze sectie beschrijven we een eenvoudige functiedefinitie zodat je kunt zien hoe die eruitzien. Dat voorbeeld gebruikt rekenkunde omdat het tot een eenvoudig voorbeeld leidt. Sommige mensen houden niet van voorbeelden met rekenkunde, echter wanneer je zo'n persoon bent, wanhoop niet. Nauwelijks enige code die we in de rest van deze introductie behandelen heeft met rekenkunde of wiskunde te maken. De voorbeelden hebben hoofdzakelijk op de een of andere manier met tekst te maken.

Een functiedefinitie bestaat uit maximaal vijf onderdelen volgend op het woord `defun`:

1. De naam van het symbool waaraan de functiedefinitie gekoppeld moet worden.
2. Een lijst met de argumenten die aan de functie zullen worden doorgegeven. Wanneer geen argumenten aan de functie zullen worden doorgegeven is dit een lege lijst, `()`.
3. Documentatie die de functie beschrijft. (Technisch optioneel, maar sterk aanbevolen.)

4. Optioneel een expressie die de functie interactief maakt zodat je die kunt gebruiken door *`M-x`* gevolgd door de naam van de functie te typen, of door het typen van een passende toets of toetscombinatie.
5. De code die de computer instrueert wat de doen de *body* van de functiedefinitie.

Het is nuttig om de vijf delen van een functiedefinitie te beschouwen als een sjabloon met vijf slots voor elk onderdeel:

```
(defun functienaam (argumenten...)
  "optionele-documentatie..."
  (interactive argument-doorgevende-info)      ; optioneel
  body...)
```

Als een voorbeeld is hier de code voor een functie die zijn argument met 7 vermenigvuldigt. (Het voorbeeld is niet interactief. Zie Sectie 3.3 “Een functie interactief maken”, pagina 29, voor die informatie.)

```
(defun vermenigvuldig-met-zeven (getal)
  "Vermenigvuldig GETAL met zeven."
  (* 7 getal))
```

Deze definitie begint met een haakje en het symbool `defun`, gevolgd door de naam van de functie.

De naam van de functie wordt gevolgd door een lijst die de argumenten bevat die aan de functie zullen worden doorgegeven. Deze lijst heet de *argumentlijst*. In dit voorbeeld heeft de lijst maar één element, het symbool `getal`. Wanneer de functie wordt gebruikt wordt het symbool gebonden aan de waarde die als argument van de functie is gebruikt.

In plaats van te kiezen voor het woord `getal` als naam van het argument had ik elke andere naam kunnen pakken. Bijvoorbeeld had ik kunnen kiezen voor het woord `multiplicant`. Ik pakte het woord “getal” omdat het vertelt wat voor soort waarde bedoeld is voor het slot, maar ik net zo goed het woord “multiplicant” kunnen kiezen om de rol aan te geven die de in dit slot geplaatste waarde speelt in de werking van de functie. Ik had het ook `foogle` kunnen noemen, maar dat zou een slechte keuze zijn omdat het niet aan mensen vertelt wat het betekent. De keuze van de naam is aan de programmeur en moet zo gekozen worden dat het de betekenis van de functie duidelijk maakt.

Je kunt inderdaad elke naam die je maar wilt kiezen voor een symbool in de argumentlijst, zelfs de naam van een symbool die in een andere functie gebruikt wordt: de naam die je gebruikt in de argumentlijst is privé voor die specifieke definitie. In die definitie refereert de naam naar een andere entiteit dan elk gebruik van de zelfde naam buiten de functiedefinitie. Stel je hebt de bijnaam “Kleintje” in je familie, dan refereren familieleden met “Kleintje” naar jou. Maar buiten jouw familie, bijvoorbeeld in een speelfilm, refereert de naam “Kleintje” naar iemand anders. Omdat de naam in de argumentlijst privé is voor de functiedefinitie kan je de waarde van zo’n symbool binnen de body van de functie wijzigen zonder de waarde buiten de functie te wijzigen. Het effect is vergelijkbaar met dat geproduceerd door de `let`-expressie. (Zie Sectie 3.6 “`let`”, pagina 33.)

De argumentlijst wordt gevolgd door de documentatiestring die de functie beschrijft. Dat is wat je ziet als je *`C-h f`* en de naam van de functie typt. Overigens,

wanneer je een documentatiestring als deze schrijft moet je van de eerste regel een complete zin maken, omdat sommige commando's, zoals `apropos`, alleen de eerste regel tonen van een meerregelige documentatiestring. Ook moet je de tweede regel van een documentatiestring, wanneer die er is, niet laten inspringen, omdat dit er raar uitziet wanner je *C-h f* (`describe-function`) gebruikt. De documentatiestring is optioneel, maar is zo nuttig, dat die deel van vrijwel elke functie die je schrijft zou moeten zijn.

De derde regel van ons voorbeeld bestaat uit de body van de functiedefinitie. (De meeste functiedefinities zijn natuurlijk langer dan deze.) In dit voorbeeld is de body de lijst `(* 7 getal)`, die zegt vermenigvuldig de waarde van *getal* met 7. (In Emacs Lisp, is `*` de functie voor vermenigvuldiging, net zoals `+` de functie voor optellen is.)

Wanneer je de functie `vermenigvuldig-met-zeven` gebruikt evalueert `getal` naar het daadwerkelijk getal dat je wilt gebruiken. Hier is een voorbeeld dat toont hoe je `vermenigvuldig-met-zeven` gebruikt, maar probeer het nu nog niet te evalueren!

```
(vermenigvuldig-met-zeven 3)
```

Het symbool `getal`, gespecificeerd in de functiedefinitie in de volgende sectie, is gebonden aan de waarde 3 in het daadwerkelijk gebruik van de functie. Merk op dat hoewel `getal` tussen haakjes in de functiedefinitie stond, het argument dat aan de functie `vermenigvuldig-met-zeven` wordt doorgegeven niet tussen haakjes staat. De haakjes worden in de functiedefinitie geschreven zodat de computer uit kan zoeken waar de argumentlijst eindigt en de rest van de functie begint.

Wanneer je dit voorbeeld evalueert, krijgt je waarschijnlijk een foutmelding. (Ga je gang, probeer het!) Dit is omdat we de functiedefinitie hebben geschreven maar de computer nog niet op de hoogte gebracht hebben van de definitie—wij hebben de functiedefinitie nog niet in Emacs geladen. Het installeren van een functie is het proces dat de Lisp interpreter op de hoogte brengt van de functie. Het installeren wordt in de volgende sectie beschreven.

3.2 Installeer een functiedefinitie

Wanneer je dit binnen Info in Emacs leest, kan je de functie `vermenigvuldig-met-zeven` proberen door eerst de functiedefinitie te evalueren en daarna `(vermenigvuldig-met-zeven 3)` te evalueren. Een exemplaar van de functiedefinitie volgt. Plaats de cursor achter het laatste haakje van de functiedefinitie en typ *C-x C-e*. Wanneer je dit doet, verschijnt `vermenigvuldig-met-zeven` in het echogebied. (Wat dit betekent is dat wanneer een functiedefinitie wordt geëvalueerd, de waarde die het teruggeeft de naam van de gedefinieerde functie is.) Tegelijkertijd installeert deze actie de functiedefinitie.

```
(defun vermenigvuldig-met-zeven (getal)
  "Vermenigvuldig GETAL met zeven."
  (* 7 getal))
```

Met het evalueren van deze `defun` heb je zojuist `vermenigvuldig-met-zeven` in Emacs geïnstalleerd. De functie is nu net zo goed onderdeel van Emacs als `forward-word`.(`vermenigvuldig-met-zeven` blijft geïnstalleerd totdat je Emacs afsluit. Om

de code automatisch opnieuw te laden wanneer je Emacs start, zie Sectie 3.5 "Installeer code permanent", pagina 32.)

Je kunt het effect van het installeren van `vermenigvuldig-met-zeven` zien dor het volgende voorbeeld te evalueren. Plaats de cursor achter de volgende expressie en typ *C-x C-e*. Het getal 21 verschijnt in het echogebied.

```
(vermenigvuldig-met-zeven 3)
```

Als je dat wilt kun je de documentatie van de functie lezen door het typen van *C-h f* (`describe-function`) en dan de naam van de functie `vermenigvuldig-met-zeven`. Wanneer je dat doet dan verschijnt een `*Help*`-venster op het scherm, dat zegt:

```
vermenigvuldig-met-zeven is an interpreted-function.

(vermenigvuldig-met-zeven GETAL)
Vermenigvuldig-met-zeven GETAL met zeven.
```

(Om terug te gaan naan een enkel venster op je scherm, typ *C-x 1*.)

3.2.1 Wijzig een functiedefinitie

Wanneer je de code in `vermenigvuldig-met-zeven` wilt wijzigen, herschrijf je die. Om een nieuwe versie in plaats van de oude te installeren evalueer je de functiedefinitie opnieuw. Dit is hoe je code in Emacs aanpast. Dat is erg eenvoudig.

Als een voorbeeld kan je de `vermenigvuldig-met-zeven`-functie wijzigen zodat die het getal zeven keer met zichzelf optelt, in plaats van het vermenigvuldigen van het getal met zeven. Het produceert hetzelfde antwoord, maar via een andere route. Tegelijkertijd voegen we commentaar toe aan de code. Commentaar is tekst die de Lisp interpreter negeert, maar die de menselijke lezer nuttig of verhelderend kan vinden. Het commentaar is dat dit de tweede versie is.

```
(defun vermenigvuldig-met-zeven (getal)        ; Tweede versie.
  "Multiply GETAL by seven."
  (+ getal getal getal getal getal getal getal))
```

Het commentaar volgt op een puntkomma, '`;`'. In Lisp is alles op een regel dat volgt achter een puntkomma commentaar. Het einde van de regel is het einde van het commentaar. Om het commentaar over twee of meer regels te spreiden, begin je elke regel met een puntkomma.

Zie Sectie 16.3 "Starten met een `.emacs` bestand", pagina 203, en Sectie "Comments" in *The GNU Emacs Lisp Reference Manual*, voor meer over commentaar.

Je kunt deze versie van `vermenigvuldig-met-zeven` installeren door het op dezelfde manier te evalueren als je de eerste functie geëvalueerd hebt: plaats de cursor achter het laatste haakje en typ *C-x C-e*.

Samengevat, dit is hoe je code in Emacs Lisp schrijft: je schrijft een functie, installeert die, test die en dan maak je verbeteringen of uitbreidingen en installeert die opnieuw.

3.3 Een functie interactief maken

Je maakt een functie interactief door een lijst die begint met de speciale vorm `interactive` direct achter de documentatie. Een gebruiker kan een interactieve

functie starten door het typen van *M-x* en daarna de naam van de functie, of door het typen van de toetscombinatie waaraan die is gebonden, bijvoorbeeld door het typen van *C-n* voor `next-line` of *C-x h* voor `mark-whole-buffer`.

Het is interessant dat wanneer je een interactieve functie interactief aanroept, de teruggegeven waarde niet automatisch in het echogebied wordt getoond. Dit is omdat je vaak een interactieve functie aanroept wegens de zij-effecten, zoals een woord of regel verder gaan, en niet voor de teruggegeven waarde. Wanneer de teruggegeven waarde elke keer dat je een toets aanslaat in het echogebied getoond zou worden, zou dat erg afleiden.

Zowel het gebruik van de speciale vorm `interactive` en een manier om de waarde in het echogebied te tonen kan door het maken van een interactieve versie van `vermenigvuldig-met-zeven` worden geïllustreerd.

Hier is de code:

```
(defun vermenigvuldig-met-zeven (getal)        ; Interactieve versie.
  "Vermenigvuldig GETAL met zeven."
  (interactive "p")
  (message "Het resultaat is %d" (* 7 getal)))
```

Je installeert deze code door de cursor er achter te plaatsen en *C-x C-e* te typen. De naam van de functie verschijnt in je echogebied. Vervolgens kan je deze code gebruiken door het typen van *C-u* en een getal, en daarna *M-x vermenigvuldig-met-zeven* te typen en op `RET` te drukken. De zin '`Het resultaat is ...`' gevolgd door het product verschijnt in het echogebied.

Meer algemeen gesproken roep je een functie zoals deze aan op een van de twee manieren:

1. Door het typen van een prefix-argument dat het door te geven getal bevat en daarna *M-x* en de naam van de functie te typen, zoals met *C-u 3 M-x forward-sentence*; of,
2. Door welke toets of toetscombinatie dan ook de functie aan gebonden is, zoals met *C-u 3 M-e*.

Beide zojuist genoemde voorbeelden werken identiek om point drie zinnen verder te plaatsen. (Omdat `vermenigvuldig-met-zeven` niet aan een toets gebonden is, kan dit niet gebruikt worden als voorbeeld van een key binding.)

(Zie Sectie 16.7 "Sommige key bindings", pagina 207, om te leren hoe je een commando aan een key bindt.)

Een *prefix-argument* wordt aan een interactieve functie gegeven door het aanslaan van de `META` toets gevolg door een cijfer, bijvoorbeeld *M-3 M-e*, of door het aanslaan van *C-u* en daarna een getal, bijvoorbeeld *C-u 3 M-e* (wanneer je *C-u* zonder getal aanslaat, is de standaard waarde 4).

3.3.1 Een interactieve `vermenigvuldig-met-zeven`

Laten we naar het gebruik van de speciale vorm `intertive` kijken en daarna naar de functie `message` in de interactieve versie van `vermenigvuldig-met-zeven`. Je herinnert dat de functiedefinitie er zo uitziet:

```
(defun vermenigvuldig-met-zeven (getal)        ; Interactieve versie.
  "Vermenigvuldig GETAL met zeven."
  (interactive "p")
  (message "Het resultaat is %d" (* 7 getal)))
```

In deze functie is de expressie `(interactive "p")` een lijst met twee elementen. De `"p"` vertelt Emacs om het prefix-argument door te geven aan de functie en zijn waarde te gebruiken voor het argument van de functie.

Het argument is een getal. Dit betekent dat het symbool `getal` wordt gebonden aan een getal in de regel:

```
(message "Het resultaat is %d" (* 7 getal))
```

Bijvoorbeeld wanneer je prefix-argument 5 is, zal de Lisp interpreter de regel evalueren alsof het was:

```
(message "Het resultaat is %d" (* 7 5))
```

(Wanneer je dit in GNU Emacs leest, kan je zelf deze expressie evalueren.) Eerst zal de interpreter de binnenste lijst evalueren, dus `(* 7 5)`. Dit geeft een waarde van 35 terug. Daarna zal het de buitenste lijst evalueren, en daarbij de waardes van tweede en volgende elementen van de lijst doorgeven aan de functie `message`.

Zoals we hebben gezien is `message` een Emacs Lisp functie specifiek ontworpen om een eenregelige boodschap naar de gebruiker te sturen. (Zie Sectie 1.8.5 “De `message` functie”, pagina 14.) Samengevat, de `message` functie toont zijn argumenten in het gebied zoals ze zijn, met uitzondering van gevallen zoals ‘`%d`’ of ‘`%s`’ (en diverse andere %-reeksen die we nog niet genoemd hebben). Wanneer het een control-reeks ziet, kijkt de functie naar het tweede en volgende argumenten en toont de waarde van de argumenten in de string op de plek waar de control-reeks staat.

In de interactieve `vermenigvuldig-met-zeven`-functie is de control-reeks ‘`%d`’, die een getal vereist, en de waarde die de evaluatie van `(* 7 5)` teruggeeft is het getal 35. Daarom wordt het getal 35 getoond op de plaats van ‘`%d`’ en is de boodschap ‘`Het resultaat is 35`’.

(Merk op dat wanneer je de functie `vermenigvuldig-met-zeven` aanroept, de boodschap zonder aanhalingstekens getoond wordt. Dit komt omdat de waarde die door `message` wordt teruggeven in het echogebied verschijnt wanneer je een expressie evalueert wiens eerste element `message` is, maar wanneer het in een functie is ingebed, toont `message` de tekst als een zij-effect zonder de aanhalingstekens.)

3.4 Verschillende opties voor `interactive`

In het voorbeeld gebruikte `vermenigvuldig-met-zeven "p"` als argument voor `interactive`. Dit argument vertelde Emacs om door wat je typte, hetzij *`C-u`* gevolgd door een getal of `META` gevold door een cijfer, te interpreteren als een commando om dat getal door te geven aan de functie als zijn argument. Emacs heeft meer dan twintig karakters voorgedefinieerd om met `interactive` te gebruiken. In vrijwel alle gevallen maakt een van deze opties het je mogelijk de juiste informatie interactief aan een functie door te geven. (Zie Sectie “Code Characters for `interactive`” in *The GNU Emacs Lisp Reference Manual*.)

Beschouw de functie `zap-to-char`. De interactieve expressie daarvan is

```
(interactive "p\ncZap to char: ")
```

Het eerste deel van het argument van `interactive` is '`p`', waarmee je al vertrouwd bent. Dit argument vertelt Emacs een prefix als getal te interpreteren om aan de functie door te geven. Je geeft het prefix op hetzij door `C-u` gevolgd door een getal te typen, of door het typen van `META` gevolgd door een cijfer. Het prefix is het aantal van het opgegeven karakter. Dus als je prefix drie is en het opgegeven karakter '`x`', dan verwijder je alle tekst tot en met de derde volgende '`x`'. Wanneer je geen prefix gebruikt dan verwijder je alle tekst tot en met het gespecificeerde karakter, maar niet meer.

De '`c`' vertelt de functie de naam van het karakter tot waar te verwijderen.

Meer formeel, een functie met twee of meer argumenten kan voor elk argument informatie doorgegeven krijgen door delen toe te voegen aan de string die volgt op `interactive`. Wanneer je dit doet is de informatie die doorgegeven wordt in dezelfde volgorde als gespecificeerd is de `interactive` lijst. In de string is elk deel gescheiden van het volgende deel door een '`\n`', wat een nieuwe regel is. Bijvoorbeeld, je kunt '`p`' opvolgen met een '`\n`' en een '`cZap to char: `'. Dit laat Emacs de waarde doorgeven van het prefix-argument (als er een is) en het karakter.

In dit geval ziet de functiedefinitie er als volgt uit, waarbij `arg` en `char` de symbolen zijn waaraan `interactive` het prefix-argument en het gespecificeerde karakter bindt:

```
(defun name-of-function (arg char)
  "documentation..."
  (interactive "p\ncZap to char: ")
  body-of-function...)
```

(De spatie achter de dubbele punt in de prompt maakt dat het er beter uitziet wanneer je wordt geprompt. Zie Sectie 5.1 "De definitie van `copy-to-buffer`", pagina 60, bijvoorbeeld.)

Wanneer een functie geen argumenten heeft, hoeft `interactive` er geen te hebben. Zo'n functie bevat de eenvoudige expressie `(interactive)`. De functie `mark-whole-buffer` is er zo een.

Daarnaast kun je, wanneer de speciale letter-codes niet geschikt zijn voor jouw applicatie, je eigen argumenten als lijst doorgeven aan `interactive`.

Zie Sectie 4.4 "De definitie van `append-to-buffer`", pagina 52, voor een voorbeeld. Zie Sectie "Using `Interactive`" in *The GNU Emacs Lisp Reference Manual*, voor een meer complete uitleg over deze techniek.

3.5 Code permanent installeren

Wanneer je een functiedefinitie installeert door het te evalueren, blijft het geïnstalleerd totdat je Emacs sluit. De volgende keer dat je een nieuwe sessie van Emacs start, is de functie niet geïnstalleerd tenzij je de functiedefinitie opnieuw evalueert.

Op een gegeven moment wil je de code automatisch geïnstalleerd hebben wanneer je een nieuwe Emacs sessie start. Er zijn verschillende manieren om dit te doen:

- Wanneer je code hebt die uitsluitend voor jezelf is, kan je de code voor de functiedefinitie opnemen in je `.emacs` initialisatiebestand.
- Een andere manier is de functiedefinities die je geïnstalleerd wilt hebben zelf in een of meer bestanden opnemen en de functie `load` gebruiken om Emacs ze te laten evalueren en daarmee elk van de functies in de bestanden installeren. Zie Sectie 16.9 "Bestanden laden", pagina 209.
- Ten derde, wanneer je code hebt die je systeem-breed wilt gebruiken, is het gebruikelijk die in een bestand op te nemen met de naam `site-init.el`, die wordt geladen wanneer Emacs wordt gebouwd. Dit maakt de code beschikbaar voor iedereen die jouw machine gebruikt. (Zie het bestand `INSTALL` dat onderdeel is van je Emacs distributie.)

Tenslotte, wanneer je code hebt die iedereen die Emacs gebruikt zou willen hebben, dan kun je het op een computernetwerk posten of een exemplaar naar de Free Software Foundation sturen. (Wanneer je dit doet, breng dan alsjeblieft de code en de bijbehorende documentatie onder een licentie die andere mensen toestaat het te draaien, kopiëren, bestuderen, aan te passen en verder te distribueren en die je beschermt tegen dat je werk van je afgenomen wordt.) Wanneer je een exemplaar van jouw code naar de Free Software Foundation stuurt, en jezelf en anderen juist beschermt, dan zou het in een volgende release van Emacs kunnen worden opgenomen. Dit is grotendeels hoe Emacs over de jaren gegroeid is, door donaties.

Zie Sectie "Contributing" in *The GNU Emacs Manual*, over hoe bij te dragen met opname van je code in Emacs.

3.6 `let`

De `let` expressie is een speciale vorm in Lisp die je in de meeste functiedefinities nodig hebt.

`let` wordt gebruikt om een symbool aan een variabele te koppelen of te binden op zo'n manier dat de Lisp interpreter het niet verwart met een variabele met dezelfde naam die geen onderdeel van de functie is.

Om te begrijpen waarom de speciale vorm `let` nodig is, beschouw de situatie waarin je in het algemeen aan jouw huis refereert als "het huis", zoals in de zin "Het huis moet geverfd worden". Wanneer je een vriend bezoekt en de gastheer refereert aan "het huis", refereert hij waarschijnlijk aan *zijn* huis, niet het jouwe, dus aan een ander huis.

Wanneer je vriend naar zijn huis refereert en jij denkt dat hij naar jouw huis refereert, dan kan dat verwarrend zijn. Hetzelfde kan in Lisp gebeuren wanneer een variabele die binnen de ene functie gebruikt wordt dezelfde naam heeft als een variabele die binnen een andere functie gebruikt wordt, en de twee zijn niet bedoeld om naar dezelfde waarde te refereren. De speciale vorm `let` voorkomt dit soort verwarring.

De speciale vorm `let` voorkomt verwarring. `let` creëert een naam voor een *lokale variabele* die elk gebruik van dezelfde naam buiten de `let`-expressie overschaduwt (in computerwetenschappelijk jargon noemen we dit het *binden* van de variabele).

Dit is vergelijkbaar met begrijpen dat in het huis van je gastheer, wanneer hij refereert aan "het huis", hij zijn huis bedoelt en niet het jouwe. (De symbolen die gebruikt worden om functieargumenten te benoemen worden op precies dezelfde manier als lokale variabelen gebonden. Zie Sectie 3.1 "De `defun` macro", pagina 26.)

Een andere manier om over `let` te denken is dat het een speciaal gebied in je code definieert: binnen de body van de `let`-expressie, hebben de variabelen die je benoemd hebt hun eigen lokale betekenis. Buiten de `let` body hebben zij een andere betekenis (of ze zijn helemaal niet gedefinieerd). Dit betekent dat binnen de `let` body het aanroepen van `setq` voor een variabele benoemd door de `let`-expressie de waarde instelt voor de *lokale* variabele met die naam. Echter buiten de `let` body (zoals bij het aanroepen van een functie die ergens anders is gedefinieerd), `setq` aanroepen voor een variabele benoemd door de `let`-expressie heeft *geen* effect voor die lokale variabele.[1]

`let` kan meer dan een variabele tegelijk creëren. Dus `let` geeft iedere variabele die het creëert een initiële waarde, of een door jou gespecificeerde waarde, of `nil`. (In het jargon is dit het binden van de variabele aan de waarde.) Nadat `let` de variabelen heeft gecreëerd en gebonden voert het de code in de body van de `let` uit, als de waarde van de gehele `let`-expressie. ("Uitvoeren" is een jargon term die het evalueren van de lijst betekent, het komt van het gebruik van het woord met de betekenis "een praktisch effect hebben op" (*Oxford English Dictionary*) Wanneer je een expressie evalueert om een actie uit te voeren, is "uitvoeren" geëvolueerd tot synoniem voor "evalueren".)

3.6.1 De delen van een `let` expressie

De lijst van een `let`-expressie bestaat uit drie delen. Het eerste deel is het symbool `let`. Het tweede deel is een lijst, genaamd *varlist*, van wie elk element hetzij een symbool op zichzelf is, of een lijst met twee elementen, waarvan het eerste element een symbool is. Het derde deel van de `let`-expressie is de body van de `let`. De body bestaat gebruikelijk uit een of meer lijsten.

Het sjabloon voor een `let`-expressie ziet er zo uit:

```
(let varlist body...)
```

De symbolen in de varlist zijn de variabelen die een initiële waarde krijgen door de speciale vorm `let`. De symbolen zelf krijgen een initiële waarde van `nil` en elk symbool dat het eerste is van een lijst met twee elementen is gebonden aan de waarde dat de Lisp interpreter teruggeeft bij het evalueren van het tweede element.

Een varlist kan er dus als volgt uitzien: `(draad (naalden 3))`. In dit geval, in een `let`-expressie, bindt Emacs het symbool `draad` aan een initiële waarde van `nil`, en bindt het symbool `naalden` aan een initiële waarde van 3.

Wanneer je een `let`-expressie schrijft, dan plaats je de passende expressies in de slots van het sjabloon voor `let`-expressies.

Wanneer de varlist is samengesteld uit lijsten met twee elementen, wat vaak het geval is, dan ziet het sjabloon voor de `let`-expressie er zo uit:

---

[1] Dit beschrijft het gedrag van `let` bij het gebruik van een stijl met de naam "lexical binding" (zie Sectie 3.6.4 "Hoe let variabelen bindt", pagina 36).

```
(let ((variable value)
      (variable value)
      ...)
  body...)
```

3.6.2 Voorbeeld `let` expressie

De volgende expressie creëert en geeft initiele waarden aan de twee variabelen `zebra` en `tijger`. De body van de `let`-expressie is een lijst die de `message`-functie aanroept.

```
(let ((zebra "strepen")
      (tijger "woest"))
  (message "Eén soort dieren heeft %s en de ander is %s."
           zebra tijger))
```

Hier is de varlist `((zebra "strepen") (tijger "woest"))`

De twee variabelen zijn `zebra` en `tijger`. Elke variabele is het eerste element van een lijst met twee elementen en elke waarde is het tweede element van die lijst. In de varlist bindt Emacs de variabele `zebra` aan de waarde `"strepen"`[2], en bind de variabele `tijger` aan de waarde `"woest"`. In dit voorbeeld zijn beide waarden strings. De waarde had net zo goed een andere lijst of een symbool kunnen zijn. De body van de `let` is een lijst die de `message`-functie gebruikt om een string in het echogebied te tonen.

Je kunt het voorbeeld op de gebruikelijke manier evalueren door de cursor achter het laatste haakje te zetten en *`C-x C-e`* te typen. Wanneer je dit doet, verschijnt het volgende in het echogebied:

```
"Eén soort dieren heeft strepen en de ander is woest."
```

Zoals we eerder gezien hebben toont de `message`-functie zijn argument, met uitzondering van '`%s`'. In dit voorbeeld wordt de waarde van de variabele `zebra` getoond op de plek van de eerste '`%s`' en wordt de waarde van de variabele `tijger` op de plek van de tweede '`%s`' getoond.

3.6.3 Ongeïnitialiseerde variabelen in een `let` statement

Wanneer je variabelen niet aan een specifieke initiële waarde bindt in een `let` statement, dan worden die automatisch gebonden aan de initiële waarde `nil`, zoals in de volgende expressie:

```
(let ((berk 3)
      spar
      den
      (eik 'enige))
  (message
   "Hier zijn %d variabelen met %s, %s, en %s waarde."
   berk spar den eik))
```

Hier is de varlist `((berk 3) spar den (eik 'enige))`

[2] volgens Jared Diamond in *Guns, Germs, and Steel*, "... worden zebras onvoorstelbaar gevaarlijk als ze ouder worden" maar de claim hier is dat ze niet zo woest worden als een tijger. (1997, W. W. Norton and Co., ISBN 0-393-03894-2, page 171)

Wanneer je deze expressie op de gebruikelijke manier evalueert, verschijnt het volgende in je echogebied:

```
"Hier zijn 3 variabelen met nil, nil, en enige waarde."
```

In dit voorbeeld bindt Emacs het symbool `berk` aan het getal 3, bindt het symbool `spar` en `den` aan `nil`, en bindt het symbool `eik` aan de waarde `enige`.

Merk op dat in het eerste deel van de `let`-expressie de variabelen `spar` en `den` op zichzelf staan als atomen en niet omgeven zijn met haakjes. Dit is omdat ze aan `nil` gebonden worden, de lege lijst. Maar `eik` is gebonden aan `enige` en daarom is een deel van de lijst `(eik 'enige)`. `berk` is op dezelfde manier gebonden aan het getal 3 en staat daarom in een lijst met dat getal. (Omdat een getal naar zichzelf evalueert hoeft een getal niet gequote te worden. Ook wordt het getal in de boodschap getoond met '`%d`' in plaats van '`%s`'.) De vier variabelen zijn als groep in de lijst geplaatst om ze te scheiden van de body van de `let`.

3.6.4 Hoe `let` variabelen bindt

Emacs Lisp ondersteunt twee manieren om een variabele namen aan hun waarde te binden. Deze manieren raken de delen van je programma waar een bepaalde binding geldig is. Wegens historische redenen gebruikt Emacs Lisp standaard een vorm van variabele binding met de naam *dynamic binding*. Echter in deze handleiding behandelen we de voorkeursvorm voor binding, met de naam *lexical binding*, tenzij anders vermeld (de Emacs-beheerders zijn van plan in de toekomst de standaard te wijzigen naar lexical binding). Wanneer je eerder in andere talen geprogrammeerd hebt, ben je waarschijnlijk vertrouwd met hoe lexical binding zich gedraagt.

Om lexical binding in een programma te gebruiken zet je dit in de eerste regel van je Emacs Lisp bestand:

```
;;; -*- lexical-binding: t -*-
```

Voor meer informatie hierover zie Sectie "Variable Scoping" in *The Emacs Lisp Reference Manual*.

Verschillen tussen lexical en dynamische binding

Zoals eerder besproken (zie Sectie 3.5 "Permanent installeren", pagina 32), wanneer je lokale variabelen met `let` onder lexical binding creëert, zijn die variabelen alleen geldig in de body van de `let`-expressie. In andere delen van je code hebben ze geen betekenis, dus wanneer je een elders gedefinieerde functie aanroept vanuit de `let` body, kan die functie de lokale variabelen die je gemaakt hebt niet "zien". (Anderzijds, wanneer je een functie aanroept die was gedefinieerd in een `let` body. *zou* die functie de lokale variabelen van die `let`-expressie moeten kunnen zien en wijzigen.

Onder dynamische binding zijn de regels verschillend: wanneer je `let` gebruikt. zijn de lokale variabelen geldig gedurende de uitvoering van de `let`-expressie. Dit betekent dat wanneer je `let`-expressie een functie aanroept, die functie deze lokale variabelen kan zien, ongeacht waar de functie is gedefinieerd (inclusief wanneer dit zelfs in een heel ander bestand staat).

Een andere manier om over `let` bij het gebruik van dynamische binding te denken is dat elke variabele naam een globale "stapel" van bindings heeft, en wan-

neer je de naam van de variabele gebruikt, dat refereert aan de bovenkant van de stapel. (Je kan je dit voorstellen als een stapel papieren op je bureau waarop de waarden zijn geschreven.) Wanneer je een variabele dynamisch bindt met `let`, plaatst het de nieuw gespecificeerde binding bovenop de stapel en voert vervolgens de `let` body uit. Nadat de `let` body eindigt, neemt het die binding van de stapel, en toont diegene die er voor de `let`-expressie was (als er een was).

Voorbeeld van lexical vs. dynamische binding

In sommige gevallen gedragen lexical en dynamische bindingen zich gelijk. Echter in sommige gevallen kunnen zij de betekenis van je programma wijzigen. Kijk bijvoorbeeld naar wat gebeurt in deze code onder lexical binding:

```
;;; -*- lexical-binding: t -*-

(setq x 0)

(defun getx ()
  x)

(setq x 1)

(let ((x 2))
  (getx))
     ⇒ 1
```

Het resultaat van `(getx)` is hier `1`. Onder lexical binding ziet `getx` de waarde van onze `let`-expressie niet. Dit is omdat de body van `(getx)` buiten de body van onze `let`-expressie is. Omdat `getx` bovenaan, op het globale niveau van code gedefinieerd is (dus niet binnen de body van enige `let`-expressie), zoekt het en vindt `x` eveneens op het globale niveau. De huidige globale waarde van `x` is `1` bij de uitvoering van `getx`, en dus is dat wat `getx` teruggeeft.

Wanneer we in plaats daarvan dynamisch binding gebruiken, is het gedrag anders:

```
;;; -*- lexical-binding: t -*-

(defvar x) ;; Gebruik dynamic binding voor 'x'.

(setq x 0)

(defun getx ()
  x)

(setq x 1)

(let ((x 2))
  (getx))
     ⇒ 2
```

Het resultaat van `(getx)` is nu `2`. Dat is omdat onder dynamische binding bij het uitvoeren van `getx` de huidige binding voor `x` op de bovenkant van onze stapel die is van onze `let` binding. Deze keer ziet `getx` de globale waarde van `x` niet, omdat deze binding onder degene van onze `let`-expressie in de stapel van bindings ligt.

(De `defvar` declaratie hierboven maakt zogezegd de variabele "speciaal", wat zorgt dat het de dynamische binding regels volgt in plaats van de standaard binding regels. Zie Sectie 8.5 "Initialiseer een variabele met `defvar`", pagina 104.)

3.7 De `if` speciale vorm

Een andere speciale vorm is de conditionele `if`. Deze vorm wordt gebruikt om de computer te instrueren om beslissingen te nemen. Je kunt functiedefinities schrijven zonder `if` te gebruiken, maar het wordt vaak genoeg gebruikt en is belangrijk genoeg om het hier te behandelen. Het wordt bijvoorbeeld gebruikt in de code van de functie `beginning-of-buffer`.

Het basisidee achter `if` is, dat *wanneer* een test waar is, *dan* een expressie wordt geëvalueerd. Wanneer de test niet waar is, dan wordt de expressie niet geëvalueerd. Bijvoorbeeld zou je een beslissingen kunnen maken zoals "als het warm en zonnig is, dan ga ik naar het strand!".

Een `if`-expressie geschreven in Lisp gebruikt het woord "then" niet, de test en de actie zijn de tweede en derde elementen van de lijst wiens eerste element `if` is. Echter, het test-gedeelte van de lijst wordt vaak het `wanneer`-gedeelte en het tweede argument het `dan`-gedeelte genoemd.

Bij het schrijven van een `if`-expressie is het gebruikelijk de waar-of-onwaar-test op dezelfde regel te zetten als het symbool `if`, maar de uit te voeren actie wanneer de test waar is, het dan-deel, op de tweede en verdere regels. Dit maakt de `if`-expressie makkelijker te lezen.

```
(if waar-of-onwaar-test
    actie-uit-te-voeren-als-de-test-waar-is)
```

De waar-of-onwaar-test is een expressie die door de Lisp interpreter wordt geëvalueerd.

Hier is een voorbeeld dat je op de gebruikelijke manier kunt evalueren. De test is of het getal 5 groter is dan het getal 4. Omdat dit zo is, wordt de boodschap '`5 is groter dan 4!`' getoond.

```
(if (> 5 4)                          ; wanneer-deel
    (message "5 is groter dan 4!"))  ; dan-deel
```

(De functie `>` test of zijn eerste argument groter is dan zijn tweede argument en geeft waar terug indien dat zo is.)

In werkelijk gebruik is de test natuurlijk niet voor altijd vastgelegd zoals dat het geval is bij de expressie `(> 5 4)`. In plaats daarvan is tenminste een van de variabelen die in de test gebruikt worden gebonden aan een waarde die vooraf niet bekend is. (Als de waarde vooraf bekend zou zijn, hoeven we de test niet doen!)

De waarde kan bijvoorbeeld gebonden zijn aan het argument van een functiedefinitie. In de volgende functiedefinitie is het karakter van een dier een waarde die wordt doorgegeven aan de functie. Wanneer de waarde gebonden

aan `karakteristiek woest` is, dan wordt de boodschap '`Het is een tijger!`' getoond. Zo niet, dan wordt `nil` teruggeven.

```
(defun type-dier (karakteristiek)
  "Toon boodschap in echogebied afhankelijk van KARAKTERISTIEK.
Wanneer de KARAKTERISTIEK de string \"woest\" is,
dan waarschuw voor een tijger."
  (if (equal karakteristiek "woest")
      (message "Het is een tijger!")))
```

Wanneer je dit in GNU Emacs leest, dan kun je de functiedefinitie op de gebruikelijke manier evalueren, en daarna kan je de volgende twee expressies evalueren om de resultaten te zien:

```
(type-dier "woest")

(type-dier "gestreept")
```

Wanneer je `(type-dier "woest")` evalueert, zie je de volgende boodschap in het echogebied: `"Het is een tijger!"` en wanneer je `(type-dier "gestreept")` evalueert dan zie je `nil` in het echogebied.

3.7.1 De `type-dier`-functie in detail

Laten we kijken naar de `type-dier`-functie in detail

De functiedefinitie van `type-dier` is geschreven door de slots van twee sjablonen in te vullen, een voor de functiedefinitie in zijn geheel, en een tweede voor een `if`-expressie.

Het sjabloon voor elke functie die niet interactief is, is:

```
(defun naam-van-de-functie (argument-list)
  "documentatie..."
  body...)
```

De delen van de functie die overeenkomen met het sjabloon zien er zo uit:

```
(defun type-dier  (karakteristiek)
  "Toon boodschap in echogebied afhankelijk van KARAKTERISTIEK.
Wanneer de KARAKTERISTIEK de string \"woest\" is,
dan waarschuw voor een tijger."
  body: de if expressie)
```

De naam van de functie is `type-dier`, het krijgt de waarde van één argument doorgegeven. De argumentlijst wordt gevolgd door een meerregelige documentatiestring. De documentatiestring is in dit voorbeeld opgenomen omdat het een goede gewoonte is voor elke functiedefinitie een documentatiestring te schrijven. De body van de functiedefinitie bestaat uit de `if`-expressie.

Het sjabloon voor een `if`-expressie ziet er zo uit:

```
(if waar-of-onwaar-test
    uit-te-voeren-actie-wanneer-de-waar-of-onwaar-test-waar-teruggeeft)
```

In de `type-dier`-functie, ziet de code voor de `if` er zo uit:

```
(if (equal karakteristiek "woest")
    (message "Het is een tijger!"))
```

Hier is de waar-of-onwaar-test de expressie:

```
(equal karakteristiek "woest")
```

In Lisp is `equal` een functie die vaststelt of zijn eerste argument gelijk is aan zijn tweede argument. Het tweede argument is de string `"woest"` en het eerste argument is de waarde van het symbool `karakteristiek`—met andere woorden, het argument doorgegeven aan deze functie.

In de eerste oefening van `type-dier` is het argument `"woest"` doorgegeven naar `type-dier`. Omdat `"woest"` gelijk is aan `"woest"` geeft de expressie `(equal karakteristiek "woest")` de waarde waar terug. Wanneer dit gebeurt, evalueert de `if` het tweede argument oftewel het dan-deel van de `if`: `(message "Het is een tijger!")`.

Daarnaast is in de tweede oefening van `type-dier` het argument `"gestreept"` doorgegeven naar `type-dier`. `"gestreept"` is niet gelijk aan `"woest"` en daarom is het dan-deel niet geëvalueerd en is `nil` teruggeven door de `if`-expressie.

3.8 If–then–else expressies

Een `if`-expressie kan optioneel een derde argument hebben, het *anders-deel*. voor het geval de waar-of-onwaar-test onwaar teruggeeft. Wanneer dit gebeurt wordt het tweede argument, het dan-deel van de `if`-expressie *niet* geëvalueerd, maar wordt het derde deel, het anders-deel, *wel* geëvalueerd. Je kunt dit zien als het bewolkte dag alternatief voor de beslissing “als het warm en zonnig is, ga dan naar het strand, anders lees een boek!”.

In de Lisp-code wordt het woord “else” niet geschreven, het anders-deel van een `if`-expressie komt na het dan-gedeelte. In Lisp staat het anders-deel gebruikelijk op zichzelf op een nieuwe regel en minder ingesprongen dan het dan-deel:

```
(if waar-of-onwaar-test
    uit-te-voeren-actie-wanneer-de-waar-of-onwaar-test-waar-teruggeeft
  uit-te-voeren-actie-wanneer-de-waar-of-onwaar-test-onwaar-teruggeeft)
```

De volgende `if`-expressie bijvoorbeeld toont de boodschap ‘`4 is niet groter dan 5!`’ wanneer je het op de gebruikelijke manier evalueert:

```
(if (> 4 5)                                ; if-deel
    (message "4 valselijk groter dan 5!") ; dan-deel
  (message "4 is niet groter dan 5!"))     ; anders-deel
```

Merk op dat de verschillende niveaus van inspringen het makkelijk maken het dan-deel te onderscheiden van het anders-deel. (GNU Emacs heeft verschillende commando’s die automatisch `if`-expressies correct laat inspringen. Zie Sectie 1.1.3 “GNU Emacs helpt je om lijsten te typen”, pagina 3.)

We kunnen de `type-dier`-functie uitbreiden met een anders-deel door eenvoudig een aanvullend deel in de `if`-expressie in te voegen.

Je kunt de consequenties hiervan zien wanneer je de volgende versie van de `type-dier` functiedefinitie installeert en daarna de twee volgende expressies evalueert om de verschillende argumenten aan de functie door te geven.

```
(defun type-dier (karakteristiek)
  "Toon boodschap in echogebied afhankelijk van KARAKTERISTIEK.
Wanneer de KARAKTERISTIEK de string \"woest\" is,
dan waarschuw voor een tijger."
  (if (equal karakteristiek "woest")
      (message "Het is een tijger!")
    (message "Het is niet woest!")))

(type-dier "woest")

(type-dier "gestreept")
```

Wanneer je `(type-dier "woest")` evalueert, zie je de volgende boodschap in het echogebied: `"Het is een tijger!"`, maar wanneer je `(type-dier "gestreept")` evalueert zie je `"Het is niet woest!"`.

(Wanneer de *karakteristiek* `"zeer woest"` is, dan wordt de boodschap `"Het is niet woest!"` getoond, en dat is misleidend!. Wanneer je code schrijft, moet je rekening houden met de mogelijkheid dat een dergelijk argument wordt getest door de code en je programma overeenkomstig schrijven.)

3.9 Waar en onwaarheid in Emacs Lisp

Er is een belangrijk aspect aan de waarheidstest in een `if`-expressie. Tot nu toe hebben we gesproken over “waar” en “onwaar” als waarden van predicaten alsof het nieuwe soorten van Emacs Lisp objecten zijn. Feitelijk is “onwaar” onze oude vriend `nil`. Alles wat anders is—wat dan ook— is “waar”.

De expressie die de test op waar wordt geïnterpreteerd als *waar* wanneer het resultaat van de evaluatie een waarde ongelijk aan `nil` is. Met andere woorden, het resultaat van de test wordt geacht waar te zijn wanneer de teruggegeven waarde een getal is zoals 47, een string zoals `"hallo"` of een symbool (anders dan `nil`), zoals `bloemen`, of een lijst (zolang die niet leeg is) of zelfs een buffer!

Voordat we de test op waar illustreren, hebben we eerst een uitleg van `nil` nodig.

In Emacs Lisp heeft het symbool `nil` twee betekenissen. De eerste betekenis is de lege lijst. De tweede betekenis is onwaar en is de waarde wordt teruggegeven wanneer de waar-of-onwaar-test onwaar test. `nil` kan worden geschreven als de lege lijst, `()`, of als `nil`. Voor wat de Lisp interpreter betreft zijn `()` en `nil` gelijk. Mensen echter gebruiken meestal `nil` voor onwaar en `()` voor de lege lijst,

In Emacs Lisp wordt elke waarde die niet `nil` is—niet de lege lijst is— beschouwd als waar. Dit betekent dat wanneer een evaluatie iets teruggeeft dat geen lege lijst is, een `if`-expressie het als waar test. Wanneer bijvoorbeeld een getal in het slot voor de test gestopt wordt. wordt het geëvalueerd en zal het zichzelf teruggeven, omdat dat is wat getallen doen wanneer ze worden geëvalueerd. De expressie test als onwaar uitsluitend wanneer `nil`, een lege lijst, wordt teruggegeven door de expressie te evalueren.

Je kunt dit zien door het evalueren van de twee expressies in de volgende voorbeelden.

In het eerste voorbeeld wordt het getal 4 geëvalueerd als de test in de `if`-expressie en geeft zichzelf terug. Als gevolg wordt het dan-deel van de expressie geëvalueerd en teruggegeven: '`waar`' verschijnt in het echogebied. In het tweede voorbeeld, de `nil` geeft onwaar aan, als gevold wordt het anders-deel van de expressie geëvalueerd en teruggegeven: '`onwaar`' verschijnt in het echogebied.

```
(if 4
    'waar
  'onwaar)

(if nil
    'waar
  'onwaar)
```

Overigens wanneer een zinnige waarde niet beschikbaar is voor een test die waar teruggeeft, dan geeft de Lisp interpreter het symbool `t` voor waar. Bijvoorbeeld evaluatie van de expressie `(> 5 4)` geeft `t` terug, zoals je op de gebruikelijke manier kunt zien.

```
(> 5 4)
```

Anderzijds, deze functie geeft `nil` terug, wanneer het als onwaar test.

```
(> 4 5)
```

3.10 `save-excursion`

De functie `save-excursion` is de laatste speciale vorm die we in dit hoofdstuk bespreken.

In Emacs Lisp programma's voor editing is de functie `save-excursion` heel gebruikelijk. Het bewaart de lokatie van point, voert de body van de functie uit en herstelt point naar zijn vorige positie indien zijn lokatie is gewijzigd. Het primaire doel is te voorkomen dat de gebruiker verrast en verstoord wordt door onverwachte verplaatsing van point.

Voordat we `save-excursion` bespreken is het echter zinvol eerst te bekijken wat in GNU Emacs point en mark zijn. *Point* is de huidige lokatie van de cursor. Waar de cursor ook is, dat is point. Preciezer gezegd, in terminals waar de cursor bovenop een karakter verschijnt, is point direct voor dat karakter. In Emacs Lisp is point een integer (geheel getal). Het eerste karakter in een buffer is nummer één, het tweede is nummer twee, enzovoorts. De functie `point` geeft de huidige positie van de cursor als getal terug. Elk buffer heeft zijn eigen waarde voor point.

De *mark* is een andere positie in het buffer. De waarde kan ingesteld worden met een commando zoals `C-SPC` (`set-mark-command`). Wanneer een mark is ingesteld, laat je met het commando `C-x C-x` (`exchange-point-and-mark`) de cursor naar mark springen en de mark instellen op de vorige positie van point. Verder, als je een andere mark instelt wordt de positie van de vorige mark bewaard in de mark-ring. Veel mark-posities kunnen zo bewaard worden. Je laat de cursor naar een bewaarde mark springen door een of meer keer `C-u C-SPC` te typen.

De deel van het buffer tussen point en mark heet de *region*. Talrijke commando's werken op de region, inclusief `center-region`, `count-words-region`, `kill-region` en `print-region`.

De speciale vorm `save-excursion` bewaart de lokatie van point en herstelt de positie nadat de code in de body van de speciale vorm is geëvalueerd door de Lisp interpreter. Als point dus aan het begin van een stuk tekst staat en bepaalde code heeft point naar het eind van het buffer verplaatst, plaatst `save-excursion` point terug naar de lokatie waar het eerder was, nadat de expressies in de body van de functie zijn geëvalueerd.

In Emacs verplaatsen functies regelmatig point als onderdeel van de interne werking daarvan, alhoewel de gebruiker dat niet zou verwachten. Om te voorkomen dat de gebruiker lastig gevallen wordt door springen die zowel onverwacht als (vanuit het gezichtspunt van de gebruiker) onnodig zijn, wordt `save-excursion` vaak gebruikt om point op de plek te houden waar de gebruiker die verwacht. Het gebruik van `save-excursion` vormt een goede huishouding.

Om zeker te stellen dat het huis schoon blijft, herstelt `save-excursion` de waarde van point zelfs wanneer iets fout gaat in de code (of, meer preciezer en in het juiste jargon, "in geval van een abnormale exit"). Deze eigenschap is erg behulpzaam.

Naast het registreren van de waarde van point houdt `save-excursion` bij wat de huidige buffer is en herstelt die ook. Dit betekent dat je code kunt schrijven die van buffer wisselt en je `save-excursion` laat terugschakelen naar de originele buffer. Dit is hoe `save-excursion` gebruikt wordt in `append-to-buffer`. (Zie Sectie 4.4 "De definitie van `append-to-buffer`", pagina 52.)

3.10.1 Sjabloon voor een `save-excursion` expressie

Het sjabloon voor code die `save-excursion` gebruikt is simpel:

```
(save-excursion
  body...)
```

De body van de functie bestaat uit een of meer expressies die de Lisp interpreter opeenvolgend evalueert. Wanneer de body meer dan een expressie bevat, wordt de waarde van de laatste teruggegeven als waarde van de `save-excursion`-functie. De andere expressies in de body worden uitsluitend geëvalueerd voor hun zij-effecten en `save-excursion` zelf wordt alleen gebruikt voor zijn zij-effect (het herstellen van de positie van point).

Meer gedetailleerd ziet het sjabloon voor een `save-excursion`-expressie er zo uit:

```
(save-excursion
  eerste-expressie-in-body
  tweede-expressie-in-body
  derde-expressie-in-body
   ...
  laatste-expressie-in-body)
```

Een expressie kan uiteraard zelf een symbool zijn, of een lijst.

In Emacs Lisp code komt een `save-excursion` vaak voor binnen de body van een `let`-expressie. Dit ziet er zo uit:

```
(let varlist
  (save-excursion
    body...))
```

3.11 Terugblik

In de laatste hoofdstukken introduceerden we een macro en een flink aantal functies en speciale vormen. Deze bespreken we hier in het kort, samen met enkele vergelijkbare functies die we nog niet behandeld hebben.

`eval-last-sexp`
: Evalueer de laatste symbolische expressie voor de huidige lokatie van point. De waarde wordt getoond in het echogebied tenzij de functie met een argument is aangeroepen. In dat geval wordt de output in het huidige buffer getoond. Dit commando is gewoonlijk gebonden aan *C-x C-e*.

`defun`
: Definieer functie. Deze macro heeft maximaal vijf onderdelen: de naam, een sjabloon voor de argumenten die aan de functie doorgegeven worden, documentatie, een optionele interactieve declaratie en de body van de definitie.

  In Emacs is de functiedefinitie van bijvoorbeeld `dired-unmark-all-marks` als volgt:

  ```
  (defun dired-unmark-all-marks ()
    "Remove all marks from all files in the Dired buffer."
    (interactive)
    (dired-unmark-all-files ?\r))
  ```

`interactive`
: Verklaar de interpreter dat de functie interactief gebruikt kan worden. Deze speciale vorm kan worden gevolgd door een string met een of meer delen die op volgorde informatie aan de argumenten van de functie doorgeven. Deze delen kunnen de interactief ook om informatie vragen. Delen van de string zijn gescheiden door nieuwe regels, '`\n`'.

  Gebruikelijke code karakters zijn:

  `b` — De naam van een bestaand buffer

  `f` — De naam van een bestaand bestand

  `p` — Het numerieke prefix-argument. (Merk op dat dit de kleine letter `p` is)

  `r` — Point en mark als twee numerieke argumenten, de kleinste eerst. Dit is de enige letter die twee opvolgende argumenten specificeert, in plaats van een.

  Zie Sectie "Code Characters for '`interactive`'" in *The GNU Emacs Lisp Reference Manual*, voor een complete lijst van code karakters.

`let` Verklaar dat een lijst van variabelen wordt gebruikt binnen de body van de `let` en geef ze een initiële waarde, hetzij `nil` of een gespecificeerde waarde. Evalueer daarna de rest van de expressies in de body van de `let` en geef de waarde van de laatste terug. Binnen de body van de `let` ziet de Lisp interpreter de waarden niet van variabelen met dezelfde naam die gebonden zijn buiten de `let`.

Bijvoorbeeld,

```
(let ((foo (buffer-name))
      (bar (buffer-size)))
  (message
   "Dit buffer is %s en heeft %d karakters."
   foo bar))
```

`save-excursion`
Registreer de waarde van point en het huidige buffer voor het evalueren van de body van deze speciale vorm. Herstel achteraf de waarde van point en buffer.

Bijvoorbeeld,

```
(message "We zijn %d karakters in dit buffer."
         (- (point)
            (save-excursion
              (goto-char (point-min)) (point))))
```

`if` Evalueer het eerste argument van de functie. Wanneer het waar is, evalueer het tweede argument, anders evalueer het derde argument, mits die er is.

De speciale vorm `if` heet een *conditional*. Er zijn andere conditionals in Emacs Lisp, maar `if` is waarschijnlijk de meest gebruikte.

Bijvoorbeeld,

```
(if (= 22 emacs-major-version)
    (message "Dit is versie 22 Emacs")
  (message "Dit is niet versie 22 Emacs"))
```

`<`
`>`
`<=`
`>=` De functie `<` test of het eerste argument kleiner is dan het tweede argument. Een corresponderende functie, `>` test of het eerste argument groter is dan het tweede. Vergelijkbaar test `<=` of het eerste argument kleiner of gelijk is aan het tweede en `>=` test of het eerste argument groter of gelijk is aan het tweede. In alle gevallen moeten beide argumenten getallen zijn of markers (markers geven de positie in buffers aan).

`=` De functie `=` test of twee argumenten, beide nummers of markers, gelijk zijn.

`equal`
`eq` Test of twee objecten hetzelfde zijn. `equal` gebruikt een betekenis van "hetzelfde" en `eq` gebruikt een andere. `equal` geeft waar terug wanneer de twee objecten dezelfde structuur en inhoud hebben, zoals twee exemplaren van hetzelfde boek. Daarnaast geeft `eq` waar terug indien beide argumenten daadwerkelijk hetzelfde object zijn.

`string<`
`string-lessp`
`string=`
`string-equal`
De functie `string-lessp` test of het eerste argument kleiner is dan het tweede. Een kortere alternatieve naam voor dezelfde functie (een `defalias`) is `string<`.

De argumenten van `string-lessp` moeten strings of symbolen zijn, de volgorde is lexicografish, dus hoofd- of kleine letters is van belang. Het toont de namen van symbolen die worden gebruikt in plaats van de symbolen zelf.

Een lege string, '`""`', is een string zonder karakters er in, en is kleiner dan elke andere string van karakters.

`string-equal` verschaft de corresponderende test voor gelijkheid. De kortere alternatieve naam is `string=`. Recente versies van Emacs bevatten `string-greaterp`, met `string>` als korter alternatief. Er zijn geen string test- functies die corresponderen met `>=` of `<=`.

`message` Toon een boodschap in het echogebied. Het eerste argument is een string die '`%s`', '`%d`' of '`%c`' kan bevatten om de waarde van argumenten te tonen die volgen op de string. Het voor '`%s`' gebruikte argument moet een string of een symbool zijn, het voor '`%d`' gebruikte argument moet een getal zijn. Het voor '`%c`' gebruikte argument moet een ASCII codenummer zijn, het wordt getoond als het karakter met die ASCII-code. (Verschillende andere %-sequences zijn niet genoemd.)

`setq`
`set` De speciale vorm `setq` stelt de waarde van zijn eerste argument in op de waarde van het tweede argument. Het eerste argument wordt automatisch gequote door `setq`. Het doet hetzelfde voor opvolgende argumentparen.

`buffer-name`
Zonder een argument, geeft de waarde van het buffer terug als een string.

`buffer-file-name`
Zonder een argument, geeft de naam van het bestand terug dat het buffer bezoekt.

`current-buffer`
: Geeft het buffer waar Emacs actief is terug. Het is misschien niet het buffer dat zichtbaar op het scherm is.

`other-buffer`
: Geeft het meest recent geselecteerde buffer (anders dan het buffer doorgegeven aan `other-buffer` als een argument en anders dan het huidige buffer).

`switch-to-buffer`
: Selecteert een buffer voor Emacs om actief in te zijn en toon het in het huidige venster zodat de gebruiker het kan zien. Gebruikelijk gebonden aan *`C-x b`*.

`set-buffer`
: Schakelt de aandacht van Emacs naar een buffer waarop programma's gaan draaien. Verandert niet wat het venster toont.

`buffer-size`
: Geeft het aantal karakters in het huidige buffer.

`point`
: Geeft de waarde van de huidige positie van de cursor, als een integer, telt het aantal karakters vanaf het begin van het buffer.

`point-min`
: Geeft de minimaal toegestane waarde van point in het huidige buffer. Dit is 1, behalve wanneer narrowing van toepassing is.

`point-max`
: Geef de maximaal toegestane waarde van point in het huidige buffer. Dit is het einde van het buffer, tenzij narrowing van toepassing is.

3.12 Oefeningen

- Schrijf een niet-interactieve functie die de waarde van zijn argument, een getal, verdubbelt. Maak die functie interactief.
- Schrijf een functie die test of de huidige waarde van `fill-column` groter is dan het argument dat wordt doorgegeven aan de functie, en zo ja, toon het als een passende boodschap.

4 Enkele buffer gerelateerde functies

In dit hoofdstuk bestuderen we in detail verschillende in GNU Emacs gebruikte functies. We noemen dit een "rondleiding". Deze functies worden gebruikt als voorbeelden van Lisp code maar zijn geen denkbeeldige voorbeelden. Met de uitzondering van de eerste versimpelde functiedefinitie tonen deze functies de werkelijke code van GNU Emacs. Je kunt veel van deze definities leren. De hier beschreven functies zijn alle gerelateerd aan bufers. Later kijken we naar andere functies.

4.1 Meer informatie vinden

In deze rondleiding beschrijf ik elke nieuwe functie die we behandelen. Soms in detail en soms kort. Wanneer je geïnteresseerd bent vraag je op elk moment de volledige documentatie op van elke Emacs Lisp functie door `C-h f` te typen, en daarna de naam van de functie (gevolg met `RET`). Je kunt op ook de volledige documentatie van een variabele opvragen met het typen van `C-h v`, en daarna de naam van de variabele (gevolg met `RET`).

`describe-function` vertelt je ook de lokatie van de functie.

Plaats point in de naam van het bestand dat de functie bevat en druk op de `RET`-toets. In dit geval betekent `RET` `push-button` in plaats van "return" of "enter". Emacs neemt je meteen naar de functiedefinitie.

Meer in het algemeen, wanneer je de functie in het originele bronbestand wilt zien, spring je met de functie `xref-find-definitions` er heen. `xref-find-definitions` werkt met een brede variëteit van talen, niet alleen Lisp en C, en het werkt ook met non-programming tekst. `xref-find-definitions` springt bijvoorbeeld naar de verschillende nodes in de Texinfo broncode van dit document (onder voorwaarde dat je het `etags`-hulpprogramma alle nodes van alle handleidingen de met Emacs komen hebt laten registeren zie Sectie "Create Tags Table" in *The GNU Emacs Manual*).

Type `M-.` (dus druk op de punt-toet terwijl je de `META`-toets ingedrukt houdt, of typ anders de `ESC` gevold door de punt-toets) om het commando `xref-find-definitions` te gebruiken, en vervolgens, na de prompt, de naam van de functie wiens broncode je wilt zien, zoals `mark-whole-buffer` en typ daarna `RET`. (Wanneer het commando geen prompt geeft, roep het dan met een argument: `C-u M-.`). zie Sectie 3.4 "Interactieve opties", pagina 31.) Emacs schakelt buffers en toont de broncode van de functie op je scherm[1] Om naar je buffer terug te keren type je type `M-,` of `C-x b` `RET`. (Op sommige toetsenborden, heeft de `META`-toets het label `ALT`.)

Overigens worden de bestanden die Lisp code bevatten gewoonlijk *bibliotheken* genoemd. De metafoor is afgeleid van een gespecialiseerde bibliotheek, zoals een juridische bibliotheek of een technische bibliotheek, in plaats van een algemene bibliotheek. Elke bibliotheek, of bestand, bevat functies die aan bepaald onderwerp of activiteit gerelatieerd zijn, zoals `abbrev.el` voor het afhandelen van afkortingen

[1] Wanneer Emacs, in plaats van het tonen van de broncode voor een Lisp functie, je vraagt welke tags tabel te openen, roep dan `M-.` aan vanuit een buffer wiens major mode Emacs Lisp of Lisp Interaction is.

en andere snelkoppelingen voor het typen, en `help.el` for hulp. (Soms verschaffen verschillende bestanden code voor een enkele activiteit, zoals de diverse `rmail...` bestanden code voor het lezen van e-mail verschaffen.) In *The GNU Emacs Manual* zie je zinnen zoals "het *`C-h p`* commando laat je de standaard Emacs lisp bibliotheken doorzoeken op onderwerp sleutelwoorden."

4.2 Een versimpelde `beginning-of-buffer` definitie

De `beginning-of-buffer` is een goede functie om mee te starten omdat je er waarschijnlijk al bekend mee bent en makkelijk te begrijpen is. Gebruikt als een interactief commando verplaatst `beginning-of-buffer` de cursor naar het begin van het buffer, en zet de mark op de vorige positie. Het is gewoonlijk gebonden aan *`M-<`*.

In deze sectie bespreken we een verkorte versie van de functie die laat zien hoe het meestal gebruikt wordt. De verkorte functie werkt, maar heeft geen code voor een complexe optie. In een andere sectie zullen we de volledige code bespreken. (Zie Sectie 5.3 "Complete definitie van `beginning-of-buffer`", pagina 66.)

Voordat we naar de code kijken staan we stil bij wat de functiedefinitie moet bevatten: het moet een expressie bevatten die de functie interactief maakt zodat het kan worden aangeroepen met typen van *`M-x beginning-of-buffer`* of door het typen van een toetscombinatie zoals *`M-<`*. Het moet code bevatten om de mark op de originele positie te plaatsen en het moet code bevatten om de cursor naar het begin van het buffer te verplaatsen.

Hier is de complete text van de verkorte versie van de functie:

```
(defun versimpelde-beginning-of-buffer ()
  "Verplaats point naar begin van het buffer;
plaats mark op de vorige positie."
  (interactive)
  (push-mark)
  (goto-char (point-min)))
```

Zoals alle functiedefinities heeft deze definitie de vijf delen volgens de macro `defun`:

1. De naam: in dit voorbeeld `versimpelde-beginning-of-buffer`.
2. Een lijst met argumenten, in dit voorbeeld een lege lijst, `()`.
3. De documentatiestring
4. De interactive expressie
5. De body

In deze functiedefinitie is de argumentlijst leeg, dit betekent dat de functie geen argumenten vereist. (Wanneer we naar de definitie van de complete functie kijken zien we dat deze een optioneel argument kan krijgen.)

De interactive expressie vertelt Emacs dat de functie bedoeld is voor interactief gebruik. In dit voorbeeld heeft `interactive` geen argumenten omdat `versimpelde-beginning-of-buffer` die niet nodig heeft.

De body van de functie bestaat uit twee regels:

```
(push-mark)
(goto-char (point-min))
```

De eerste van de twee regels is de expressie `(push-mark)`. Wanneer de Lisp interpreter deze expressie evalueert stelt het de mark op de huidige positie van de cursor in, wat die ook is. De positie van deze mark wordt bewaard in de mark-ring.

De volgende regel is `(goto-char (point-min))`. Deze expressie springt de cursor naar het minimum punt in het buffer, dat is het begin van de buffer (of het begin van het bereikbare deel van de buffer wanneer is het versmald. Zie Hoofdstuk 6 “Versmallen en verbreden”, pagina 74.)

Het `push-mark` commando zet de mark op de plaats waar de cursor was voordat het door de `(goto-char (point-min))` expressie naar het begin van de buffer werd verplaatst. Daardoor kan je, als je dat wilt, terug gaan naar de plek waar je oorspronkelijk was door *`C-x C-x`* te typen.

Dat is alles wat er is voor deze functiedefinitie!

Wanneer je code zoals deze leest en een onbekende functie tegenkomt zoals `goto-char`, kan je met het `describe-function` commando ontdekken wat die doet. Typ *`C-h f`* gevold door de naam van de functie en toets daarna `RET` om dit commando te gebruiken. Het `describe-function` commando toont de documentatiestring van de functie in een `*Help*`-venster. Bijvoorbeeld de documentatie voor `goto-char` is:

```
Set point to POSITION, a number or marker.
Beginning of buffer is position (point-min), end is (point-max).
```

Het enige argument van de functie is de gewenste positie.

De prompt voor `describe-function` stelt je het symbool onder of voorafgaand aan de cursor voor, zodat je minder hoeft te typen door de cursor boven of meteen achter de functie te plaatsen en dan *`C-h f`* `RET` te typen.)

De functiedefinitie van `end-of-buffer` is op dezelfde manier geschreven als de `beginning-of-buffer`-definitie, belalve dat de body van de functie de expressie `(goto-char (point-max))` bevat in plaats van `(goto-char (point-min))`.

4.3 De definitie van `mark-whole-buffer`

De functie `mark-whole-buffer` is niet lastiger te begrijpen dan de `versimpelde-beginning-of-buffer`-functie. In dit geval kijken we echter naar de complete functie en niet naar een verkorte versie.

De functie `mark-whole-buffer` wordt minder vaak gebruikt dan de functie `beginning-of-buffer`, maar is toch zinvol: het markeert de hele buffer als region door de point aan het begin en de mark aan het einde van de buffer te plaatsen. Het is gebruikelijk gebonden aan *`C-x h`*.

In GNU Emacs 22 ziet de code voor de gehele functie er zo uit:

```
(defun mark-whole-buffer ()
  "Put point at beginning and mark at end of buffer.
You probably should not use this function in Lisp programs;
it is usually a mistake for a Lisp function to use any subroutine
that uses or sets the mark."
  (interactive)
  (push-mark (point))
  (push-mark (point-max) nil t)
  (goto-char (point-min)))
```

Zoals alle andere functie past de functie `mark-whole-buffer` in het sjabloon voor de functiedefinitie. Het sjabloon is als volgt:

```
(defun naam-van-de-functie (argument-list)
  "documentatie..."
  (interactive-expression...)
  body...)
```

Dit is hoe de functie werkt: de naam van de functie is `mark-whole-buffer`, gevold door een lege argumentlijst '`()`', wat inhoudt dat de functie geen argumenten vereist. De documentatie komt daarna.

De volgende regel is de `(interactive)`-expressie die Emacs vertelt dat de functie interactief gebruikt gaat worden. Deze details zijn vergelijkbaar met de in de vorige sectie beschreven functie `versimpelde-beginning-of-buffer`.

4.3.1 Body van `mark-whole-buffer`

De body van de functie `mark-whole-buffer` bestaat uit drie regels code:

```
(push-mark (point))
(push-mark (point-max) nil t)
(goto-char (point-min))
```

De eerste van deze regels is de expressie `(push-mark (point))`.

Deze regel doet precies hetzelfde als de eerste regel in de body van de functie `versimpelde-beginning-of-buffer`, waar `(push-mark)` staat. In beide gevallen zet de Lisp interpreter de mark op de huidige positie van de cursor.

Ik weet niet waarom `(push-mark (point))` in `mark-whole-buffer` staat en `(push-mark)` in de expressie `beginning-of-buffer` . Misschien wist wie de code schreef niet dat de argumenten van `push-mark` optioneel zijn en dat wanneer geen argument aan `push-mark` wordt doorgegeven, de functie automatisch standaard de lokatie van de cursor gebruikt. Of misschien was de expressie zo geschreven dat die parallel aan de structuur van de volgende regel is. In elk geval zorgt de regel dat Emacs de positie van point vaststelt en de mark daar zet.

De volgende regel van `mark-whole-buffer` was in een eerdere versie van GNU Emacs `(push-mark (point-max))`. Deze expressie zet de mark op het punt met het hoogste getal in het buffer. Deze is aan het eind van het buffer (of, wanneer het buffer versmald is, aan het eind van het bereikbare deel van de buffer, Zie Hoofdstuk 6 "Versmallen en verbreden", pagina 74.) Nadat de mark is ingesteld, is de vorige mark, die op point was gezet, niet meer ingesteld, maar Emacs herinnert zijn positie, net als het alle recente marks altijd herinnert. Dit betekent dat, als je het wilt, je naar de positie terugkeert met het twee keer typen van `C-u C-SPC`.

In GNU Emacs 22 is `(point-max)` iets meer gecompliceerd. De regel is:

```
(push-mark (point-max) nil t)
```

De expressie werkt bijna hetzelfde als hiervoor. Het stelt de mark in op de hoogst mogelijk genummerde plaats in het buffer. Echter in deze versie heeft `push-mark` twee aanvullende argumenten. Het tweede argument voor `push-mark` is `nil`. Dit vertelt de functie dat het de boodschap "Mark set" *moet* tonen wanneer het de mark pusht. Het derde argument is `t`. Dit vertelt `push-mark` om de mark te activeren wanneer Transient Mark mode aan staat. Transient Mark mode highlight de huidige actieve region. Het staat vaak uit.

De laatste regel van de functie is tenslotte `(goto-char (point-min)))`. Dit is op precies dezelfde manier geschreven als in `beginning-of-buffer`. De expressie verplaatst de cursor naar het minimale punt in het buffer, dat is het begin van het buffer (of het begin van het bereikbare deel van het buffer). Het resultaat is dat point aan het begin van het buffer en de mark aan het eind van het buffer staan. Het hele buffer is daardoor de region.

4.4 De definitie van `append-to-buffer`

Het `append-to-buffer` commando is complexer dan het `mark-whole-buffer` commando. Wat het doet is de region (dat is het deel van het buffer tussen point en mark) van het huidige buffer kopiëren naar een specifiek buffer.

Het `append-to-buffer` commando gebruikt de functie `insert-buffer-substring` om de region te kopiëren. `insert-buffer-substring` doet wat de naam zegt: het neemt een substring van een buffer en voegt die in een andere buffer in.

Het grootste deel van `append-to-buffer` houdt zich bezig met het voorbereiden van de condities voor het werk van `insert-buffer-substring`: de code specificeert het buffer waar de tekst heen gaat, het venster waar het vandaan komt en naar toe gaat en de region die gekopieerd wordt.

Hier is een mogelijke implementatie van de functie:

```
(defun append-to-buffer (buffer start end)
  "Voeg tekst van de region toe aan het gespecificeerde buffer.
Het wordt voor de point in dat buffer ingevoegd.

Geef bij het aanroepen vanuit een programma drie argumenten:
BUFFER (of buffernaam), START en END.
START en END specificeren het te kopiëren deel van het huidige buffer."
  (interactive
   (list (read-buffer "Append to buffer: " (other-buffer
                                            (current-buffer) t))
         (region-beginning) (region-end)))
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (let* ((append-to (get-buffer-create buffer))
             (windows (get-buffer-window-list append-to t t))
             point)
        (set-buffer append-to)
        (setq point (point))
        (barf-if-buffer-read-only)
        (insert-buffer-substring oldbuf start end)
        (dolist (window windows)
          (when (= (window-point window) point)
            (set-window-point window (point))))))))
```

De functie kan begrepen worden door het te bekijken als een serie ingevulde sjablonen.

Het buitenste sjabloon is voor de functiedefinitie. In deze functie ziet het er zo uit (met verschillende ingevulde slots):

```
(defun append-to-buffer (buffer start end)
  "documentatie..."
  (interactive ...)
  body...)
```

De eerste regel van de functie bevat de naam en drie argumenten. De argumenten zijn de `buffer` waar het de tekst naar toe kopieert, en de `start` en `end` van de te kopiëren region in het huidige buffer.

Het volgende deel is de documentatie die helder en compleet is. De drie argumenten zijn, volgens de conventie, in hoofdletters geschreven, zodat je ze makkelijk kunt opmerken. Nog beter is dat ze in dezelfde volgorde staan als in de argumentlijst.

Merk op de dat de documentatie de buffer en zijn naam onderscheidt. (De functie kan met beide overweg.)

4.4.1 De `append-to-buffer` interactieve expressie

Omdat de `append-to-buffer` functie interactief gebruikt gaat worden, moet de functie een `interactive` expressie hebben. (Voor een terugblik op `interactive`, zie Sectie 3.3 “Een functie interactief maken”, pagina 29.

De exressie is als volgt:

```
(list (read-buffer
       "Append to buffer: "
       (other-buffer (current-buffer) t))
      (region-beginning)
      (region-end)))
```

Deze expressie is er niet een met letters die voor de delen staan, zoals eerder beschreven. In plaats daarvan begint het met een lijst met deze delen:

Het eerste deel van de lijst is een expressie die de naam van het buffer leest en als een string teruggeeft. Dat is `read-buffer`. De functie vereist een prompt as het eerste argument, '`"Append to buffer: "`'. Het tweede argument vertelt het commando welke waarde te verschaffen wanneer je er geen specificeert.

In dit geval is het tweede argument een expressie bestaand uit de functie `other-buffer`, een utizondering en een '`t`', dat voor waar staat.

Het eerste argument voor `other-buffer`, de uitzondering, is opnieuw een functie, `current-buffer`. Die wordt niet teruggegeven. Het tweede argument is het symbool voor waar, `t`. Deze vertelt `other-buffer` dat het zichtbare buffers mag tonen (behalve dat het in dit geval niet het huidige buffer toont, wat logisch is).

De expressie ziet er zo uit:

```
(other-buffer (current-buffer) t)
```

Het tweede en derde argument in de `lijst` expressies zijn `(region-beginning)` en `(region-end)`. Deze twee functies specificeren het begin en einde van de toe te voegen tekst.

Oorspronkelijk gebruikte het commando de letters '`B`' en '`r`'. De gehele `interactive` expressie zag er zo uit:

```
(interactive "BAppend to buffer: \nr")
```

Maar toen dat gereed was, bleek de standaard waarde van het buffer waar naar toe geschakeld was onzichtbaar. Dat was niet gewenst.

(De prompt werd gescheiden van het tweede argument met een nieuwe regel, '`\n`'. Het werd gevolgd door een '`r`' die Emacs vertelde om de twee argumenten die volgen op het symbool `buffer` in de argumentlijst van de functie (te weten, `start` en `end`) aan de waarden van point en mark te binden. Dat argument werkte prima.)

4.4.2 De body van `append-to-buffer`

De body van de functie `append-to-buffer` begint met `let`.

Zoals we eerder zagen (zie Sectie 3.6 "`let`", pagina 33), is het doel van een `let` expressie om een of meer variabelen die uitsluitend worden gebruikt binnen de body van de `let` te creëeren en een initiële waarde te geven. Dit betekent dat zo'n variabele niet verward wordt met een variabele met dezelfde naam buiten de `let` expressie.

We kunnen zien hoe de `let`-expressie past in de functie als geheel, door een overzicht van het sjabloon te tonen voor de `append-to-buffer` met de `let`-expressie.

```
(defun append-to-buffer (buffer start end)
  "documentatie..."
  (interactive ...)
  (let ((variable value))
       body...))
```

De `let`-expressie heeft drie elementen:

1. Het symbool `let`;
2. Een varlist die, in dit geval, een enkele lijst met twee elementen bevat, (*`variable value`*);
3. De body van de `let`-expressie.

In de functie `append-to-buffer` ziet de varlist er zo uit:

```
(oldbuf (current-buffer))
```

In dit deel van de `let` de ene variabele, `oldbuf`, is gebonden aan de waarde die de `(current-buffer)` expressie teruggeeft. De variabele, `oldbuf` wordt gebruikt om het buffer bij te houden in welk je werkt en vanwaar je kopieert.

Het element of de elementen van een varlist zijn omgeven door een paar haakjes, zodat de Lisp interpreter de varlist kan onderscheiden van de body van de `let`. Als gevolg hiervan is de lijst met twee elementen binnen de varlist omgeven door een omtrekkende paar haakjes. De regel ziet er zo uit:

```
(let ((oldbuf (current-buffer)))
  ... )
```

De twee haakjes voor `oldbuf` kunnen je verrassen wanneer je je niet realseerde dat het eerste haakje voorafgaand aan `oldbuf` de grens van de varlist markeert en het tweede haakje het begin van de lijst met twee elementen, `(oldbuf (current-buffer))`.markeert.

4.4.3 `save-excursion` in `append-to-buffer`

De body van de `let` expressie in `append-to-buffer` bestaat uit een `save-excursion` expressie.

De functie `save-excursion` bewaart de lokatie van point en herstelt point naar die positie na afloop van de uitvoering van de expressies in de body van `save-excursion`. Verder houdt `save-excursion` bij wat het oorspronkelijke buffer is en herstelt die. Dit is hoe `save-excursion` wordt gebruikt in `append-to-buffer`.

Overigens is het waardevol hier op te merken dat een Lisp functie normaal zo geformatteerd is dat alles wat over meerdere regels verspreid is, meer inspringt dan het eerste symbool. In deze functiedefinitie springt de `let` meer in dan de `defun` en de `save-excursion` springt meer in dan de `let`, zoals dit:

```
(defun ...
  ...
  ...
  (let...
    (save-excursion
      ...
```

De formattering-conventie maakt het makkelijk te zien dat de regels in de body van `save-excursion` zijn ingesloten in de haakjes geassocieerd met `save-excursion`, net zoals de `save-excursion` ingesloten is in de haakjes geassocieerd met de `let`:

```
(let ((oldbuf (current-buffer)))
  (save-excursion
    ...
    (set-buffer ...)
    (insert-buffer-substring oldbuf start end)
    ...))
```

Het gebruik van de functie `save-excursion` kan beschouwd worden als een proces om de slots in het sjabloon in te vullen:

```
(save-excursion
  eerste-expressie-in-body
  tweede-expressie-in-body
   ...
  laatste-expressie-in-body)
```

In deze functie bevat de body van save-excursion slechts één expressie, de let* expressie. Je kent de let-functie. de functie let* is anders. Het maakt Emacs mogelijk om elke variabele in de varlist op volgorde in te stellen, de een na de ander. zo dat de variabelen in een later gedeelte van de varlist gebruik kunnen maken van waarden die Emacs eerder in de varlist instelde.

Kijkend naar de let* expressie in append-to-buffer:

```
(let* ((append-to (get-buffer-create buffer))
       (windows (get-buffer-window-list append-to t t))
       point)
  BODY...)
```

zien we dat append-to gebonden is aan de waarde teruggegeven door (get-buffer-create buffer). Op de volgende regel is append-to gebruikt als argument voor get-buffer-window-list. Dit zou niet mogelijk geweest zijn met de let-expressie. Merk op dat point automatisch is gebonden aan nil, net zo als dat zou zijn gebeurd in de let-statement.

Laten we ons nu focussen op de functies set-buffer en insert-buffer-substring in de body van de let* expressie.

In het verleden was de set-buffer-expressie eenvoudigweg:

```
(set-buffer (get-buffer-create buffer))
```

maar nu is het

```
(set-buffer append-to)
```

Dit is omdat append-to gebonden was aan (get-buffer-create buffer) eerder in de let*-expressie.

De append-to-buffer functiedefinitie voegt tekst in van het buffer waar je nu in bent naar een genoemd buffer. Toevallig doet insert-buffer-substring precies het omgekeerde—het kopieert tekst van een ander buffer naar het huidige buffer—en dat is waarom de append-to-buffer definitie begint met de let dat het lokale symbool oldbuf bindt aan de door current-buffer teruggegeven waarde.

De insert-buffer-substring expressie ziet er zo uit:

```
(insert-buffer-substring oldbuf start end)
```

De functie insert-buffer-substring kopieert een string *van* het buffer gespecificeerd als eerste argument en voegt de string in het huidige buffer. In de geval is het argument van insert-buffer-substring de waarde van de variabele die de let creëerde en bond, namelijk de waarde van oldbuf, wat het huidige buffer was toen je het append-to-buffer commando gaf.

Nadat insert-buffer-substring zijn werk gedaan heeft, herstelt save-excursion de actie naar het originele buffer en heeft append-to-buffer zijn werk gedaan.

Hoog over bekeken ziet de werking van de body er zo uit:

```
(let (bind-oldbuf-to-value-of-current-buffer)
  (save-excursion                       ; Bijhouden van het buffer.
    change-buffer
    insert-substring-from-oldbuf-into-buffer)

  schakel-terug-naar-oorspronkelijke-buffer-indien-gereed
laat-de-lokale-betekenis-van-oldbuf-verdwijnen-indien-gereed
```

Samengevat werkt `append-to-buffer` als volgt: het bewaart de waarde van het huidige buffer in de variabele met de naam `oldbuf`. Het krijgt de nieuwe buffer (en maakt er zo nodig een aan) en schakelt de aandacht van Emacs daar naar toe. Met de waarde van `oldbuf` voegt het de tekst van de region van het oude buffer in het nieuwe in. Daarna brengt het je met `save-excursion` terug naar je oorspronkelijke buffer.

Met het kijken naar `append-to-buffer`, heb je een behoorlijk complexe functie verkend. Het toont het gebruik van `let` en `save-excursion` en hoe naar een ander buffer te schakelen en terug te komen. Veel functiedefinities gebruiken `let`, `save-excursion` en `set-buffer` op deze manier.

4.5 Terugblik

Hier is een korte samenvatting van de verschillende in dit hoofdstuk besproken functies.

`describe-function`
`describe-variable`
: Toon de documentatie voor een functie of variabele. Gewoonlijk gebonden aan*C-h f* en *C-h v*.

`xref-find-definitions`
: Vindt het bestand dat de broncode voor een functie of variabele bevat en schakel buffers daar naar toe en plaats point aan het begin van dat item. Gewoonlijk gebonden aan *M-.* (dat is een punt volgend op de `META` toets).

`save-excursion`
: Bewaar de lokatie van point en herstel zijn waarde nadat de argumenten van `save-excursion` zijn geëvalueerd. Onthoud ook het huidige buffer en keer daar naar terug.

`push-mark` Stel mark in op een lokatie en bewaarde waarde van voorgaande mark op de mark-ring. De mark is een lokatie in het buffer dat zijn relatieve positie behoudt, zelfs wanneer in het buffer tekst is toegevoegd of verwijderd.

`goto-char` Zet point op de lokatie gespecificeerd door de waarde van het argument, dat een getal kan zijn, een marker, of een expressie die ene getal of een positie teruggeeft, zoals `(point-min)`.

`insert-buffer-substring`
: Kopieer een region met tekst van een buffer dat als argument is doorgegeven aan de functie en voeg de region in het huidige buffer in.

`mark-whole-buffer`
: Markeer het gehele buffer als region. Normaal gebonden aan *C-x h*.

`let*` Declareer een lijst met variabelen en geef die een initiële waarde. Evalueer de rest van de expressies in de body van `let*`. De waarden van

de variabelen kunnen gebruikt worden om daaropvolgende variabelen in de lijst te binden.

`set-buffer`
: Richt de aandacht van Emacs op een ander buffer, maar wijzig het getoonde venster niet. Wordt gebruikt wanneer een programma in plaats van een mens op een ander buffer werkt.

`get-buffer-create`
`get-buffer`
: Vindt een genoemd buffer of maak er een wanneer een buffer met die naam nog niet bestaat. De functie `get-buffer` geeft `nil` terug wanneer het genoemde buffer niet bestaat.

4.6 Oefeningen

- Schrijf je eigen `versimpelde-beginning-of-buffer` functiedefinitie. Test daarna om te kijken of het werkt.
- Gebruik `if` en `get-buffer` om een functie te schrijven die een boodschap toon die aangeeft of een buffer wel of niet bestaat.
- Zoek met `xref-find-definitions` de bron voor de functie `copy-to-buffer`.

5 Enkele meer complexe functies

In dit hoofdstuk bouwen we voort op wat in de vorige hoofdstukken hebben geleerd door naar meer complexe functies te kijken. De functie `copy-to-buffer` illustreert het gebruik van twee `save-excursion` expressies in één definitie, terwijl de `insert-buffer` het gebruik van een ster in een `interactive` expressie, het gebruik van `or`, en het belangrijke onderscheid tussen een naam en het object waar die aan refereert illustreert.

5.1 De definitie van `copy-to-buffer`

Wanneer je begrijpt hoe `append-to-buffer` werkt, is het eenvoudig `copy-to-buffer` te begrijpen. Deze functie kopieert tekst naar een buffer, maar in plaats van het aan het tweede buffer toe te voegen vervangt het alle voorgaande tekst in het tweede buffer.

De body van `copy-to-buffer` ziet er zo uit.

```
...
(interactive "BCopy to buffer: \nr")
(let ((oldbuf (current-buffer)))
  (with-current-buffer (get-buffer-create buffer)
    (barf-if-buffer-read-only)
    (erase-buffer)
    (save-excursion
      (insert-buffer-substring oldbuf start end)))))
```

De functie `copy-to-buffer` heeft een simpeler `interactive` expressie dan `append-to-buffer`.

De definitie zegt daarna:

```
(with-current-buffer (get-buffer-create buffer) ...
```

Kijk ten eerste naar de vroegste binnenste expressie, die wordt het eerste geëvalueerd. De expressie start met `get-buffer-create buffer`. De functie vertelt de computer het buffer te gebruiken met de naam gespecificeerd als diegene waarnaar je wilt kopiëren, of die aan te maken als die niet bestaat. Daarna evalueert de functie `with-current- buffer` zijn body met dat als tijdelijk huidige buffer, waarna het terugschakelt naar de buffer waar wij nu zijn[1].

(Dit demonstreert een andere manier om de aandacht van de computer maar niet die van de gebruiker te schakelen. De functie `append-to-buffer` toonde dit te doen met `save-excursion` and `set-buffer`. `with-current-buffer` is een nieuwer en misschien zelfs makkelijker mechanisme.)

De functie `barf-if-buffer-read-only` stuurt je een foutmelding die zegt dat het buffer read-only is wanneer je het niet mag wijzigen.

De volgende regel heeft de functie `erase-buffer` als enige inhoud. Het wist het buffer.

[1] Dit is vergelijkbaar met `(save- excursion (set-buffer ...) ...)` in één keer aanroepen, maar het is een beetje anders gedefinieerd, wat de geïnteresseerde lezer met `describe-function` kan ontdekken.

De laatste twee regels tenslotte bevatten de `save-excursion` expressie met `insert-buffer-substring` als body. De `insert-buffer-substring` expressie kopieert de tekst van het buffer waar je nu in bent (en je hebt niet gezien dat de computer zijn aandacht verplaatst heeft, waardoor je niet weet dat dit buffer nu `oldbuf` heet.)

Overigens is dit wat bedoeld wordt met "vervanging". Om tekst te vervangen wist Emacs de vorige tekst en voegt daarna de nieuwe tekst in.

In hoofdlijnen ziet de body van `copy-to-buffer` er zo uit:

```
(let (bind-oldbuf-aan-de-waarde-van-current-buffer)
    (met-het-buffer-waarheen-je-kopieert
      (maar-wis-of-kopieer-niet-naar-een-read-only-buffer)
      (erase-buffer)
      (save-excursion
        insert-substring-van-oldbuf-in-buffer)))
```

5.2 De definitie van `insert-buffer`

`insert-buffer` is weer een andere buffer-gerelateerde functie. Het commando kopieert een ander buffer *naar* het huidige buffer. Het is het omgekeerde van `append-to-buffer` of `copy-to-buffer`, deze kopiëren immers tekst *van* het huidige buffer naar een ander buffer.

Hier is een bespreking gebaseerd op de originele code. De code is vereenvoudigd in 2003 en is moeilijker te begrijpen.

(Zie Sectie 5.2.6 "Nieuwe body voor `insert-buffer`", pagina 66, voor een bespreking van de nieuwe body.

Daarnaast illustreert deze code het gebruik van `interactive` met een buffer dat *read-only* zou kunnen zijn en het belangrijke onderscheid tussen de naam van een object en het object waar daadwerkelijk aan gerefereerd wordt.

Hier is de eerdere code:

```
(defun insert-buffer (buffer)
  "Insert after point the contents of BUFFER.
Puts mark after the inserted text.
BUFFER may be a buffer or a buffer name."
  (interactive "*bInsert buffer: ")
  (or (bufferp buffer)
      (setq buffer (get-buffer buffer)))
  (let (start end newmark)
    (save-excursion
      (save-excursion
        (set-buffer buffer)
        (setq start (point-min) end (point-max)))
      (insert-buffer-substring buffer start end)
      (setq newmark (point)))
    (push-mark newmark)))
```

Net als met andere functiedefinities kun je een sjabloon gebruiken voor een overzicht van de functie.

```
(defun insert-buffer (buffer)
  "documentatie..."
  (interactive "*bInsert buffer: ")
  body...)
```

5.2.1 De interactieve expressie in `insert-buffer`

In `insert-buffer` heeft het argument van de `interactive` declaratie twee onderdelen, een sterretje, '`*`' en '`bInsert buffer: `'.

Een read-only buffer

Het sterretje is voor de situatie dat het huidige buffer een read-only buffer is—een buffer dat niet kan worden gewijzigd. Wanneer `insert-buffer` wordt aangeroepen terwijl het huidige buffer read-only is, wordt een boodschap hiervan in het echogebied getoond en de terminal kan piepen of knipperen, het is niet toegestaan iets in het huidige buffer in te voegen. Het sterretje hoeft niet opgevolgd te worden met een nieuwe regel om het van het volgende argument te scheiden.

'`b`' in een interactive expressie

Het volgende argument in de interactive-expressie begint met een kleine letter '`b`'. (Dit is verschillend met de code voor `append-to- buffer`, die gebruikt een hoofdletter '`B`'.) De kleine letter '`b`' vertelt de Lisp interpreter dat het argument voor `insert-buffer` een bestaand buffer moet zijn, of de naam daarvan. (De optie hoofdletter '`B`' is voor de mogelijkheid dat het buffer nog niet bestaat.) Emacs vraagt je naar de naam van het buffer en biedt een standaard buffer aan met ingeschakelde completie voor de naam. Wanneer het buffer niet bestaat, krijg je een boodschap "No match", en de terminal kan daarbij ook piepen.

De nieuwe en vereenvoudigde code genereert een lijst voor `interactive`. Het gebruikt de functies `barf-if-buffer-read-only` en `read-buffer`, waar we al vertrouwd mee zijn en de speciale vorm `progn` waarmee we dat nog niet zijn. (Die wordt later beschreven.)

5.2.2 De body van de `insert-buffer` functie

De body van de functie `insert-buffer` heeft twee belangrijke delen: een `or` expressie en een `let` expressie. Het doel van de `or` is zeker te stellen dat het argument van `buffer` gebonden is aan een buffer en niet slechts aan de naam van een buffer. De body van de `let` expressie bevat de code die het andere buffer in het huidige kopieert.

Globaal passen de twee expressies als volgt in de functie `insert-buffer`:

```
(defun insert-buffer (buffer)
  "documentatie..."
  (interactive "*bInsert buffer: ")
  (or ...
      ...
  (let (varlist)
      body-of-let... )
```

Om te begrijpen hoe de `or` zeker stelt dat het argument van `buffer` gebonden is aan een buffer en niet aan de naam van een buffer, is het eerst nodig de functie `or` te begrijpen.

Laat mij eerst, voor dat we dat doen, dit deel van de functie met `if` herschrijven, zodat je kunt zien wat er gebeurt op een manier die vertrouwd is.

5.2.3 `insert-buffer` met een `if` in plaats van een `or`

De taak die gedaan moet worden is zorgen dat de waarde van `buffer` echt een buffer en niet de naam van een buffer. Wanneer de waarde de naam is, dan moet het buffer zelf worden verkregen.

Stel je voor dat je op een conferentie bent en de ceremoniemeester loopt rond met een lijst waar je naam op staat en is op zoek naar je: de ceremoniemeester is gebonden aan je naam, niet aan jou, maar wanneer die je vindt en je bij je arm neemt, raakt de ceremoniemeester gebonden aan jou.

In Lisp kan je deze situatie als volgt beschrijven:

```
(if (not (gast-vasthouden))
    (vind-en-neem-gast-bij-de arm))
```

Wij willen hetzelfde doen met een buffer—wanneer we niet het buffer zelf hebben, willen we het krijgen.

Met een predicate met de naam `bufferp` die ons vertelt of we een buffer (in plaats van een naam) hebben schrijven we de code als volgt:

```
(if (not (bufferp buffer))              ; if-deel
    (setq buffer (get-buffer buffer)))  ; dan-deel
```

De waar-of-onwaar-test van de `if` expresse is hier `(not (bufferp buffer))` en het dan-deel is de expressie `(setq buffer (get-buffer buffer))`.

In de test geeft de functie `bufferp` waar terug wanneer het argument een buffer is—maar onwaar wanneer het argument de naam van een buffer is. (Het laatste karakter van de functienaam `bufferp` is het karakter '`p`', zoals we eerder zagen is het gebruik van '`p`' een conventie die aangeeft dat de functie een predicate is, wat een term is die betekent dat de vaststelt of een bepaalde eigenschap waar of onwaar is. Zie Sectie 1.8.4 "Het verkeerde object type als argument gebruiken", pagina 13.)

De functie `not` gaat vooraf aan de expressie `(bufferp buffer)`, waardoor de waar-of-onwaar-test er zo uitziet:

```
(not (bufferp buffer))
```

`not` is een functie die waar teruggeeft wanneer zijn argument onwaar is en onwaar als zijn argument waar is. Dus wanneer `(bufferp buffer)` waar teruggeeft, geeft de `not` onwaar terug, en omgekeerd.

Met deze test werkt de `if` expressie als volgt: wanneer de waarde van de variabele `buffer` inderdaad een buffer is in plaats van zijn naam, geeft de waar-of-onwaar-test onwaar terug en evalueert de `if` expressie het dan-deel niet. Dat is prima, omdat we niets hoeven te doen met de variabele `buffer` wanneer die werkelijk een buffer is.

Aan de andere kant, wanneer de waarde van `buffer` geen buffer is, maar de naam van een buffer, geeft de waar-of-onwaar-test waar terug en wordt het dan-deel geëvalueerd. In dit geval is het dan-deel `(setq buffer (get-buffer buffer))`. Deze expressie gebruikt de functie `get-buffer` om op basis van de naam het buffer zelf terug te geven. De `setq` stelt vervolgens de variabele `buffer` in op de waarde van het buffer zelf en vervangt zo de waarde (die de naam van het buffer was).

5.2.4 De `or` in de body

Het doel van de `or` expressie in de functie `insert-buffer` is zeker te stellen dat het argument van `buffer` gebonden is aan een buffer en niet slechts aan de naam van een buffer. De vorige sectie liet zien hoe deze taak met een `if` expressie gedaan kan worden. Echter, de functie `insert-buffer` gebruikt `or`. Om dit te begrijpen, is het nodig te begrijpen hoe `or` werkt.

Een `or` functie kan allerlei argumenten hebben. Het evalueert elk argument op zijn beurt en geeft de waarde terug van het eerste van de argumenten dat niet `nil` is. Dus, en dit is een cruciale eigenschap van `or`, evalueert het geen verdere argumenten na het teruggeven van de eerste non-`nil` waarde.

De `or` expressie ziet er zo uit:

```
(or (bufferp buffer)
    (setq buffer (get-buffer buffer)))
```

Het eerste argument voor `or` is de expressie `(bufferp buffer)`. Deze expressie geeft waar terug (een non-`nil` waarde) wanneer het buffer inderdaad een buffer is en niet slechts de naam van een buffer. De `or` expressie, wanneer dit het geval is, geeft deze waar waarde terug en evalueert de volgende expressie niet—en dat is prima voor ons, omdat we niets anders willen dan de waarde van `buffer` wanneer die echt een buffer is.

Anderzijds, als de waarde van `(bufferp buffer)` `nil` is, wat het is wanneer de waarde van `buffer` de naam van buffer is, evalueert de Lisp interpreter het volgende element van de `or` expressie. Dit is de expressie `(setq buffer (get-buffer buffer))`. Deze expressie geeft een non-`nil` waarde terug, wat de waarde is waarop het de variabele `buffer` instelt—en deze waarde is het buffer zelf, niet de naam.

Het resultaat van dit alles is dat het symbool `buffer` altijd gebonden is aan een buffer in plaats van aan de naam van een buffer. Dit is allemaal noodzakelijk omdat

de functie `set-buffer` op een volgende regel alleen werkt met een buffer, niet met de naam van een buffer.

Overigens, met `or` is de situatie met de ceremoniemeester als volgt:

```
(or (heeft-arm-van-gast) (vind-en-neem-de-arm-van-de-gast))
```

5.2.5 De `let` expresse in `insert-buffer`

Na zeker stelling dat de variabele `buffer` naar een buffer en niet slechts naar de naam van een buffer refereert, gaat de functie `insert-buffer` verder met de `let` expressie. Deze specificeert drie lokale variabelen, `start`, `end` en `newmark` en bindt ze aan de initiele waarde `nil`. Deze variabelen worden binnen de rest van de `let` gebruikt en verbergen tijdelijk elke andere variabele met eenzelfde naam in Emacs tot het einde van de `let`.

De body van de `let` bevat twee `save-excursion` expressies. Eerst kijken we in detail naar de binnenste `save-excursion` expressie. De expressie ziet er zo uit:

```
(save-excursion
  (set-buffer buffer)
  (setq start (point-min) end (point-max)))
```

De expressie `(set-buffer buffer)` schakelt de aandacht van Emacs van het huidige buffer naar een waarvan de tekst gekopieerd zal worden. In dat buffer zijn met het gebruik van de commando's `point-min` en `point-max` de variabelen `start` en `end` ingesteld op het begin en het eind van het buffer. Merk op dat we hier zien hoe `setq` twee variabelen in een enkele expressie kan instellen. Het eerste argument van `setq` is ingesteld op de waarde van de tweede, en het derde argument op de waarde van de vierde.

Nadat de body van de binnenste `save-excursion` is geëvalueerd herstelt `save-excursion` het oorspronkelijke buffer, maar `start` en `eind` blijven ingesteld op de waarden van het begin en eind van het buffer van waar de tekst gekopieerd wordt.

De buitenste `save-excursion` expressie ziet er zo uit:

```
(save-excursion
  (binnenste-save-excursion-expressie
     (ga-naar-nieuw-buffer-en-zet-start-en-end)
  (insert-buffer-substring buffer start end)
  (setq newmark (point)))
```

De functie `insert-buffer-substring` kopieert de tekst *in* het huidige buffer *van* de region aangeduid met `start` en `eind` in `buffer`. Omdat het geheel van het tweede buffer tussen `start` en `eind` ligt, wordt het geheel van het tweede buffer gekopieerd naar het buffer dat je aan het editen bent. Vervolgens wordt de waarde van point, die aan het eind van de ingevoegde tekst is, vastgelegd in de variabele `newmark`.

Nadat de body van de buitenste `save-excursion` is geëvalueerd, wordt point naar zijn oorspronkelijke plek teruggeplaatst.

Het is echter handig om de mark aan het eind van de nieuw ingevoegde tekst te plaatsen en point aan het begin. De variabele `newmark` registreert het eind van de ingevoegde tekst. In de laatste regel van de `let` expressie zet de `(push-mark newmark)` expressie op deze lokatie. (De vorige plek van de mark is nog steeds bereikbaar: het is opgeslagen in de mark-ring en je kunt teruggaan met `C-u C-SPC`.) Ondertussen staat point aan het begin van de ingevoegde tekst, waar deze was

voordat je de insert-functie aanriep, die plek was bewaarde met de eerste `save-excursion`.

De hele `let` ziet er zo uit:

```
(let (start end newmark)
  (save-excursion
    (save-excursion
      (set-buffer buffer)
      (setq start (point-min) end (point-max)))
    (insert-buffer-substring buffer start end)
    (setq newmark (point)))
  (push-mark newmark))
```

De functie `insert-buffer` gebruikt, net als de functie `append-to-buffer`, `let`, `save-excursion` en `set-buffer`. Daarnaast illustreert de functie een manier om `or` te gebruiken. Al deze functies zijn bouwblokken die we keer op keer zullen gebruiken.

5.2.6 Nieuwe body voor `insert-buffer`

De body in de GNU Emacs 22 versie is meer verwarrend dan het orgineel.

Het bestaat uit twee expressies

```
(push-mark
 (save-excursion
   (insert-buffer-substring (get-buffer buffer))
   (point)))

 nil
```

behalve, en dit is wat veel beginners in de war brengt, wordt veel werk verricht in de `push-mark` expressie.

De functie `get-buffer` geeft een buffer met de opgegeven naam terug. Merk op dat de functie *niet* `get-buffer-create` heet, het maakt geen buffer wanneer er geen bestaat. De buffer die `get-buffer` teruggeeft, een bestaand buffer, wordt doorgegeven aan `insert-buffer-substring`, de het gehele buffer invoegt (omdat je niets anders opgaf).

De plek waar het buffer is ingevoegd is vastgelegd door `push-mark`. Daarna geeft de functie `nil` terug, de waarde van het laatste commando. Anders gezegd, de functie `insert-buffer` bestaat alleen om een zij-effect te produceren, invoegen in een ander buffer en niet om enige waarde terug te geven.

5.3 Complete definitie van `beginning-of-buffer`

De basisstructuur van de functie `beginning-of-buffer` hebben we eerder besproken. (Zie Sectie 4.2 “Een versimpelde `beginning-of-buffer` definitie”, pagina 49.) Deze sectie beschrijft het complexe deel van de definitie.

Wanneer `beginning-of-buffer` zonder argument wordt aangeroepen plaatst het, zoals eerder beschreven, de cursor naar het begin van het buffer (eerlijk gezegd, het begin van het bereikbare deel van het buffer), en laat de mark op de voorgaande positie. Echter, wanneer het commando wordt aangeroepen met een getal tussen één en tien, dan beschouwt de functie dat getal als een fractie van de lengte van het buffer, gemeten in tienden, en verplaatst Emacs de cursor die fractie vanaf het begin

van het buffer. Dus je kunt de functie aanroepen met het commando *M-<*, wat de cursor naar het begin van het buffer verplaatst, of met een toets commando zoals *C-u 7 M-<* wat de cursor op een punt 70% in het buffer plaatst. Bij gebruik van een getal groter dan tien als argument gaat het naar het einde van het buffer.

De functie `beginning-of-buffer` kan met of zonder argument worden aangeroepen. Het gebruik van een argument is optioneel.

5.3.1 Optionele argumenten

Tenzij anders verteld verwacht Lisp dat een functie met een argument in zijn functiedefinitie wordt aangeroepen met een waarde voor dat argument. Wanneer dat niet gebeurt, dan krijg je een fout en de boodschap '`Wrong number of arguments`'.

Optionele argumenten zijn echter een eigenschap van Lisp: een specifiek *sleutelwoord* wordt gebruikt om aan de Lisp interpreter te vertellen dat een argument optioneel is. Het sleutelwoord is `&optional`. (De '`&`' aan de voorkant van '`optional`' is onderdeel van het sleutelwoord.) Wanneer een argument in een functiedefinitie volgt op het sleutelwoord `&optional` dan behoeft geen waarde aan het argument te worden doorgegeven bij het aanroepen van de functie.

De eerste regel van de functiedefinitie van `beginning-of-buffer` ziet er daardoor als volgt uit:

```
(defun beginning-of-buffer (&optional arg)
```

Globaal ziet de hele functie er zo uit:

```
(defun beginning-of-buffer (&optional arg)
  "documentatie..."
  (interactive "P")
  (or (is-het-argument-een-cons-cel arg)
      (and zijn-zowel-transient-mark-mode-als-mark-active-waar)
      (push-mark))
  (let (stel-grootte-vast-en-stel-in)
    (goto-char
      (als-er-een-argument-is
          zoek-uit-waar-heen-te-gaan
        anders-ga-naar
        (point-min))))
  wees-beleefd
```

De functie is vergelijkbaar met de `versimpelde-beginning-of-buffer` functie, behalve dat de `interactive` expressie als argument `"P"` heeft en de functie `goto-char` gevolgd wordt door een if-then-else expressie die uitzoekt waar de cursor geplaatst moet worden wanneer er een argument is dat geen cons-cel is.

(Omdat ik een cons-cel pas een aantal hoofdstukken verderop uitleg, negeer alsjeblieft de functie `consp`. Zie Hoofdstuk 9 "Hoe lijsten zijn geïmplementeerd", pagina 108, en Sectie "Cons Cell and List Types" in *The GNU Emacs Lisp Reference Manual*.)

De `"P"` in de `interactive` expressie vertelt Emacs om een prefix-argument, wanneer die er is, ongewijzigd aan de functie door te geven. Een prefix-argument maak je het typen van de `META`-toets, gevolgd door een getal, of door *C-u* te typen en dan een getal. (Als je geen getal typt, dan leidt *C-u* standaard naar een cons-cel

met een 4). De kleine letter `"p"` in de `interactive` expressie laat de functie de prefix naar een getal converteren.

De waar-of-onwaar-test in de `if`-expressie ziet er gecompliceerd uit, maar is het niet: het controleert of `arg` een waarde heeft die niet `nil` is en of het wel of niet een cons-cel is. (Dat is wat `consp` doet, het controleert of zijn argument een cons-cel is.) Wanneer `arg` een waarde heeft die ongelijk aan `nil` is (en ook geen cons-cel is), wat het geval is wanneer `beginning-of-buffer` wordt aangeroepen met een numeriek argument, dan geeft deze waar-of-onwaar-test waar terug en wordt het dan-deel van de `if`-expressie geëvalueerd. Anderzijds, wanneer `beginning-of-buffer` niet met een argument is aangeroepen, heeft `arg` de waarde `nil` en wordt het anders-deel van de `if`-expressie geëvalueerd. Dit anders-deel is eenvoudigweg `point-min`, en wanneer dit de uitkomst is, dan is de gehele `goto-char`-expressie `(goto-char (point-min))`, wat overeenkomt met hoe we de functie `beginning-of-buffer` zagen in de vereenvoudigde vorm.

5.3.2 `beginning-of-buffer` met een argument

Wanneer `beginning-of-buffer` wordt aangeroepen met een argument, dan wordt een expressie geëvalueerd die de waarde berekent om aan `goto-char` door te geven. Deze expressie is op het eerste gezicht nogal complex. Het bevat een binnenliggende `if` en flink wat rekenkunde. Het ziet er zo uit:

```
(if (> (buffer-size) 10000)
    ;; Avoid overflow for large buffer sizes!
    (* (prefix-numeric-value arg)
       (/ size 10))
  (/
   (+ 10
      (* size
         (prefix-numeric-value arg)))
   10))
```

Net als andere expressies die er complex uitzien, kan de conditionele expressie in `beginning-of-buffer` worden ontrafeld door het te bekijken als delen van een sjabloon, in dit geval het sjabloon voor een if-then-else-expressie. Globaal ziet de expressie er zo uit:

```
(if (buffer-is-groot
    deel-buffer-grootte-door-10-en-vermenigvuldig-met-arg
  anders-gebruik-alternatieve-berekening
```

De waar-of-onwaar-test van de binnenliggende `if`-expressie controleert de grootte van het buffer. De reden hiervoor is, is dat de oude versie 18 van Emacs met getallen werkte die niet hoger kunnen zijn dan ongeveer acht miljoen en in de berekening die volgt, de programmeur bang was dat Emacs zou proberen te grote getallen te gebruiken wanneer de buffers groot zijn. De term “overflow” die in het commentaar genoemd wordt betekent dat getallen te groot zijn. Meer recente versies van Emacs gebruiken grotere getallen, maar de code is niet gewijzigd, al is het alleen maar omdat mensen nu met buffers werken die vele malen groter zijn dan ooit.

Er zijn twee gevallen: wanneer de buffer groot is, en wanneer niet.

Wat in een groot buffer gebeurt

In In `beginning-of-buffer` test de binnenliggende `if`-expressie of de grootte van het buffer groter is dan 10.000 karakters. Om dit te doen gebruikt het de functie `>` en de berekening van `size` die van de let-expressie komt.

Oorspronkelijk werd de functie `buffer-size` gebruikt. Niet alleen werd die functie meerdere keren aangeroepen, het gaf de grootte van het gehele buffer, niet het bereikbare deel. De berekening is veel logischer wanneer het alleen het bereikbare deel gebruikt. (Zie Hoofdstuk 6 "Versmallen en verbreden", pagina 74, voor meer informatie over het richten van de aandacht op een bereikbaar deel.)

De regel ziet er zo uit:

```
(if (> size 10000)
```

Wanneer het buffer groot is, dan wordt het dan-deel van de `if`-expressie geëvalueerd. Dit staat er (na het formatteren voor makkelijk lezen):

```
(*
  (prefix-numeric-value arg)
  (/ size 10))
```

Deze expressie is een vermenigvuldiging met twee argumenten voor de functie `*`.

Het eerste argument is `(prefix-numeric-value arg)`. Wanneer `"P"` wordt gebruik als argument voor `interactive`, is de waarde die als argument wordt doorgegeven een *ongewijzigd prefix-argument*, en niet een getal. (Het is een getal in een list.) Om de berekening uit te voeren is een conversie nodig, en `prefix-numeric-value` doet dat.

Het tweede argument is `(/ size 10)`. Deze expressie deelt het getal door tien—de numerieke waarde van de grootte van het bereikbare deel van de buffer. Dit produceert een getal dat vertelt hoeveel karakters een tiende van de buffergrootte vormen. (In Lisp wordt `/` gebruikt voor deling, net als `*` wordt gebruikt voor vermenigvuldiging.)

In de complete vermenigvuldiging expressie wordt dit getal vermenigvuldigd met de waarde van het prefix-argument—de vermenigvuldiging ziet er zo uit:

```
(* numerieke-waarde-van-het-prefix-arg
   aantal-karakters-in-een-tiende-van-het-bereikbare-buffer)
```

Als bijvoorbeeld het prefix-argument '7' is, wordt de een-tiende waarde vermenigvuldigd met 7, om op een positie van 70% uit te komen.

Het resultaat van dit alles is dat wanneer het bereikbare deel van het buffer groot is, de `goto-char` als volgt is:

```
(goto-char (* (prefix-numeric-value arg)
              (/ size 10)))
```

Dit plaatst de cursor waar wij die willen hebben.

Wat in een klein buffer gebeurt

Wanneer het buffer minder dan 10.000 karakters bevat wordt een iets andere berekening uitgevoerd. Je zou kunnen denken dat dit niet nodig is, omdat de eerste berekening dat zou kunnen doen. Maar in een klein buffer zou de eerste methode de cursor niet precies op de gewenste regel kunnen plaatsen. De tweede methode doet dat beter.

De code ziet er zo uit:

```
(/ (+ 10 (* size (prefix-numeric-value arg))) 10)
```

Dit is de code waarin je uitzoekt wat er gebeurt door te ontdekken hoe de functies tussen de haakjes zijn ingebed. Het wordt makkelijker te lezen wanneer je het zo formatteert dat elke expressie dieper inspringt dan zijn omsluitende expressie:

```
(/
 (+ 10
    (*
     size
     (prefix-numeric-value arg)))
 10)
```

Naar de haakjes kijkend zien we dat de meest binnenste bewerking `(prefix-numeric-value arg)` is, die het ongewijzigde argument naar een getal converteert. In de volgende expressie wordt dit getal vermenigvuldigd met de grootte van het bereikbare deel van het buffer:

```
(* size (prefix-numeric-value arg))
```

Deze vermenigvuldiging creëert een getal dat groter zou kunnen zijn dan de grootte van het buffer—zeven keer groter als het argument 7 is, bijvoorbeeld. Hierna wordt tien bij dit getal opgeteld en uiteindelijk wordt het grote getal gedeeld door tien om een waarde te geven die één karakter groter is dan de procentuele positie in het buffer.

Het getal dat de uitkomst is van dit alles wordt doorgegeven aan `goto-char` en de cursor wordt naar dat punt verplaatst.

5.3.3 De complete `beginning-of-buffer`

Hiet is de complete tekst van de functie `beginning-of-buffer`:

```
(defun beginning-of-buffer (&optional arg)
  "Move point to the beginning of the buffer;
leave mark at previous position.
With \\[universal-argument] prefix,
do not set mark at previous position.
With numeric arg N,
put point N/10 of the way from the beginning.

If the buffer is narrowed,
this command uses the beginning and size
of the accessible part of the buffer.

Don't use this command in Lisp programs!
\(goto-char (point-min)) is faster
and avoids clobbering the mark."
  (interactive "P")
  (or (consp arg)
      (and transient-mark-mode mark-active)
      (push-mark))
  (let ((size (- (point-max) (point-min))))
    (goto-char (if (and arg (not (consp arg)))
                   (+ (point-min)
                      (if (> size 10000)
                          ;; Avoid overflow for large buffer sizes!
                          (* (prefix-numeric-value arg)
                             (/ size 10))
                        (/ (+ 10 (* size (prefix-numeric-value arg)))
                           10)))
                 (point-min))))
  (if (and arg (not (consp arg))) (forward-line 1)))
```

Afgezien van twee kleine punten toont de voorgaande bespreking hoe deze functie werkt. Het eerste punt betreft een detail in de documentatiestring en het tweede punt betreft de laatste regel van de functie.

In de documentatiestring is een referencie naar een expressie:

`\\[universal-argument]`

Een '`\\`' wordt gebruikt voor de eerste vierkante haak in deze expressie. Deze '`\\`' vertelt de Lisp interpreter de toets te vervangen door wat dan ook thans gebonden is aan de '`[...]`'. In het geval van het universele argument, die meestal gebonden is aan `C-u`, maar dit kan anders zijn (Zie Sectie "Tips for Documentation Strings" in *The GNU Emacs Lisp Reference Manual*, for more information.)

Tenslotte zegt de laatste regel van het `beginning-of-buffer` commando de point te verplaatsten naar het begin van de volgende regel wanneer het commando met een argument is aangeroepen:

```
(if (and arg (not (consp arg))) (forward-line 1))
```

Dit plaatst de cursor aan het begin van de eerste regel na de juiste tiende deel positie in het buffer. Deze elegantie zorgt dat de cursor altijd *tenminste* het aantal gewenste tienden in het buffer is geplaatst, wat een misschien onnodige verfraaiing is, die, als dat niet zou gebeuren ongetwijfeld tot klachten zou leiden. (Het `(not (consp arg))` deel is zo dat wanneer je het commando met `C-u` maar zonder getal aanroept, wat inhoudt dat het prefix-argument eenvoudig een cons-cel is, het commando je niet aan het begin van de tweede regel plaatst.)

5.4 Terugblik

Hier is een korte samenvatting van enkele van de onderwerpen in dit hoofdstuk.

`or` Evalueer elk argument op volgorde, en geef de waarde van het eerste argument dat niet `nil` is. Wanneer niet een van hen een waarde ongelijk aan `nil` teruggeeft, geef dan `nil` terug. In het kort, geef de eerste waarde van de argumenten die waar is terug, geef een waarde waar terug wanneer een *of* meer van de andere waar zijn.

`and` Evalueer elk argument op volgorde, en als er een `nil` is, geef dan `nil` terug. Als niet een van hen `nil` is, geef dan de waarde van het laatste argument terug. In het kort, geef een waarde waar terug uitsluitend wanneer alle argumenten waar zijn, geef een waarde waar terug wanneer een *en* alle andere waar zijn.

`&optional` Een sleutelwoord dat aangeeft dat een argument van een functiedefinitie optioneel is, dit betekent dat de functie desgewenst zonder dit argument kan worden geëvalueerd,

`prefix-numeric-value`
Converteer het ongewijzigde prefix-numeric-value geproduceerd door `(interactive "P")` naar een numerieke waarde.

`forward-line`
Verplaats point voorwaarts naar het begin van de volgende regel, of wanneer het argument groter is dan één, dat aantal regels voorwaarts. Wanneer het niet zo ver voorwaarts kan als het bedoeld is, gaat `forward-line` zo ver mogelijk voorwaarts en geeft vervolgens het aantal terug van de extra regels die het bedoeld was te verplaatsen maar dat niet kon.

`erase-buffer`
Verwijder de gehele inhoud van het huidige buffer.

`bufferp` Geef `t` terug wanneer het argument een buffer is, zo niet geef `nil` terug.

5.5 `optioneel` argument oefening

Schrijf een interactieve functie met een optioneel argument dat test of het argument, een getal, groter of gelijk is aan, dan wel kleiner is dan de waarde van `fill-column`, en je in een boodschap vertelt welke het geval is. Wanneer je echter geen argument aan de functie doorgeeft, gebruik dan 56 als standaard waarde.

6 Versmallen en verbreden

Versmallen is een eigenschap van Emacs die het mogelijk maakt je aandacht te richten op een specifiek deel van een buffer, en te werken zonder dat je per ongeluk iets in de andere delen wijzigt. Versmallen is normaal gesproken uitgeschakeld omdat het verwarrend is voor beginners.

Met versmallen is de rest van het buffer onzichtbaar, alsof het er niet is. Dit is een voordeel wanneer je bijvoorbeeld een woord in een deel van het buffer wilt vervangen, maar niet in een ander deel: je versmalt naar het deel dat je wilt en de vervanging wordt alleen in dat deel uitgevoerd en niet in de rest van het buffer. Zoeken werkt ook binnen een versmald gebied, niet daarbuiten, zodat als je een deel van een document aan het verbeteren bent, je jezelf beschermt tegen per ongeluk vinden van delen die je niet wilt verbeteren door te versmallen naar precies dat gebied dat je wilt. (De toetscombinatie voor `narrow-to-region` is *C-x n n*.)

Versmallen maakt echter de rest van het buffer onzichtbaar, wat mensen bang kan maken die onbedoeld versmalling aanroepen en denken dat ze een deel van het buffer verwijderd hebben. Verder zet het `undo` commando (wat normaal gebonden is aan *C-x u*) het versmallen niet uit (en moet dat ook niet doen), wat mensen wanhopig kan maken wanneer zij niet weten dat ze de zichtbaarheid van de rest van het buffer kunnen herstellen met het `widen` commando. (De toetscombinatie voor `widen` is *C-x n w*.)

Versmallen is net zo zinvol voor de Lisp interpreter als voor mensen. Vaak is een Emacs Lisp functie ontworpen om op slechts een deel van een buffer te werken, of omgekeerd, een Emacs Lisp functie moet werken op het gehele buffer dat is versmald. De functie `what-line` bijvoorbeeld, haalt de versmalling van een buffer als die versmald is en wanneer het zijn taak gedaan heeft, herstelt de versmalling naar wat het was. Anderzijds gebruikt de functie `count-lines` versmalling om zichzelf te beperken tot dat deel van het buffer waarin het geïnteresseerd is en herstelt daarna de vorige situatie.

6.1 De speciale vorm `save-restriction`

In Emacs Lisp gebruik je de speciale vorm `save-restriction` om bij te houden welke versmalling van toepassing is. Wanneer de Lisp interpreter `save-restriction` tegenkomt, voert het de code in de body van de `save-restriction` expressie uit en maakt de eventuele wijzigingen die het in de versmalling heeft aangebracht ongedaan. Wanneer bijvoorbeeld het buffer versmald is en de code die volgt op `save-restriction` de versmalling weghaalt, herstelt `save-restriction` na afloop het buffer naar zijn versmalde region. In het `what-line` commando wordt de versmalling die het buffer mogelijk heeft ongedaan gemaakt met het `widen` commando, dat onmiddellijk volgt op het `save-restriction`. De eventuele oorspronkelijke versmalling wordt hersteld vlak voor het beëindigen van de functie.

Het sjabloon voor een save-restriction expressie is eenvoudig:

```
(save-restriction
  body... )
```

De body van de save-restriction bestaat uit een of meer expressies die de Lisp interpreter op volgorde evalueert.

Tenslotte, een punt om op te merken: wanneer je zowel save-excursion en save-restriction gebruikt, de een na de ander, dan moet je save-excursion als de buitenste gebruiken. Wanneer je ze in omgekeerde volgorde schrijft, kan het bijhouden van de versmalling in het buffer waar Emacs naar toe schakelt na het aanroepen van save-excursion fout gaan. Wanneer je daarom samen gebruikt, schrijf je save-excursion en save-restriction als volgt:

```
(save-excursion
  (save-restriction
    body...))
```

In andere omstandigheden, waar ze niet samen gebruikt worden, schrijf je de speciale vormen save-excursion en save-restriction in de volgorde die past bij de functie.

Bijvoorbeeld,

```
(save-restriction
  (widen)
  (save-excursion
  body...))
```

6.2 what-line

Het what-line commando vertelt het je het regelnummer waarop de cursor staat. De functie illustreert het gebruik van de save-excursion en save-restriction commando's. Hier is de oorspronkelijke tekst van de functie.

```
(defun what-line ()
  "Print the current line number (in the buffer) of point."
  (interactive)
  (save-restriction
    (widen)
    (save-excursion
      (beginning-of-line)
      (message "Line %d"
               (1+ (count-lines 1 (point)))))))
```

(In moderne versies van GNU Emacs is de functie what-line uitgebreid zodat het zowel het regelnummer in het versmalde buffer als het regelnummer in het verbrede buffer vertelt. De moderne versie is complexer dan de versie die we hier laten zien. Wanneer je een avontuurlijke bui hebt, kan je er naar kijken nadat je uitgezocht hebt hoe deze versie werkt. Je hebt daar waarschijnlijk *C-h f* (describe-function) bij nodig.) De nieuwere versie gebruikt een conditional om vast te stellen of het buffer is versmald.

Verder maakt de moderne versie van what-line gebruik van line-number-at-pos, die net als andere eenvoudige expressies, zoals (goto-char (point-min)),

`(forward-line 0)` in plaats van `beginning-of-line` gebruikt om point naar het begin van de huidige regel te verplaatsen.

De functie `what-line` zoals hier getoond heeft zoals je zou verwachten een documentatieregel en is interactief. De volgende twee regels gebruiken de functies `save-restriction` en `widen`.

De speciale vorm `save-restriction` noteert eventuele versmalling die van kracht is in het huidige buffer en herstelt de versmalling nadat de code in de body van `save-restriction` is geëvalueerd.

De speciale vorm `save-restriction` wordt gevolgd door `widen`. Deze functie maakt eventuele versmalling in het huidige buffer ongedaan voor zover die er was toen `what-line` werd aangeroepen. (De versmalling die daar was is de versmalling die `save-restriction` onthoudt.) Deze verbreding maakt het voor de commando's die de regels tellen mogelijk om vanaf het begin van het buffer te tellen. Anders zouden ze beperkt zijn tot het tellen binnen het bereikbare gebied. Eventuele oorspronkelijke versmalling wordt hersteld vlak voor de voltooiing van de functie door de speciale vorm `save-restriction`.

De aanroep van `widen` wordt gevolgd door `save-excursion` die de lokatie van de cursor (dus van point) bewaart, en herstelt die nadat de code in de body van de `save-excursion` de functie `beginning-of-line` gebruikt om point te verplaatsen.

(Merk op dat de `(widen)` expressie tussen de speciale vormen `save-restriction` en de `save-excursion` staat. Wanneer je de twee `save- ...` expressies opeenvolgend schrijft, zet je `save-excursion` als buitenste.)

De laatste twee regels van de functie `what-line` zijn functies die het aantal regels in het buffer tellen en daarna dat aantal in het echogebied tonen.

```
(message "Line %d"
         (1+ (count-lines 1 (point)))))))
```

De functie `message` toont een eenregelige boodschap onderaan het Emacs scherm. Het eerste argument staat tussen aanhalingstekens en wordt getoond als een string karakters. Het mag echter een '`%d`' expressie bevatten om het volgende argument te tonen. '`%d`' toont het argument als een decimaal getal, zodat de boodschap iets als '`Line 243`' zegt.

Het getal dat getoond wordt op de plek van de '`%d`' wordt berekend door de laatste regel van de functie:

```
(1+ (count-lines 1 (point)))
```

Wat dit doet is het tellen van de regels vanaf de eerste positie van het buffer, aangegeven door de `1`, tot de `(point)` en telt daar vervolgens één bij op. (De functie `1+` telt één bij zijn argument op.) We tellen één er bij op omdat regel 2 maar één regel voor zich heeft, en `count-lines` alleen de regels *voor* de huidige regel telt.

Nadat `count-lines` zijn werk heeft gedaan en de boodschap in het echogebied is getoond, herstelt `save-excursion` point naar zijn oorspronkelijke positie en herstelt `save-restriction` de eventuele oorspronkelijke versmalling.

6.3 Oefening met versmalling

Schrijf een functie dat de eerste 60 karakters van het huidige buffer toont, zelfs wanneer je het buffer hebt versmald naar zijn tweede helft zodat de eerste regel onbereikbaar is. Herstel point, mark en de versmalling. Gebruik voor deze oefening de gehele potpourri van functies, inclusief `save-restriction`, `widen`, `goto-char`, `point-min`, `message` en `buffer-substring`.

(`buffer-substring` is een eerder ongenoemde functie die je zelf moet onderzoeken, of misschien moet je gebruik maken van . `buffer-substring-no-properties` of `filter-buffer-substring` ..., nog meer andere functies. Teksteigenschappen zijn een eigenschap die overigens hier niet besproken wordt. Zie Sectie "Text Properties" in *The GNU Emacs Lisp Reference Manual*.)

Daarnaast, heb je `goto-char` of `point-min` echt nodig? Of kan je de functie zonder hen schrijven?

7 car, cdr, cons: Fundamentele functies

car, cdr en cons zijn fundamentele functies in Lisp. De functie cons wordt gebruikt om lijsten te construeren, en de functies car en cdr worden gebruikt om ze uit elkaar te halen.

In de rondleiding door de functie copy-region-as-kill kijken we naar cons en naar twee varianten van cdr, namelijk setcdr en nthcdr.

De naam van de functie cons is niet onredelijk, het is een afkorting van het woord "construct". De herkomst van de name voor car en cdr aan de andere kant zijn esoterisch: car is een acroniem van de zin "Contents of the Address part of the Register", en cdr (uitgesproken als "coeld-er") is een acroniem van de zin "Contents of the Decrement part of the Register". Deze zinnen refereren aan de IBM 704 computer waarop de oorspronkelijke Lisp werd ontwikkeld.

De IBM 704 is een voetnoot in de geschiedenis, maar deze namen zijn nu geliefde tradities van Lisp.

7.1 car en cdr

De CAR van een lijst is, vrij eenvoudig, het eerste item in de lijst. Dus de CAR van de lijst (roos viool madelief boterbloem) is roos.

Wanneer je dit in Info in GNU Emacs leest kan je zien dit door het volgende te evalueren:

```
(car '(roos viool madelief boterbloem))
```

Na het evalueren van de expressie verschijnt roos in het echogebied.

car verwijdert het eerste element van de lijst niet, het vertelt alleen wat het is. Nadat car op een lijst is toegepast is de lijst nog steeds hetzelfde. car is "niet-destructief", in het jargon. Deze eigenschap blijkt belangrijk te zijn.

De CDR van een lijst is de rest van de lijst, dat wil zeggen dat de CDR het deel van de lijst dat volgt op het eerste item. Dus terwijl de CAR van de lijst '(roos viool madelief boterbloem) roos is, is de rest van de, de waarde die de functie cdr teruggeeft, is (viool madelief boterbloem).

Je ziet door het volgende op de gebruikelijke manier te evalueren:

```
(cdr '(roos viool madelief boterbloem))
```

Wanneer je dit evalueert, verschijnt (viool madelief boterbloem) in het echogebied.

Net als car verwijdert cdr geen elementen uit de lijst—het vertelt aleen wat het de tweede en volgende elementen zijn.

Overigens in het voorbeeld is de lijst met bloemen gequote. Wanneer dat niet zo was zou de Lisp interpreter de lijst proberen te evalueren door roos als een functie aan te roepen. In dit voorbeeld willen we dat niet.

Voor het werken met lijsten zouden de namen first en rest logischer zijn dan de namen car en cdr. Sommige programmeurs definieren inderdaad first en rest als aliassen voor car en cdr en gebruiken daarna first en rest in de rest van hun code.

Lijsten in Lisp worden echter gebouwd met een onderliggende structuur bekend als "cons cells" (zie Hoofdstuk 9 "Implementatie van lijsten", pagina 108), waarin niet zo iets als "eerste" of "rest" bestaat, en de CAR en de CDR symmetrisch zijn. Lisp probeert niet het bestaan van cons cellen te verstoppen en programma's gebruiken ze ook voor andere dingen dan lijsten. Om deze reden helpen de namen de programmers te herinneren dat `car` en `cdr` symmetrisch zijn, ondanks de asymmetrische manier waarop ze in lijsten gebruikt worden.

Wanneer `car` en `cdr` worden toegepast op lijsten van symbolen, zoals de lijst `(den spar eik esdoorn)`, is het element van de lijst dat de functie `car` teruggeeft het symbool `den`, zonder haakjes er omheen. `den` is het eerste symbool in de lijst. De CDR van de lijst is echter een list op zichzelf, `(spar eik esdoorn)`, zoals je ziet door de volgende expressie op de gebruikelijke manier te evalueren:

```
(car '(den spar eik esdoorn))

(cdr '(den spar eik esdoorn))
```

Anderzijds is in een lijst van lijsten het eerste element op zichzelf een lijst. `car` geeft het eerste element als lijst terug. De volgende lijst bijvoorbeeld bevat drie sublijsten, een lijst van carnivoren, een lijst van herbivoren en een lijst van zeezoogdieren:

```
(car '((leeuw tijger cheeta)
       (gazelle antiloop zebra)
       (walvis dolfijn zeehond)))
```

In dit voorbeeld is het eerste element of de CAR van de lijst de lijst met carnivoren, `(leeuw tijger cheeta)` en is de rest van de lijst `(gazelle antiloop zebra) (walvis dolfijn zeehond)`.

```
(cdr '((leeuw tijger cheeta)
       (gazelle antiloop zebra)
       (walvis dolfijn zeehond)))
```

Het is het waard nog een keer te zeggen dat `car` en `cdr` niet-destructief zijn, zij wijzigen de lijsten waarop worden toegepast niet. Dit is erg belangrijk voor hoe ze worden gebruikt.

Daarnaast zei ik bij de bespreking van atomen in het eerste hoofdstuk dat in Lisp sommige soorten atomen, zoals een array, in delen gescheiden kunnen worden maar dat het mechanisme om dit te doen afwijkt van het mechanisme om een lijst te splijten. Voor zover het Lisp aangaat, zijn de atomen van lijsten onsplijtbaar. (Zie Sectie 1.1.1 "Lisp atomen", pagina 1.) De functies `car` en `cdr` worden gebruikt om lijsten te splijten en worden als fundamenteel voor Lisp beschouwd. Aangezien zij een array niet kunnen splijten of toegang krijgen tot de delen van een array, wordt een array beschouwd als een atoom. Omgekeerd kan de andere fundamentele functie `cons` gebruikt worden om een lijst samen te stellen, maar niet een array. (Arrays worden behandeld met array-specifieke functies Zie Sectie "Arrays" in *The GNU Emacs Lisp Reference Manual*.)

7.2 cons

De cons functie construeert lijsten. Het is de inverse van car en cdr. cons kan bijvoorbeeld gebruikt worden om de lijst van vier elementen te maken van een lijst met drie elementen, bijvoorbeeld (spar eik esdoorn):

```
(cons 'den '(spar eik esdoorn))
```

Na evaluatie van de lijst zie je

```
(den spar eik esdoorn)
```

in het echogebied verschijnen. cons veroorzaakt de creatie van een nieuwe lijst waarin het element wordt gevolgd door de elementen van de oorspronkelijke lijst.

We zeggen vaak dat cons een nieuw element aan het begin van de lijst plaatst, of dat het een element aan de lijst vastmaakt of pusht, maar deze formulering is misleidend, omdat cons geen bestaande lijst verandert, maar een nieuwe maakt.

Net als car en cdr, is cons niet-destructief.

cons moet een lijst hebben om aan vast te maken[1] Je kunt niet van absoluut niets beginnen. Wanneer je een lijst bouwt, dan moet je tenminste een lege lijst als begin geven. Hier is een serie van cons expressies die een lijst van bloemen opbouwen. Wanneer je dit in Info in GNU Emacs leest, kan je elk van expressies op de gebruikelijke manier evalueren. De waarde wordt in deze tekst getoond na '⇒', wat je kunt zien als "evalueert naar".

```
(cons 'boterbloem ())
     ⇒ (boterbloem)

(cons 'madelief '(boterbloem))
     ⇒ (madelief boterbloem)

(cons 'viool '(madelief boterbloem))
     ⇒ (violet madelief boterbloem)

(cons 'roos '(viool madelief boterbloem))
     ⇒ (roos viool madelief boterbloem)
```

In het eerste voorbeeld wordt de lege lijst getoond als () en wordt lijst geconstrueerd van boterbloem gevold door de lege lijst. Zoals je ziet wordt de lege lijst niet getoond in de geconstrueerde lijst. Het enige wat je ziet is (boterbloem). De lege lijst telt niet als een element van de lijst omdat er niets is in een lege lijst. Over het algemeen is een lege lijst onzichtbaar.

Het tweede voorbeeld, (cons 'madelief '(boterbloem)) construeert een nieuwe lijst met twee elementen door madelief voor boterbloem te plaatsen, en het derde voorbeeld construeert een lijst met drie elementen door viool voor madelief en boterbloem te plaatsen.

---

[1] Eigenlijk kan je met cons een element aan een atoom consen om een dotted pair te maken. Dotted pairs worden hier niet behandeld. Zie Sectie "Dotted Pair Notation" in *The GNU Emacs Lisp Reference Manual*.

7.2.1 Achterhaal de lengte van een lijst: `length`

Je achterhaalt hoeveel elementen in een lijst zijn met de Lisp functie `length`, zoals in de volgende voorbeelden:

```
(length '(boterbloem))
     ⇒ 1

(length '(madelief boterbloem))
     ⇒ 2

(length (cons 'viool '(madelief boterbloem)))
     ⇒ 3
```

In het derde voorbeeld is de functie `cons` gebruikt om een lijst met drie elementen te construeren die daarna als argument wordt doorgegeven aan de functie `length`.

We kunnen `length` ook gebruiken om het aantal elementen in een lege lijst te tellen:

```
(length ())
     ⇒ 0
```

Zoals je zou verwachten is het aantal elementen van een lege lijst nul.

Een interessant experiment is te ontdekken wat gebeurt wanneer je probeert de lengte van totaal geen lijst te achterhalen, dus als je probeert `length` zonder enig argument aan te roepen, zelfs geen lege lijst.

```
(length )
```

Wat je ziet als je dit evalueert is de foutmelding

```
Lisp error: (wrong-number-of-arguments length 0)
```

Dit betekent dat de functie een verkeerd aantal argumenten krijgt, nul, terwijl het een ander aantal argumenten verwacht. In dit geval wordt één argument verwacht, een argument dat een lijst is wiens lengte de functie moet meten. (Merk op dat *één* lijst *één* argument is, zelfs als de lijst veel elementen in zich heeft.)

Het deel van de foutmelding dat '`length`' zegt, is de naam van de functie.

7.3 `nthcdr`

De functie `nthcdr` is geassocieerd met de functie `cdr`. Wat het doet is dat het herhaaldelijk de CDR van een lijst neemt.

Wanneer je de CDR van de lijst `(den spar eik esdoorn)` neemt, krijg je de lijst `(spar eik esdoorn)`. Wanneer je dit herhaalt op wat je terugkreeg, krijg je de lijst `(eik esdoorn)` terug. (Als je herhaaldelijk de CDR van de originele lijst neemt, dan krijg je natuurlijk de originele CDR terug, omdat deze functie de lijst niet verandert. Je moet de CDR van de CDR nemen, enzovoort.) Wanneer je hiermee doorgaat, dan krijg je uiteindelijk de lege lijst terug, welke in dit geval in plaats van `()` getoond wordt als `nil`.

Ter beoordeling is hier een lijst van herhaaldelijke CDR's, de tekst volgend op '⇒' toont wat wordt teruggegeven.

```
(cdr '(den spar eik esdoorn))
     ⇒ (spar eik esdoorn)

(cdr '(spar eik esdoorn))
     ⇒ (eik esdoorn)

(cdr '(eik esdoorn))
     ⇒ (esdoorn)

(cdr '(esdoorn))
     ⇒ nil

(cdr 'nil)
     ⇒ nil

(cdr ())
     ⇒ nil
```

Het is ook mogelijk meerdere CDR's te doen zonder de waarden er tussen te tonen.

```
(cdr (cdr '(den spar eik esdoorn)))
     ⇒ (eik esdoorn)
```

in dit voorbeeld evalueert de Lisp interpreter eerst de binnenste lijst. De binnenste lijst is gequote, en geeft dus de lijst zoals die is door aan de binnenste `cdr`. Deze `cdr` geeft de lijst, bestaande uit het tweede en volgende elementen van de lijst door aan de buitenste `cdr`, die de lijst produceert die bestaat uit het derde en volgende elementen van de oorspronkelijke lijst. In dit voorbeeld wordt de `cdr` herhaald en geeft een lijst terug die bestaat uit de oorspronkelijke lijst zonder de eerste twee elementen.

De functie `nthcdr` doet hetzelfde als een herhaalde aanroep van `cdr`. In het volgende voorbeeld wordt het argument 2 doorgegeven aan de functie `nthcdr`, samen met de lijst, en de teruggegeven waarde is de lijst zonder de eerste twee elementen, wat exact hetzelfde is als twee keer `cdr` herhalen op de lijst.

```
(nthcdr 2 '(den spar eik esdoorn))
     ⇒ (eik esdoorn)
```

Je kunt met de oorspronkelijke lijst van vier elementen zien wat gebeurt wanneer je verschillende numerieke argumenten doorgeeft aan `nthcdr`, waaronder 0, 1 en 5:

```
;; Houd de lijst zoals die was.
(nthcdr 0 '(den spar eik esdoorn))
     ⇒ (den spar eik esdoorn)

;; Geef een exemplaar terug zonder eerste element.
(nthcdr 1 '(den spar eik esdoorn))
     ⇒ (spar eik esdoorn)

;; Geef een exemplaar terug zonder drie elementen.
(nthcdr 3 '(den spar eik esdoorn))
     ⇒ (esdoorn)

;; Geef een exemplaar terug waar alle vier elementen ontbreken.
(nthcdr 4 '(den spar eik esdoorn))
     ⇒ nil

;; Geef een exemplaar terug zonder alle elementen.
(nthcdr 5 '(den spar eik esdoorn))
     ⇒ nil
```

7.4 nth

De functie `nthcdr` neemt herhaaldelijk de CDR van een lijst. De functie `nth` neemt de CAR van het resultaat dat door `nthcdr` wordt teruggegeven. Het geeft het N-de element van de lijst terug.

Dus, als het, in verband met de snelheid, niet in C zou zijn gedefinieerd zou dit de definitie van `nth` zijn:

```
(defun nth (n list)
  "Returns the Nth element of LIST.
N counts from zero.  If LIST is not that long, nil is returned."
  (car (nthcdr n list)))
```

(Oorspronkelijk was `nth` gedefinieerd in Emacs Lisp in `subr.el`, maar de definitie was in de jaren tachtig opnieuw gedaan in C.)

De functie `nth` geeft een enkel element van een lijst terug. Dit kan erg handig zijn.

Merk op de de elementen vanaf nul zijn genummerd, niet een. Dat wil zeggen dat het eerste element van een lijst, zijn CAR, is het nulde element. Deze nul-gebaseerde telling is lastig voor mensen die gewend zijn die gewend zijn dat het eerste element in een lijst nummer een is, wat een-gebaseerd is.

Bijvoorbeeld:

```
(nth 0 '("een" "twee" "drie"))
    ⇒ "een"

(nth 1 '("een" "twee" "drie"))
    ⇒ "twee"
```

Het is waardevol op te merken dat `nth`, net als `nthcdr` en `cdr`, de oorspronkelijke lijst niet verandert—de functie is niet-destructief. Dit is in scherp contrast met de functies `setcar` en `setcdr`.

7.5 setcar

Zoals je wellicht door hun namen vermoedt, stellen de functies `setcar` en `setcdr` de CAR of de CDR van een lijst in op een nieuwe waarde. Zij wijzigen echt de oorspronkelijke lijst, in tegenstelling tot `car` en `cdr` die de oorspronkelijke lijst laten zoals die was. Een manier om te ontdekken hoe dit werkt is door te proberen. Wij beginnen met de functie `setcar`.

Eerst maken we een lijst en stellen die met de speciale vorm `setq` als de waarde in van een variabele. Omdat we van plan zijn `setcar` te gebruiken om de lijst te wijzigen, moet deze `setq` geen gebruik van de gequote vorm `'(antiloop giraf leeuw tijger)` maken, omdat dit een lijst oplevert die onderdeel van het programma is en slechte dingen zouden kunnen gebeuren wanneer we proberen een deel van het programma te wijzigen terwijl dat loopt. In het algemeen moeten de componenten van een Emacs Lisp programma constant zijn (of ongewijzigd) terwijl het programma loopt. In plaats daarvan construeren we daarom als volgt een lijst met de functie `list`:

```
(setq dieren (list 'antiloop 'giraf 'leeuw 'tijger))
```

Wanneer je dit in Info binnen GNU Emacs leest kan je deze expressie op de gebruikelijke manier evalueren, door de cursor achter de expressie te plaatsen en `C-x C-e` te typen. (Ik doe dat meteen hier terwijl ik dit schrijf. Dat is een van de voordelen van een ingebouwde interpreter in je computer omgeving. Overigens, wanneer is niets op de regel na het laatste haakje staat, zoals commentaar, kan point ook de volgende regel staan. Dus als point op de eerste kolom van de regel staat, hoef je die niet te verplaatsten. Emacs staat zelfs een willekeurige hoeveelheid witte ruimte achter het laatste haakje toe.)

Wanneer we de variabele `dieren` evalueren dan zien we dat die gebonden is aan de lijst `(antiloop giraf leeuw tijger)`:

```
dieren
     ⇒ (antiloop giraf leeuw tijger)
```

Anders gezegd, de variabele `dieren` wijst naar de lijst `(antiloop giraf leeuw tijger)`.

Vervolgens evalueren we de functie `setcar` terwijl we daar twee argumenten aan doorgeven, de variabele `dieren` en het geqoute symbool `nijlpaard`. Dit doen we door de lijst van drie elementen `(setcar dieren 'nijlpaard)` te schrijven en die dan op de gebruikelijke manier te evalueren.

```
(setcar dieren 'nijlpaard)
```

Na het evalueren van deze expressie, evalueer je de variabele `dieren` nogmaals. Je zult zien dat de lijst gewijzigd is:

```
dieren
     ⇒ (nijlpaard giraf leeuw tijger)
```

Het eerste element van de lijst, `antiloop` is vervangen met `nijlpaard`.

We zien dus dat `setcar` geen nieuw element aan de lijst heeft toegevoegd, wat `cons` gedaan zou hebben, maar `antiloop` verving met `nijlpaard`. Het *wijzigde* de lijst.

7.6 `setcdr`

De functie `setcdr` is vergelijkbaar met de functie `setcar`, behalve dat de functie de tweede en volgende elementen van een lijst vervangt in plaats van het eerste element.

(Kijk vooruit naar "De `kill-new` functie", pagina 98, om te zien hoe het laatste element van een list te wijzigen. De `kill-new` functie gebruikt de `nthcdr` en de `setcdr` functies.)

Om de zien hoe dit werkt stel je waarde van de variabele in op een lijst van gedomesticeerde dieren door de volgende expressie te evalueren:

```
(setq gedomesticeerde-dieren (list 'paard 'koe 'schaap 'gijt))
```

Wanneer je nu de lijst evalueert, krijg je de lijst `(paard koe schaap gijt)` terug.

```
gedomesticeerde-dieren
     ⇒ (paard koe schaap gijt)
```

Evalueer vervolgens `setcdr` met twee argumenten, de naam van de variabele die een lijst als waarde heeft, en de lijst waarop we de CDR van de eerste lijst willen instellen:

```
(setcdr gedomesticeerde-dieren '(kat hond))
```

Wanneer je deze expressie evalueert verschijnt de lijst `(kat hond)` in het echogebied. Dit is de waarde die de functie teruggaf. Het resultaat waar wij in geïnteresseerd zijn is in het zij-effect, wat we kunnen zien wanneer we de variabele `gedomesticeerde-dieren` evalueren:

```
gedomesticeerde-dieren
     ⇒ (paard kat hond)
```

Inderdaad is de lijst gewijzigd van `(paard koe schaap gijt)` naar `(paard kat hond)`. De CDR van de lijst is gewijzigd van `(koe schaap gijt)` naar `(kat hond)`.

7.7 Oefening

Construeer een lijst met vier vogels door verschillende expressies met `cons` te evalueren. Ontdek wat gebeurt wanneer je een lijst met zichzelf `const`-t. Vervang het eerste element van de lijst met vier vogels met een vis. Vervang de rest van de lijst met een lijst van andere vissen.

8 Tekst knippen en opslaan

Steeds als je in GNU Emacs tekst uit een buffer snijdt of clipt met een *kill* commando, wordt het bewaard in een lijst en kan je het terughalen met een *yank* commando.

(Het gebruik van het woord "kill" in Emacs voor processen die *niet* specifiek de waarden van de entiteiten vernietigen, is een ongelukkig historisch ongeluk. Een veel beter word zou "clip" zijn, omdat dat is wat kill commando's doen. Zij clippen tekst uit het buffer en bergen dat in een opslagplaats op van waar het teruggebracht kan worden. Ik ben vaak in de verleiding gekomen om overal waar "kill" voorkomt in de Emacs broncode dat met "clip" te vervangen en overal waar "killed" voorkomt met "clipped".)

Wanneer tekst uit een buffer wordt gesneden, wordt het in een lijst opgeslagen. Opvolgende stukken tekst worden in de lijst opeenvolgend opgeslagen, zodat de lijst er zo uit kan zien:

```
("een stuk tekst" "vorig stuk")
```

De functie `cons` kan gebruikt worden om een nieuwe lijst te maken van een stuk tekst (een "atoom", om het jargon te gebruiken) en een bestaande lijst, zoals hier:

```
(cons "nog een stuk"
      '("een stuk tekst" "vorig stuk"))
```

Wanneer je deze expressie evalueert, verschijnt een lijst met drie elementen in het echogebied:

```
("nog een stuk" "een stuk tekst" "vorig stuk")
```

Met de functies `car` en `nthcdr` haal je een willekeurig stuk tekst naar keuze op. Bijvoorbeeld, in de volgende code geeft `nthcdr 1 ...` de lijst met eerste element verwijderd terug, en de `car` geeft het eerst element van de rest terug—het tweede element van de oorspronkelijke lijst:

```
(car (nthcdr 1 '("nog een stuk"
                 "een stuk tekst"
                 "vorig stuk")))
      ⇒ "een stuk tekst"
```

De echte functies in Emacs zijn natuurlijk meer complex dan dit. De code voor het snijden en terughalen van tekst moet zo zijn geschreven dat Emacs kan uitzoeken welk element in de lijst je wilt—eerste, tweede, derde, of welke dan ook. Verder, wanneer je aan het einde van lijst komt, moet Emacs je het eerste element van de lijst geven, in plaats van helemaal niets.

De lijst die de stukken tekst bevat heet de *killring*. Dit hoofdstuk werkt naar een beschijving van de killring toe en hoe die gebruikt wordt door eerst te volgen hoe de functie `zap-to-char` werkt. Deze functie roept een functie aan die een functie start die de killring manipuleert. Dus, voordat we de bergen bereiken, beklimmen we eerst de voetheuvels.

Een volgend hoofdstuk beschrijft hoe tekst dat uit het buffer is gesneden wordt teruggehaald. Zie Hoofdstuk 10 "Tekst terug yanken", pagina 113.

8.1 `zap-to-char`

Laten we naar de interactieve functie `zap-to-char` kijken.

De functie zap-to-char verwijdert de tekst in de region tussen de lokatie van de cursor (dus van point) tot en met waar het gespecificeerde karakter verschijnt. De tekst die zap-to-char verwijdert wordt in de killring gestopt en het kan uit de killring worden opgehaald door C-y (yank) te typen.

Wanneer het gespecificeerde karakter niet gevonden is, zegt zap-to-char "Search failed", en welk karakter je typte, en verwijdert geen tekst.

Om vast te stellen hoeveel tekst te verwijderen, gebruikt zap-to-char een zoekfunctie. Zoeken wordt extensief gebruikt in code die tekst manipuleert, en wij zullen aandacht aan ze besteden, net als aan de delete commando's.

Hier is de complete tekst van de vesie 22 implementatie van de functie:

```
(defun zap-to-char (arg char)
  "Kill up to and including ARG'th occurrence of CHAR.
Case is ignored if `case-fold-search' is non-nil in the current buffer.
Goes backward if ARG is negative; error if CHAR not found."
  (interactive "p\ncZap to char: ")
  (if (char-table-p translation-table-for-input)
      (setq char (or (aref translation-table-for-input char) char)))
  (kill-region (point) (progn
                         (search-forward (char-to-string char)
                                         nil nil arg)
                         (point))))
```

De documentatie is grondig. Je moet weten wat in het jargon het woord "kill" betekent.

The version 22 documentation string for zap-to-char uses ASCII grave accent and apostrophe to quote a symbol, so it appears as ‘case-fold-search’. This quoting style was inspired by 1970s-era displays in which grave accent and apostrophe were often mirror images suitable for use as quotes. On most modern displays this is no longer true, and when these two ASCII characters appear in documentation strings or diagnostic message formats, Emacs typically transliterates them to *curved quotes* (left and right single quotation marks), so that the abovequoted symbol appears as ‘case-fold-search’. Source-code strings can also simply use curved quotes directly.

8.1.1 De interactive expressie

De interactive expressie in het `zap-to-char` commando ziet er zo uit:

```
(interactive "p\ncZap to char: ")
```

Het gedeelte tussen de aanhalingstekens, `"p\ncZap to char: "`, specificeert twee verschillende dingen. Het eerste en meest eenvoudige is de '`p`'. Dit deel is met een nieuwe regelteken '`\n`' gescheiden van het volgende deel. Het '`p`' betekent dat het eerste argument van de functie de waarde van een *processed prefix* doorgeeft. Het prefix-argument wordt doorgegeven door *`C-u`* en een getal te typen, of *`M-`* en een cijfer. Wanneer een functie interactief wordt aangeroepen zonder een prefix, dan wordt 1 als argument doorgegeven.

Het tweede deel van `"p\ncZap to char: "` is '`cZap to char: `'. In dit deel geeft de kleine letter '`c`' aan dat `interactive` een prompt verwacht en dat het argument een karakter is. De prompt volgt op de '`c`' en is de string '`Zap to char: `' (met een spatie achter de dubbele punt zodat het er goed uitziet.)

Dit alles bereidt de argumenten van `zap-to-char` voor, zodat die van het juiste type zijn en de gebruiker een prompt geven.

In een read-only buffer kopieert de functie `zap-to-char` de tekst naar de killring, maar verwijdert die niet. Het echogebied toont een boodschap dat het buffer read-only is. Ook kan de terminal piepen of knipperen.

8.1.2 De body van `zap-to-char`

De body van de functie `zap-to-char` bevat code die de tekst in de region van de huidige cursorpositie tot en met het gespecificeerde karakter killt (dat wil zeggen, verwijdert).

Het eerste deel van de code ziet er zo uit:

```
(if (char-table-p translation-table-for-input)
    (setq char (or (aref translation-table-for-input char) char)))
(kill-region (point) (progn
                       (search-forward (char-to-string char) nil nil arg)
                       (point)))
```

`char-table-p` functie zien we voor het eerst. Het stelt vast of het argument een karakter-tabel is. Wanneer dat het geval is, dan stelt die het karakter dat aan `zap-to-char` wordt doorgegeven aan een van hen in. (Dit is belangrijk voor bepaalde karakters in niet-Europese talen. De functie `aref` extraheert een element uit een array. Het is een array-specifieke functie die we in dit document niet bespreken. Zie Sectie "Arrays" in *The GNU Emacs Lisp Reference Manual*.)
`(point)` is de huidige cursorpositie.

Het volgende deel van de code is een expressie met `progn`. De body van de `progn` bestaat uit aanroepen van `search-forward` en `point`.

Het is eenvoudiger te begrijpen hoe `progn` werkt nadat we `search-forward` bestudeerd hebben, dus we kijken eerst naar `search-forward` en dan naar `progn`.

8.1.3 De functie `search-forward`

De functie `search-forward` wordt door `zap-to-char` gebruikt om het zapped-for-karakter te vinden. Wanneer het zoeken succesvol is, zet `search-forward` point onmiddellijk achter het laatste karakter in de doelstring. (In `zap-to-char` is de

doelstring slechts één karakter lang. `zap-to-char` gebruikt de functie `char-to-string` om zeker te stellen dat de computer het karakter als string behandelt.) Wanneer het zoeken achterwaarts is dan zet `search-forward` point onmiddellijk voor het eerste karakter in het doel. Ook geeft `search-forward` `t` terug, voor waar. (Het verplaatsen van point is daarom een zij-effect.)

In `zap-to-char` ziet de functie `search-forward` er zo uit:

```
(search-forward (char-to-string char) nil nil arg)
```

De functie `search-forward` heeft vier argumenten:

1. Het eerste argument is het doel, waar naar gezocht wordt. Dit moet een string zijn, zoals '`"z"`'.

   Het argument dat aan `zap-to-char` wordt doorgegeven is een enkel karakter. Door de manier waarop computers zijn gebouwd kan de Lisp interpreter een enkel karakter anders behandelen dan een string met karakters. Binnen de computer heeft een enkel karakter een ander elektronisch formaat dan een string van één karakter. (Een enkel karakter kan vaak worden opgeslagen als precies één byte, maar een string kan langer zijn en de computer moet daarop voorbereid zijn.) Omdat de functie `search-forward` naar een string zoekt, moet het karakter dat de functie `zap-to-char` ontvangt in de computer worden geconverteerd van het ene formaat in het andere, anders faalt de functie `search-forward`. Deze conversie wordt met de functie `char-to-string` gemaakt.
2. Het tweede argument begrenst het zoeken, het is gespecificeerd als een positie in het buffer. In dit geval kan het zoeken tot het einde van het buffer doorgaan, daarom is geen grens bepaald en is het tweede argument `nil`.
3. Het derde argument vertelt de functie wat het moet doen indien het zoeken faailt—het kan een fout signaleren (en een boodschap tonen) of het kan `nil` teruggeven. Een `nil` als derde argument zorgt dat de functie een fout signaleert wanneer het zoeken faalt.
4. Het vierde argument van `search-forward` is het aantal te herhalen—naar hoe veel herhalingen van de string te zoeken. Dit argument is optioneel en wanneer de functie is aangeroepen zonder dit aantal, wordt een argument met de waarde 1 doorgegeven. Wanneer dit argument negatief is, vindt het zoeken achterwaarts plaats.

In slaboonvorm ziet een `search-forward` expressie er zo uit:

```
(search-forward "doel-string"
                zoeklimiet
                wat-te-doen-als-zoeken-faalt
                aantal-herhalingen)
```

We kijken nu naar `progn`.

8.1.4 De speciale vorm `progn`

De speciale vorm `progn` laat elk van zijn argumenten achtereenvolgens evalueren en geeft daarna de waarde van de laatste terug. De voorgaande expressies worden uitsluitend geëvalueerd voor de zij-effecten die zij uitvoeren. De waarden die zij produceren worden weggegooid.

Het sjabloon voor een `progn` expressie is erg eenvoudig:

```
(progn
  body...)
```

In `zap-to-char` moet de `progn` expressie twee dingen doen: zet point exact op de juiste positie en geef de lokatie van point zodat `kill-region` weet hoe ver het moet killen.

Het eerste argument van de `progn` is `search-forward`. Wanneer `search-forward` de string vindt, zet het point onmiddellijk achter het laatste karakter in de doelstring. (In dit geval is de doelstring slechts één karakter lang.) Wanneer het achterwaarts zoekt zet `search-forward` point onmiddellijk voor het eerste karakter van het doel. Het verplaatsen van point is een zij-effect.

Het tweede en laatste argument van `progn` is de expressie `(point)`. Deze expressie geeft de waarde van point, wat in dit geval de lokatie is waar het naar verplaatst is door `search-forward`. (In de broncode is een regel die de functie vertelt om naar het voorgaande karakter te gaan wanneer het voorwaarts gaat, die in 1999 uitgecomment is. Ik kan me niet herinneren of die eigenschap of onjuiste eigenschap ooit een deel van de gedistribueerde broncode was.) De waarde van `point` wordt teruggeven door de `progn` expressie en wordt doorgegeven aan `kill-region` als het tweede argument van `kill-region`.

8.1.5 `zap-to-char` samengevat

Nu dat we hebben gezien hoe `search-forward` en `progn` werken, kunnen we zien hoe de functie `zap-to-char` als geheel werkt.

Het eerste argument voor `kill-region` is de cursorpositie op het moment dat het `zap-to-char` commando werd gegeven. Binnen `progn` verplaatst de zoekfunctie vervolgens point naar onmiddellijk achter het zapped-to-character en `point` geeft de waarde van deze lokatie terug. De `kill-region` voegt deze twee waarden van point, het eerste als het begin van de region en en de tweede als einde van de region, en verwijdert de region.

De speciale vorm `progn` is noodzakelijk omdat het `kill-region` twee argumenten verwacht, en het zou falen wanneer de `search-forward` en `point` expressies als twee opeenvolgende argumenten geschreven zouden worden. De `progn` expressie is een enkel argument voor `kill-region` en geeft de ene waarde terug die `kill-region` nodig heeft als tweede argument.

8.2 `kill-region`

De functie `zap-to-char` gebruikt de `kill-region`. Deze functie clipt de tekst van een region en kopieert die tekst naar de killring, van waar het kan worden opgehaald.

De Emacs 22 versie van die functie gebruikt `condition-case` en `copy-region-as-kill` die wij beide uitleggen. `condition-case` is een belangrijke speciale vorm.

De functie `kill-region` in essentie roept `condition-case` aan, die drie argumenten verwacht. Het eerste argument deze functie doet niets. Het tweede argument bevat de code die het werk doet als alles goed gaat. Het derde argument bevat de code die aangeroepen wordt in het geval van een fout.

We gaan zo meteen door de code van `condition-case`. Laten we eerst naar de definitie van `kill-region` kijken, met toegevoegd commentaar:

```
(defun kill-region (beg end)
  "Kill (\"cut\") text between point and mark.
This deletes the text from the buffer and saves it in the kill ring.
The command \\[yank] can retrieve it from there. ... "

  ;; • Omdat volgorde belangrijk is, geeft eerst point door.
  (interactive (list (point) (mark)))
  ;; • En vertel ons wanneer we geen tekst kunnen snijden.
  ;; 'unless' is een 'if' zonder een dan-deel.
  (unless (and beg end)
    (error "The mark is not set now, so there is no region"))

  ;; • 'condition-case' verwacht drie argumenten.
  ;;    Wanneer het eerste argument nil is, zoals hier,
  ;;    wordt information over het fout signaal niet
  ;;    bewaard voor gebruik in een andere functie.
  (condition-case nil

      ;; • Het tweede argument van 'condition-case' vertelt de
      ;;    Lisp interpreter wat te doen als alles goed gaat.

      ;;    Het start met een 'let' functie die de string extraheert
      ;;    en test of die bestaat. Zo ja (dat is wat de
      ;;    'when' checkt), roept het een 'if' functie aan die vaststelt
      ;;    of het vorige commando een andere aanroep was van
      ;;    'kill-region'; zo ja, wordt de nieuwe tekst achter de
      ;;    vorige tekst geplakt; zo niet dan wordt een andere functie,
      ;;    'kill-new', aangeroepen.

      ;;    De 'kill-append' functie plakken de nieuwe string en
      ;;    de oude aan elkaar. De 'kill-new' functie voegt tekst in als een
      ;;    nieuw item in de killring.

      ;;    'when' is een 'if' zonder een dan-deel.  De tweede 'when'
      ;;    controleert opnieuw of de bestaande string bestaat;
      ;;    daarnaast, controleert het of het vorige commando een
      ;;    andere aanroep van 'kill-region' was.  Wanneer een van beide
      ;;    condities waar zijn, dan stelt het het huidige commando in
      ;;    op 'kill-region'.
      (let ((string (filter-buffer-substring beg end t)))
        (when string                    ;STRING is nil if BEG = END
          ;; Voeg die string op of andere manier aan de killring toe.
          (if (eq last-command 'kill-region)
```

```
            ;;    - 'yank-handler' is een optional argument van
            ;;    'kill-region' dat aan de functies 'kill-append' en
            ;;    'kill-new' hoe om te gaan met eigenschappen toegevoegd
            ;;    aan de tekst, zoals 'bold' of 'italics'.
            (kill-append string (< end beg) yank-handler)
          (kill-new string nil yank-handler)))
      (when (or string (eq last-command 'kill-region))
        (setq this-command 'kill-region))
      nil)

;;  • Het derde argument van 'condition-case' vertelt de interpreter
;;    hoe om te gaan met een fout.
;;    Het derde argument heeft een condities deel en een body deel.
;;    Wanneer aan de condities is voldaan (in dit geval,
;;              wanneer tekst of buffer read-only zijn)
;;    dan wordt de body uitgevoerd.
;;    The first part of the third argument is the following:
((buffer-read-only text-read-only) ;; the if-part
 ;; ...  the then-part
 (copy-region-as-kill beg end)
 ;;    Vervolgens, dus als deel van het dan-deel, zet this-command, zodat
 ;;    het wordt gebruikt in geval van een fout
 (setq this-command 'kill-region)
 ;;    Tenslotte, in het dan-deel, stuur een boodschap wanneer je de
 ;;    tekst niet naar de the killring kan kopiëren zonder een fout te signaleren
 ;;    maar doe dat niet als je het niet mag.
 (if kill-read-only-ok
     (progn (message "Read only text copied to kill ring") nil)
   (barf-if-buffer-read-only)
   ;; Als het buffer niet read-only is, is de tekst.
   (signal 'text-read-only (list (current-buffer)))))))
```

8.2.1 `condition-case`

Wanneer de Emacs Lisp interpreter problemen heeft met het evalueren van een expressie dat geeft het je hulp, zoals we eerder gezien hebben (zie Sectie 1.3 “Een foutboodschap genereren”, pagina 4). In het jargon heet dit “een fout signaleren”. Meestal stopt de computer het programma en toont je een boodschap.

Sommige programma’s voeren echter gecompliceerde acties uit. Zij moeten bij een fout niet stoppen. In de functie `kill-region` is de meest voor de hand liggende fout dat je probeert tekst te killen die read-only is en niet kan worden verwijderd. De functie `kill-region` bevat daarom code om met deze situatie om te gaan. Deze code in de body van de functie `kill-region` bevindt zich in een speciale vorm `condition-case`.

Het sjabloon voor `condition-case` ziet er zo uit:

```
(condition-case
  var
  bodyform
  error-handler...)
```

Het tweede argument, *bodyform* is eenvoudig. De speciale vorm `condition-case` laat de Lisp interpreter de code in *bodyform* evalueren.

Kort gezegd, het *bodyform*-deel van een `condition-case` expressie bepaalt wat moet gebeuren wanneer alles correct werkt.

Echter, wanneer een fout optreedt, definieert de functie, die naast zijn andere activiteiten het foutsignaal genereert, een of meer conditienamen.

Een error-handler is het derde argument van `condition-case`. Een error-handler heeft twee delen, een *conditienaam* en een *body*. Wanneer het *conditienaam*-deel van de error-handler overeenkomt met de conditienaam gegeneerd door een fout, dan wordt het *body*-deel van de error-handler uitgevoerd.

Zoals je zou verwachten kan het *conditienaam*-deel van de error-handler een enkele conditienaam zijn of een lijst met conditienamen.

Een complete `condition-case` expressie kan dus meer dan één error-handler bevatten. Wanneer een fout optreedt, wordt de eerst toepasbare handler uitgevoerd.

Tenslotte, het eerste argument van de `condition-case` expressie, het *var* argument is soms gebonden aan een variabele die informatie over de fout bevat. Wanneer dat argument echter `nil` is, zoals dat bij `kill-region` het geval is, wordt die informatie genegeerd.

In het kort, in de functie `kill-region` werkt de code `condition-case` als volgt:

```
If geen errors, voer alleen deze code uit
    maar, if errors, voer deze andere code uit.
```

8.2.2 Lisp macro

Het deel van de `condition-case` expressie die wordt geëvalueerd in de verwachting dat alles goed gaat heeft een `when`. De code stelt met `when` of de variabele `string` naar bestaande tekst wijst.

Een `when`-expressie maakt het de programmeur makkelijker. Het is als een `if`, zonder de mogelijkheid van een anders-clausule. In je gedachten kun je `when` met `if` vervangen om te begrijpen wat er gebeurt. Dat is wat de Lisp interpreter doet.

Technisch gesproken is `when` een Lisp macro. Een Lisp macro stelt je in staat nieuwe control-constructies en andere taaleigenschappen te definieren. Het vertelt de interpreter hoe het een andere Lisp expressie te berekenen die op zijn beurt de waarde berekend. In dit geval is de andere expressie een `if` expressie.

De `kill-region` functiedefinitie heeft ook een `unless` macro. Dit is het tegenovergestelde van `when`. De `unless` macro is net als `if`, behalve dat het geen dan-clausule heeft, en het verstrekt daarvoor een impliciete `nil`.

Voor meer over Lisp macro's, zie Sectie "Macros" in *The GNU Emacs Lisp Reference Manual*. De programmeertaal C vertrekt ook macro's. Deze zijn anders, maar ook nuttig.

Ten aanzien van de `when` macro, wanneer de string in de `condition-case` expressie een inhoud heeft, dat wordt een andere conditionele expressie uitgevoerd. Dit is een `if` met zowel een dan-deel en een anders-deel.

```
(if (eq last-command 'kill-region)
    (kill-append string (< end beg) yank-handler)
  (kill-new string nil yank-handler))
```

Het dan-deel wordt geëvalueerd wanneer het voorgaande commando een andere aanroep van `kill-region` was. Zo niet, dan wordt het anders-deel geëvalueerd.

`yank-handler` is een optioneel argument voor `kill-region` dat aan de functies `kill-append` en `kill-new` vertelt hoe het met de eigenschappen moet omgaan die aan de tekst zijn gekoppeld, zoals bold of italics.

`last-command` is een variabele in Emacs die we nog niet eerder gezien hebben. Steeds wanneer een functie wordt uitgevoerd, stelt Emacs Normaal gesproken de waarde van `last-command` in op het voorgaande commando.

In dit segment van de definitie controleert de `if` expressie of het voorgaande commando `kill-region` was. Zo ja, plakt

```
(kill-append string (< end beg) yank-handler)
```

een kopie van de nieuw geclipte tekst aan de voorgaande zojuist geclipte tekst in de killring.

8.3 copy-region-as-kill

De functie `copy-region-as-kill` kopieert een region met tekst van een buffer en (via hetzij `kill-append` of `kill-new`) bewaart het in de `kill-ring`.

Wanneer je `copy-region-as-kill` meteen na een `kill-region` commando aanroept, plakt Emacs de nieuw gekopieerde tekst aan de voorgaande gekopieerde tekst. Dit betekent dat als je de tekst terug-yankt, je alles krijgt, van zowel deze als de voorgaande operatie. Anderzijds, wanneer een ander commando aan de `copy-region-as-kill` voorafging, dan kopieert de functie de tekst in een separaat item in de killring.

Hier is de complete tekst van de versie 22 `copy-region-as-kill` functie.

```
(defun copy-region-as-kill (beg end)
  "Save the region as if killed, but don't kill it.
In Transient Mark mode, deactivate the mark.
If `interprogram-cut-function' is non-nil, also save the text for a window
system cut and paste."
  (interactive "r")
  (if (eq last-command 'kill-region)
      (kill-append (filter-buffer-substring beg end) (< end beg))
    (kill-new (filter-buffer-substring beg end)))
  (if transient-mark-mode
      (setq deactivate-mark t))
  nil)
```

Zoals gebruikelijk kan deze functie worden gedeeld in componenten:

```
(defun copy-region-as-kill (argument-list)
  "documentatie..."
  (interactive "r")
  body...)
```

De argumenten zijn `beg` en `end` en de functie is interactief met `"r"`, dus de twee argumenten moeten refereren aan het begin en eind van de region. Wanneer je dit document van het begin hebt gelezen, dan is het begrijpen van deze delen van een functie al bijna routine.

De documentatie is ietwat verwarrend tenzij je herinnert dat het woord "kill" een andere betekenis dan gebruikelijk heeft. De Transient Mark en `interprogram-cut-function` commentaren verhelderen bepaalde zij-effecten.

Nadat je ooit een mark hebt gezet, bevat een buffer altijd een region. Wanneer je dat wilt, kan je met Transient Mark de region tijdelijk highlighten. (Niemand wil de region voortdurend gehighlight hebben, dus Transient Mark mode highlight het alleen op passende momenten. Veel mensen zetten Transient Mark mode uit, waardoor de region nooit wordt gehighlight.)

Ook kun je in windowing systemen tussen verschillende programma's kopiëren, snijden en plakken. In het X windowing system bijvoorbeeld is de functie `interprogram-cut-function x-select-text`, die met het equivalent voor de Emacs killring van het windowing systeem werkt.

The body of the `copy-region-as-kill` function starts with an `if` clause. What this clause does is distinguish between two different situations: whether or not this command is executed immediately after a previous `kill-region` command. In the first case, the new region is appended to the previously copied text. Otherwise, it is inserted into the beginning of the kill ring as a separate piece of text from the previous piece.

De laatste twee regels van de functie voorkomen highighting van de region wanneer Transient Mark mode aan staat.

De body van `copy-region-as-kill` is bespreking in detail waard.

8.3.1 De body van `copy-region-as-kill`

De functie `copy-region-as-kill` werkt op een vergelijkbare manier als de functie `kill-region`. Beide zijn zo geschreven dat twee of meer kills op een rij de tekst in een enkele entry combineren. Wanneer je de tekst van de killring terugyankt, dan krijg alles in een stuk. Bovendien, kills die voorwaarts vanaf de huidige cursorpositie bewegen worden toegevoegd aan het einde van de vorige gekopieerde tekst, en commando's die tekst achterwaarts kopiëren worden aan het begin van de vorige gekopieerde tekst toegevoegd. Op deze manier blijven de woorden in tekst op de juiste volgorde.

Net als `kill-region`, maakt de functie `copy-region-as-kill` gebruik van de variabele `last-command` die het vorige Emacs commando bijhoudt.

Wanneer een functie wordt uitgevoerd, stelt Emacs normaalgesproken de waarde van `this-command` in op de functie die wordt uitgevoerd (wat in dit geval `copy-`

`region-as-kill` zou zijn). Tegelijkertijd stelt Emacs de waarde van `last-command` in op de vorige waarde van `this-command`.

In het eerste deel van de body van de functie `copy-region-as-kill` stelt een `if` expressie vast of `kill-region` de waarde van `last-command` is. Zo ja, dan wordt het dan-deel van de `if` expressie geëvalueerd. Het gebruikt de functie `kill-append` om de tekst die bij deze aanroep gekopieerd is te plakken aan de tekst die al in het eerste element (the CAR) van de killring staat. Anders, wanneer de waarde van de `last-command` niet `kill-region` is, dan voegt de functie `copy-region-as-kill` met functie `kill-new` een nieuw element aan de killring toe.

De `if` expressie is als volgt, met `eq`:

```
(if (eq last-command 'kill-region)
    ;; dan-deel
    (kill-append  (filter-buffer-substring beg end) (< end beg))
  ;; anders-deel
  (kill-new  (filter-buffer-substring beg end)))
```

(De functie `filter-buffer-substring` geeft een gefilterde substring van het buffer, wanneer die er is. Optioneel—de argumenten zijn hier niet, dus geen enkele wordt uitgevoerd—kan de functie de initiële tekst deleten of de tekst zonder eigenschappen teruggeven. Deze functie is een vervanging voor de oudere functie `buffer-substring` die al bestond voordat tekst-eigenschappen werden geïmplementeerd.)
De functie `eq` test of het eerste argument hetzelfde Lisp object is als het tweede argument. De functie `eq` test net als de functie `equal` voor gelijkheid, met het verschil dat deze vaststelt of twee representaties daadwerkelijk hetzelfde object in de computer zijn, maar met verschillende namen. `equal` stelt vast of de structuur en inhoud van twee expressies hetzelfde zijn.

Wanneer `kill-region` het voorgaande commando was, dan roept de Emacs Lisp interpreter de functie `kill-append` aan.

De functie `kill-append`

De functie `kill-append` ziet er zo uit:

```
(defun kill-append (string before-p &optional yank-handler)
  "Append STRING to the end of the latest kill in the kill ring.
If BEFORE-P is non-nil, prepend STRING to the kill.
... "
  (let* ((cur (car kill-ring)))
    (kill-new (if before-p (concat string cur) (concat cur string))
              (or (= (length cur) 0)
                  (equal yank-handler
                         (get-text-property 0 'yank-handler cur)))
              yank-handler)))
```

De functie `kill-append` is redelijk eenvoudig. Het gebruikt de functie `kill-new`, zie we straks in meer detail bespreken.

(De functie verschaft ook een optioneel argument die `yank-handler` heet. wanneer aangeroepen vertelt dit argument de functie hoe het met eigenschappen van de tekst moet omgaan, zoals bold of italics.)

Het heeft een functie `let*` om de waarde van het eerste element van de killring in te stellen op `cur`. (Ik weet niet waarom de functie in plaats hiervan geen `let`

gebruikt, slechts één waarde wordt in de expressie ingesteld. Misschien dat dit een bug is die geen problemen veroorzaakt?)

Bekijk de conditional die een van de twee argumenten van `kill-new` is. Het gebruikt `concat` om de nieuwe tekst aan de CAR van de kill ring te plakken. Of het vooraan of achteraan de tekst plakt hangt af van het resultaat van een `if` expressie:

```
(if before-p                    ; if-deel
    (concat string cur)         ; dan-deel
  (concat cur string))          ; anders-deel
```

Wanneer de region die gekilled wordt vooraf gaat aan de region die in het vorige commando was gekilled, dan moet het vooraan het materiaal dat in de vorige kill was bewaard komen en, omgekeerd, wanneer de gekillde tekst volgt op wat zojuist was gekilled, dan moet het achter die voorgaande tekst komen. De `if` expressie is afhankelijk van het predikaat `before-p` om te bepalen of het de nieuwe bewaarde tekst voor of achter de voorgaande bewaarde tekst moet plaatsen.

Het symbool `before-p` is de naam van een van de argumenten van `kill-append`. Het evalueren van de functie `kill-append` bindt die aan de waarde teruggegeven door de evaluatie van het argument. In dit geval de expressie `(< end beg)`. Deze expressie stelt niet direct vast of de gekillde tekst voor of na de kill-tekst van het vorige commando staat. Het stelt vast of de waarde van de variabele `end` kleiner is dan de waarde van de variabele `beg`. Wanneer dat zo is, betekent dit dat de gebruiker hoogstwaarschijnlijk richting het begin van het buffer beweegt. Het resultaat van het evalueren van de predikaat-expressie, `(< end beg)`, is daardoor waar, en de tekst wordt voor de voorgaande tekst geplakt. Wanneer anderzijds de waarde van de variabele `end` groter is dan de waarde van de variabele `beg`, wordt de tekst aan het eind van de voorgaande tekst geplakt.

Wanneer de nieuw opgeslagen tekst aan de voorzijde komt, wordt de string met de nieuwe tekst voor de oude tekst geplakt.

```
(concat string cur)
```

Maar als de tekst aan de achterzijde komt, wordt de string achter de oude tekst geplakt.

```
(concat cur string))
```

Om te begrijpen hoe dit werkt, moeten we eerst de functie `concat` bekijken. De functie `concat` knoopt twee strings aan elkaar of verenigt die. Het resultaat is een string. Bijvoorbeeld:

```
(concat "abc" "def")
     ⇒ "abcdef"

(concat "nieuw "
        (car '("eerste element" "tweede element")))
     ⇒ "nieuw eerste element"

(concat (car
        '("eerste element" "tweede element")) " aangepast")
     ⇒ "eerste element aangepast"
```

We begrijpen nu `kill-append`: het past de inhoud van de killring aan. De killring is een lijst, elk element daarvan is opgeslagen tekst. De functie `kill-append` gebruikt de functie `kill-new` die op zijn beurt de functie `setcar` gebruikt.

De functie `kill-new`

In versie 22 ziet de functie `kill-new` er zo uit:

```
(defun kill-new (string &optional replace yank-handler)
  "Make STRING the latest kill in the kill ring.
Set `kill-ring-yank-pointer' to point to it.
If `interprogram-cut-function' is non-nil, apply it to STRING.
Optional second argument REPLACE non-nil means that STRING will replace
the front of the kill ring, rather than being added to the list.
..."
  (if (> (length string) 0)
      (if yank-handler
          (put-text-property 0 (length string)
                             'yank-handler yank-handler string))
    (if yank-handler
        (signal 'args-out-of-range
                (list string "yank-handler specified for empty string"))))
  (if (fboundp 'menu-bar-update-yank-menu)
      (menu-bar-update-yank-menu string (and replace (car kill-ring))))
  (if (and replace kill-ring)
      (setcar kill-ring string)
    (push string kill-ring)
    (if (> (length kill-ring) kill-ring-max)
        (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
  (setq kill-ring-yank-pointer kill-ring)
  (if interprogram-cut-function
      (funcall interprogram-cut-function string (not replace))))
```

(Merk op dat de functie niet interactief is.)

Zoals gebruikelijk kijken we naar de functie in delen.

De functiedefinitie heeft een optioneel argument `yank-handler`, dat wanneer het aangeroepen wordt vertelt hoe het met de eigenschappen van de tekst moet omgaan, zoals bold of italics. Dit slaan we over.

De eerste regel van de documentatie is logisch:

```
Make STRING the latest kill in the kill ring.
```

Laten voor dit moment de rest van de documentatie overslaan.

Laten we ook de initiële `if` expressie overslaan en die regels code met betrekking tot `menu-bar-update-yank-menu`. Die leggen we hieronder uit.

De kritische regels zijn deze:

```
(if (and replace kill-ring)
    ;; dan
    (setcar kill-ring string)
  ;; anders
  (push string kill-ring)
  (if (> (length kill-ring) kill-ring-max)
      ;; voorkom een te lange kill ring
      (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
(setq kill-ring-yank-pointer kill-ring)
(if interprogram-cut-function
    (funcall interprogram-cut-function string (not replace))))
```

De conditionele test is `(and replace kill-ring)`. Deze is waar wanneer twee condities voldoen: de killring bevast iets en de variabele `replace` variable is waar.

Wanneer de functie `kill-append` `kill-append` op waar instelt en wanneer de killring tenminste een element bevat, wordt de `setcar` expressie uitgevoerd:

```
(setcar kill-ring string)
```

De functie `setcar` wijzigt daadwerkelijk het eerste element van de `kill-ring` lijst in de waarde van `string`. Het vervangt het eerste element.

Aan de andere kant, wanneer de killring leeg, of replace onwaar is, wordt het anders-deel van de conditie uitgevoerd:

```
(push string kill-ring)
```

`push` plaatst het eerste argument op het tweede. Het is vergelijkbaar met het oudere

```
(setq kill-ring (cons string kill-ring))
```

of het nieuwere

```
(add-to-list kill-ring string)
```

Wanneer het onwaar is, construeert de expressie eerst een nieuwe versie van de killring door `string` vooraan als nieuw element aan de bestaande killring te plaatsen (dat is wat de `push` doet.) Daarna voert het een tweede `if`-clausule uit. Deze tweede `if`-clausule voorkomt dat de killring te groot groeit.

Laten we in volgorde naar deze twee expressies kijken.

De `push` regel in het anders-deel stelt de nieuwe waarde van de kill ring in op wat het resultaat is van het aan de oude killring toevoegen van de gekillde string.

We zijn hoe dit werkt aan de hand van een voorbeeld.

Eerst

```
(setq voorbeeldlijst '("hier is een clausule" "andere clausule"))
```

Na het evalueren van deze expressie met *C-x C-e*, evalueer je `voorbeeldlijst` en kijk naar wat die teruggeeft.

```
voorbeeldlijst
     ⇒ ("hier is een clausule" "andere clausule")
```

We voegen nu een nieuw element toe aan deze lijst door de volgende expressie te evalueren:

```
(push "een derde clausule" voorbeeldlijst)
```

Wanneer we `voorbeeldlijst` evalueren, ontdekken we dat zijn waarde is:

```
voorbeeldlijst
     ⇒ ("een derde clausule" "hier is een clausule" "andere clausule")
```

De derde clausule is dus door `push` toegevoegd aan de lijst.

Nu het tweede deel van de `if` clausule. Deze expressie voorkomt dat de killring te groot groeit. Het ziet er zo uit:

```
(if (> (length kill-ring) kill-ring-max)
    (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil))
```

De code controleert of de lengte va de killring groter is dan de maximum toegestane lengte. Dit is de waarde van `kill-ring-max` (die standaard 120 is). Wanneer de lengte van de kill ring te groot is, dan stelt deze code het laatste element van de killring in op `nil`. Het doet dit met twee functies, `nthcdr` and `setcdr`.

We keken eerder naar `setcdr`, (zie Sectie 7.6 “`setcdr`”, pagina 85). Het zet de CDR van een lijst, net als `setcar` de CAR van de lijst zet. In dit geval echter zet `setcdr` niet de de CDR van de gehele killring, de functie `nthcdr` zorgt dat het de CDR van het een-na-laatste element van de killring zet—dit betekent dat omdat de CDR van het een-na-laatste element het laatste element van de killring is, het het laatste element van de killring zal zetten.

De functie `nthcdr` werkt door herhaaldelijk de CDR van een lijst te nemen—het neemt de CDR van de CDR van de CDR ... Het doet dit N keer en geeft dan het resultaat terug. (Zie Sectie 7.3 “`nthcdr`”, pagina 81.)

Dus als we een lijst met vier elementen hebben die drie elementen lang behoort te zijn, kunnen we de CDR van het een-na-laatste element op `nil` zetten en daardoor de lijst korter maken. (Wanneer je het laatste element op iets anders dan `nil` zet, wat je kunt doen, dan zou je lijst niet korter gemaakt hebben. Zie Sectie 7.6 “`setcdr`”, pagina 85.)

Je kan dit inkorten zien door de volgende drie expressies beurtelings te evalueren. Eerst zet je de waarde van `bomen` op `esdoorn eik spar berk`, daarna zet je de CDR van zijn tweede CDR op `nil` en vindt de waarde van `bomen`.

```
(setq bomen (list 'esdoorn 'eik 'spar 'berk))
      ⇒ (esdoorn eik spar berk)

(setcdr (nthcdr 2 bomen) nil)
     ⇒ nil

bomen
      ⇒ (esdoorn eik spar)
```

(De waarde teruggeven door de `setcdr` expressie is `nil` omdat dat is waarop de CDR is gezet.)

Nogmaals, in `kill-new`, neemt de functie `nthcdr` een aantal keer de CDR, één minder dan de maximaal toegestane grootte van de killring en de `setcdr` zet de CDR van dat element (wat de rest van de elementen in de killring is) op `nil`. Dit voorkomt dat de killring te groot groeit.

De een-na-laatste expressie in de kill ring is

```
(setq kill-ring-yank-pointer kill-ring)
```

De `kill-ring-yank-pointer` is een globale variabele die is ingesteld om de `kill-ring` te zijn.

Ondanks dat de `kill-ring-yank-pointer` een `pointer` heet, is het een variabele net als de killring. De naam is echter gekozen om mensen te helpen begrijpen hoe de variabele wordt gebruikt.

Nu gaan we terug naar een expressie in het begin van de body van de functie:

```
(if (fboundp 'menu-bar-update-yank-menu)
     (menu-bar-update-yank-menu string (and replace (car kill-ring))))
```

Het begint met een `if` expressie

In dit geval test de expressie eerst of de functie `menu-bar-update-yank-menu` bestaat en zo ja, roept die aan. De functie `fboundp` geeft waar terug wanneer het symbool dat het test een functiedefinitie heeft die niet leeg is. Wanneer de functiedefinitie van het symbool leeg is, krijgen we een foutmelding, zoals we die kregen toen we expres fouten maakten (zie Sectie 1.3 “Een foutboodschap genereren”, pagina 4).

Het dan-deel bevat een expressie wiens eerste argument de functie `and` is.

De speciale vorm `and` evalueert elk van zijn argumenten totdat een van de argumenten de waarde `nil` teruggeeft, in welk geval de `and` expressie `nil` teruggeeft. Wanneer echter geen van de argumenten een waarde `nil` teruggeven, dan wordt de waarde afkomstig van de evaluatie van het laatste argument teruggegeven. (Omdat zo’n waarde niet `nil` is, wordt het in Emacs Lisp beschouwd als waar.) Met andere woorden, een `and` expressie geeft alleen waar terug als al zijn argumenten waar zijn. (Zie Sectie 5.4 “Tweede buffer gerelateerde terugblik”, pagina 72.)

De expressie stelt vast of het tweede argument van `'menu-bar-update-yank-menu` waar is of niet.

`menu-bar-update-yank-menu` is een van de functies die het mogelijk maken om het “Select and Paste” menu in het Edit-item in de menubar te gebruiken. Met de muis kan je zien elke verschillende stukken tekst je hebt opgeslagen en een stuk selecteren om te pasten.

De laatste expressie in de functie `kill-new` voegt de nieuw gekopieerde string toe aan wat voor faciliteit dan ook bestaat voor het kopiëren en plakken tussen de verschillende programma’s die in het windowing systeem draaien. In het X windowing systeem bijvoorbeeld, pakt de functie `x-select-text` de string op en bewaart die in het geheugen dat X bewerkt. Je kan de string in een andere programma pasten, zoals een Xterm.

De expressie ziet er zo uit:

```
(if interprogram-cut-function
    (funcall interprogram-cut-function string (not replace))))
```

Wanneer een `interprogram-cut-function` bestaat, dan voert Emacs `funcall` uit, die op zijn beurt het eerste argument als een functie aanroept en de resterende argumenten daar aan doorgeeft. (Overigens, voor zover ik het kan zien, kan de `if`-expressie vervangen worden door een `and` expressie vergelijkbaar met die in het eerste deel van de functie.)

We bepreken verder geen windowing systems en andere programma's, maar merken slechts op dat dit een mechanisme is dat het GNU Emacs mogelijk maakt om makkelijk en goed met andere programma's samen te werken.

De code voor het plaatsen van de tekst in de killring, hetzij geplakt aan een bestaand element hetzij als een nieuw element, leidt ons naar de code om de verwijderde tekst terug te halen—het yank commando. Voordat we het yank commando gaan bespreken, is het echter beter te ontdekken hoe lijsten in de computer zijn geïmplementeerd. Dit verheldert mysteries zoals het gebruik van de term "pointer". Maar voor we dat doen, weiden we uit naar C.

8.4 Uitwijding naar C

De functie `copy-region-as-kill` function (zie Sectie 8.3 "`copy-region-as-kill`", pagina 94) gebruikt de functie `filter-buffer-substring`, die op zijn beurt de functie `delete-and-extract-region` gebruikt. Het verwijdert de inhoud van een region en die kun je niet terugkrijgen.

In tegenstelling tot andere code die we hier besproken hebben, is de functie `delete-and-extract-region` niet in Emacs Lisp geschreven, het is geschreven in C en is een van de primitieven van het GNU Emacs systeem. Omdat het erg eenvoudig is, dwaal ik even af van Lisp en beschrijf het hier.

Net als veel van de andere Emacs primitieven, is `delete-and-extract-region` geschreven als een C macro, een macro die een sjabloon voor code is. De complete macro ziet er zo uit:

```
DEFUN ("delete-and-extract-region", Fdelete_and_extract_region,
       Sdelete_and_extract_region, 2, 2, 0,
       doc: /* Delete the text between START and END and return it.  */)
  (Lisp_Object start, Lisp_Object end)
{
  validate_region (&start, &end);
  if (XFIXNUM (start) == XFIXNUM (end))
    return empty_unibyte_string;
  return del_range_1 (XFIXNUM (start), XFIXNUM (end), 1, 1);
}
```

Zonder verder op de details van het schrijven van macro's in te gaan, wil ik er op wijzen dat de macro start met het woord `DEFUN`. Het woord `DEFUN` was gekozen omdat de code hetzelfde doel dient als `defun` in Lisp. (De C macro `DEFUN` is gedefinieerd in `emacs/src/lisp.h`.)

Het woord `DEFUN` wordt gevolgd door zeven delen binnen haakjes:

- Het eerste deel is de naam die de functie in Lisp krijgt, `delete-and-extract-region`.
- Het tweede deel is de naam van de functie in C, `Fdelete_and_extract_region`. Volgens de conventie begint die met een '`F`'. Omdat C geen streepjes in namen gebruikt, zijn in plaats daarvan underscores gebruikt.
- Het derde deel is de naam van de C constant structure die informatie over de deze functie opslaat voor intern gebruik. Het is de naam van de functie in C maar begint met een '`S`' in plaats van een '`F`'.
- Het vierde en vijfde deel specificeren het minimum en maximum aantal argumenten dat de functie kan krijgen. Deze functie vereist precies 2 argumenten.
- Het zesde deel is bijna hetzelfde als het argument dat volgt op de `interactive` declaratie in een functie geschreven in Lisp: een letter, misschien gevold door een prompt. Het enige verschil met Lisp is wanneer de macro zonder argumenten wordt aangeroepen. Dan schrijf je een `0` (wat een null-string is) zoals in deze macro.

  Wanneer je argumenten specificeert, dan zet je die tussen aanhalingstekens. De C macro voor `goto-char` bevat `"NGoto char: "` op deze plek om aan te gevne dat de functie een raw prefix verwacht, in dit geval een numerieke lokatie in het buffer, en een prompt verschaft.
- Het zevende deel is een documentatiestring, net zoals voor een functie geschreven in Emacs Lisp. Dit is geschreven als C commentaar. (Wanneer je Emacs bouwt extraheert het programma `lib-src/make-docfile` deze commentaren en gebruikt die om de documentatie te maken.)

In een C macro komen de formele parameters daarna, met een statement over het soort object dat ze zijn, gevolgd door de body van de macro. Voor `delete-and-extract-region` bestaat de body uit de volgende vier regels:

```
validate_region (&start, &end);
if (XFIXNUM (start) == XFIXNUM (end))
  return empty_unibyte_string;
return del_range_1 (XFIXNUM (start), XFIXNUM (end), 1, 1);
```

De functie `validate_region` controleert of de waarden die als begin en eind zijn doorgegeven, van het juiste type en binnen de range zijn.

De functie `del_range_1` verwijdert daadwerkelijk de tekst. Het is een complexe functie waar we niet naar kijken. Het update het buffer en doet andere dingen. Echter het is het waard te kijken naar de twee argumenten die doorgeven worden aan `del_range_1`. Dit zijn `XFIXNUM (start)` en `XFIXNUM (end)`.

Wat de C-taal betreft, zijn `start` en `end` twee vage waarden die het begin en eind van de te verwijderen region markeren. Meer precies, en met meer expertise genodigd om te begrijpen, zijn de twee waarden van het type `Lisp_Object`, wat een C-pointer kan zijn, of een C-integer, of een C-`struct`. Het kan de C code normaalgesproken niet schelen hoe `Lisp_Object` is geïmplementeerd.

De breedte van `Lisp_Object` hangt af van de machine, en is typisch 32 of 64 bits. Enkele van de bits worden gebruikt om het type informatie te specificeren, de resterende bits voor de inhoud.

'`XFIXNUM`' is een C macro die de relevante integer uit de langere collectie bits extraheert. De type-bits worden genegeerd.

Het commando in `delete-and-extract-region` ziet het zo uit:

```
del_range_1 (XFIXNUM (start), XFIXNUM (end), 1, 1);
```

Het verwijdert de region tussen de beginpostitie, `start`, en de eindpositie, `end`.

Vanuit het gezichtspunt van de persoon die Lisp schrijft, is Emacs heel eenvoudig, maar eronder is een grote hoeveelheid compliciteit verborgen die het allemaal laat werken.

8.5 Initialiseer een variabele met `defvar`

De functie `copy-region-as-kill` is in Emacs Lisp geschreven. Twee functies in het, `kill-append` en `kill-new`, kopiëren een region in een buffer en bewaren het in een variabele met de naam `kill-ring`. Deze sectie beschijft hoe de `kill-ring` met de speciale vorm `defvar` is gecreëerd en geïnitialiseerd.

(Opnieuw merken we op dat de term `kill-ring` een misleidende naam is. De tekst die uit het buffer is geknipt kan teruggehaald worden. Het is niet een ring van lijken, maar een ring van herleefbare tekst.)

In Emacs Lisp wordt een variabele zoals `kill-ring` met de speciale vorm gecreëerd en van een initiële waarde voorzien. De naam komt van "define variable".

De speciale vorm `defvar` stelt de waarde van een variabele in, net als `setq`. Het verschilt van `setq` op drie manieren: ten eerste markeert het de variabele als "speciaal" zodat het altijd dynamisch gebonden is, zelfs wanneer `lexcial-binding` `t` is (zie Sectie 3.6.4 "Variabelen", pagina 36). Ten tweede, het stelt alleen de waarde van de variabele in wanneer de variabele nog geen waarde heeft. Wanneer de variabele al een waarde heeft, overschrijft `defvar` de bestaande waarde niet. Ten derde, `defvar` heeft een documentatiestring.

(Er is een gerelateerde macro, `defcustom`, gemaakt voor variabelen die mensen customizen. Het heeft meer eigenschappen dan `defvar`. (Zie Sectie 16.2 "Variabelen specificeren met `defcustom`", pagina 201.)

Je kunt de huidige waarde van een variabele, elke variabele, zien met de functie `describe-variable`, die meestal wordt aangeroepen met *`C-h v`*. Wanneer je *`C-h v`* typt, en na de prompt `kill-ring` (gevolgd door `RET`), krijg je de huidige inhoud van je killring te zien—dit kan best veel zijn! Omgekeerd, wanneer je in deze Emacs sessie niets anders gedaan hebt dan dit document lezen, kan er helemaal niets in zijn. Ook krijg je de documentatie voor `kill-ring` te zien.

```
Documentation:
List of killed text sequences.
Since the kill ring is supposed to interact nicely with cut-and-paste
facilities offered by window systems, use of this variable should
interact nicely with `interprogram-cut-function' and
`interprogram-paste-function'.  The functions `kill-new',
`kill-append', and `current-kill' are supposed to implement this
interaction; you may want to use them instead of manipulating the kill
ring directly.
```

De killring is gedefinieerd door een `defvar` op de volgende manier:

```
(defvar kill-ring nil
  "List of killed text sequences.
...")
```

In deze variabele definitie krijgt de variabele de initiële waarde `nil`, wat logisch is, omdat je nog niets hebt bewaard, wil je niets terughalen met een `yank` commando. De documentatiestring schrijf je op dezelfde manier als de documentatiestring van een `defun`. Net als bij de documentatiestring van de `defun`, moet de eerste regel van de documentatie ene volledige zin zijn omdat sommige commando's, zoals `apropos` alleen de eerste documentatieregel tonen. Opvolgende regels moeten niet ingesprongen worden, anders zien ze er raar uit als je *`C-h v`* (`describe-variable`) gebruikt.

8.5.1 `defvar` en een sterretje

In het verleden gebruikte Emacs de speciale vorm `defvar` voor zowel interne variabelen waarvan je niet verwacht dat een gebruiker die aanpast, als voor variabelen waarvan je wel verwacht dat een gebruiker die aanpast. Hoewel je nog steeds `defvar` voor door de gebruiker aan te passen variabelen kunt gebruiken, gebruik je alsjeblieft `defcustom` omdat dit een pad biedt naar de Customization commando's. (Zie Sectie 16.2 "Variabelen specificeren met `defcustom`", pagina 201.)

Wanneer je een variabele met de speciale vorm `defvar` gespecificeerd hebt, kon je onderscheid maken tussen een variabele die de gebruiker mag veranderen en de andere, door het typen van een sterretje, '`*`', in de eerste kolom van de documentatiestring. Bijvoorbeeld:

```
(defvar shell-command-default-error-buffer nil
  "*Buffer name for `shell-command' ... error output.
... ")
```

Je kon (en kan nog steeds) het commando `set-variable` gebruiken om de waarde van `shell-command-default-error-buffer` tijdelijk te wijzigen. Echter, opties gezet met `set-variable` zijn alleen voor de duur van je editing-sessie gezet. De nieuwe waarden worden tussen sessies niet bewaard. Steeds wanneer Emacs start, leest het de oorspronkelijke waarde, tenzij je de waarde in je `.emacs` bestand aanpast, door het handmatig te zetten of met gebruik van `customize`. Zie Hoofdstuk 16 "Jouw `.emacs` bestand", pagina 200.

Het belangrijkste gebruik van het commando `set-variable` voor mij is om variabelen voor te stellen die ik in mijn `.emacs` bestand zou willen zetten. Er zijn nu meer dan 700 van dat soort variabelen, veel te veel om makkelijk te herinneren. Gelukkig kan je de `TAB` toets gebruiken na het commando `M-x set-variable` om een lijst met variabelen te zien. (Zie Sectie "Examining and Setting Variables" in *The GNU Emacs Manual*.)

8.6 Terugblik

Hier is een korte samenvatting van sommige recent geïntroduceerde functies.

`car`
`cdr` `car` geeft het eerste element van een lijst terug. `cdr` geeft het tweede en volgende elementen van de lijst terug.

Bijvoorbeeld:

```
(car '(1 2 3 4 5 6 7))
     ⇒ 1
(cdr '(1 2 3 4 5 6 7))
     ⇒ (2 3 4 5 6 7)
```

`cons` `cons` construeer een lijst door het eerste element voor het tweede te zetten.

Bijvoorbeeld:

```
(cons 1 '(2 3 4))
     ⇒ (1 2 3 4)
```

`funcall` `funcall` evalueert het eerste argument als functie. Het geeft de resterende argumenten door aan het eerste argument.

`nthcdr` Geeft het resultaat van n keer CDR nemen van een lijst. De n^{th} `cdr` De "rest van de rest" als het ware.

Bijvoorbeeld:

```
(nthcdr 3 '(1 2 3 4 5 6 7))
     ⇒ (4 5 6 7)
```

`setcar`
`setcdr` `setcar` wijzigt het eerste element van een lijst, `setcdr` wijzigt het tweede en volgende elementen van een lijst.

Bijvoorbeeld:

```
(setq triple (list 1 2 3))

(setcar triple '37)

triple
     ⇒ (37 2 3)
(setcdr triple '("foo" "bar"))

triple
     ⇒ (37 "foo" "bar")
```

`progn` Evalueer achtereenvolgens elk argument en geef de waarde van de laatste terug.

Bijvoorbeeld:

```
(progn 1 2 3 4)
     ⇒ 4
```

`save-restriction`
: Bewaar de versmalling die eventueel actief is in het huidige buffer en herstel die versmalling na het evalueren van de argumenten.

`search-forward`
: Zoek voor een string, en wanneer die string gevonden is, verplaats point. Met een reguliere expressie gebruik de vergelijkbare `re-search-forward`. (Zie Hoofdstuk 12 "Reguliere expressie zoekopdrachten", pagina 143, voor een uitleg van reguliere expressies patronen en zoekopdrachten.)

  `search-forward` en `re-search-forward` hebben vier argumenten:

  1. De string of reguliere expressie om naar te zoeken.
  2. Optioneel, de limiet van de zoekopdracht.
  3. Optioneel, wat te doen als de zoekopdracht mislukt, geef `nil` terug of een fout.
  4. Optioneel, hoeveel keer de zoekopdracht te herhalen. Wanneer negatief, zoek achterwaarts.

`kill-region`
`delete-and-extract-region`
`copy-region-as-kill`
: `kill-region` verwijdert de tekst tussen point en mark uit het buffer en bewaart die tekst in de killring, zodat je die met yanken terug kunt krijgen.

  `copy-region-as-kill` kopieert de tekst tussen point en mark naar de killring, van waar je het met yanken terug kunt krijgen. De functie verwijdert de tekst niet uit het buffer.

`delete-and-extract-region` verwijdert de tekst tussen point en mark uit het buffer en gooit het weg. Je kunt het niet terug krijgen. (Dit is geen interactief commando).

8.7 Zoek-oefeningen

- Schrijf een interactieve functie die naar een string zoekt. Wanneer de string gevonden is, plaats point er achter en toon een boodschap die zegt "Gevonden!". (Maak geen gebruik van `search-forward` als naam van deze functie, wanneer je dat doet overschrijf je de bestaande versie van `search-forward` die bij Emacs hoort. Gebruik in plaats daarvan een naam zoals `test-zoeken`.)
- Schrijf een functie die het derde element van de killring toont in het echogebied als dat element er is. Wanneer de killring geen derde element heeft, toon dat een passende boodschap.

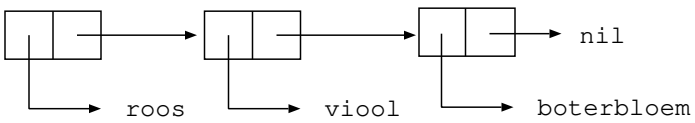
9 Hoe lijsten zijn geïmplementeerd

In Lisp worden atomen op een eenvoudige manier vastgelegd. Zelfs als de implementatie niet eenvoudig is, dan is de theorie toch eenvoudig. Het atoom '`roos`' bijvoorbeeld is vastgelegd als vier aaneengesloten letters, '`r`', '`o`', '`o`' en '`s`'. Een lijst, anderzijds, wordt anders bewaard. Het mechanisme is net zo eenvoudig, maar het kost even tijd om aan het idee gewend te raken. Een lijst wordt bewaard met een reeks paren van pointers. In de reeks wijst de eerste pointer van elk paar pointers naar een atoom of naar een andere lijst, en de tweede pointer in elk paar pointers wijst naar het volgende paar, of naar het symbool `nil`, wat het eind van de lijst markeert.

Een pointer zelf is heel eenvoudig het elektronische adres van datgene, waar het naar toe wijst. Een lijst wordt dus bewaard als een reeks elektronische adressen.

Bijvoorbeeld de lijst `(roos, viool, boterbloem)` heeft drie elementen, '`roos`', '`viool`' en '`boterbloem`'. In de computer wordt het elektronische adres van '`roos`' bewaard in een segment van het computergeheugen dat een '`cons-cel`' heet (omdat dat is wat de functie `cons` werkelijk creëert). Deze cons-cel bevat ook het adres van een tweede cons-cel, wiens CAR het atoom '`viool`' is, en dat adres (degene die vertelt waar '`viool`' is bewaard) samen met het adres van een derde cons-cel, die het adres voor het atoom '`boterbloem`' bevat.

Dit klinkt gecompliceerder dan het is en is makkelijker te zien in een diagram:

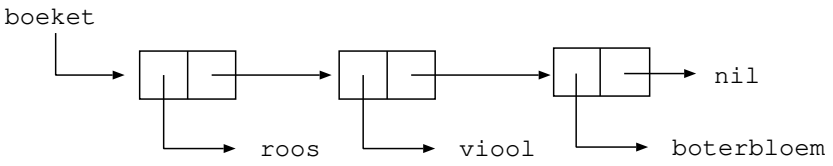


Elk vakje in het diagram representeert een computergeheugen woord dat een Lisp object bevat, meestal in de vorm van een geheugenadres. De vakjes, oftewel de adressen, zijn paren. Elke pijl wijst naar wat het adres een adres van is, hetzij een atoom of een ander paar adressen. Het eerste vakje is het elektronische adres van '`roos`' en de pijl wijst naar '`roos`'. Het tweede vakje is het adres van het volgende paar vakjes, waarvan het eerste deel het adres van '`viool`' is en waarvan het tweede deel het adres van het volgende paar is. Het allerlaatste vakje wijst naar het symbool `nil`, dat het einde van de lijst markeert.

Wanneer een variabele met een operatie zoals `setq` wordt ingesteld op een lijst, dan slaat dat het adres op van het eerste vakje. Dus, evaluatie van de expressie

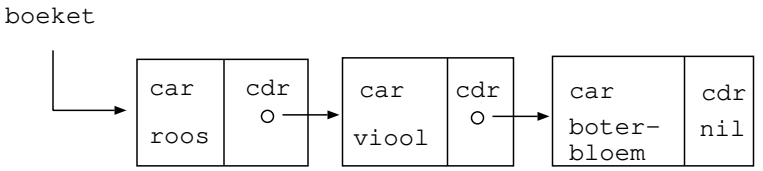
```
(setq boeket '(roos viool boterbloem))
```

creëert een situatie zoals dit:



In dit voorbeeld bevat het symbool `boeket` het adres van het eerste paar vakjes.

Dezelfde lijst kan met een ander soort vakjes-notatie worden geïllustreerd, zoals dit:

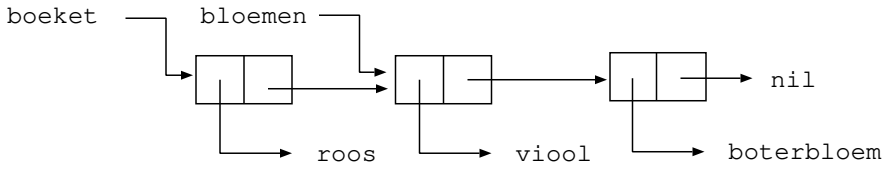


(Symbolen bestaan uit meer dan paren van adressen, maar de structuur van een symbool is gemaakt van adressen. Het symbool `boeket` bestaat inderdaad uit een groep van adresvakjes, waarvan een het adres van het getoonde woord '`boeket`' is, een tweede daarvan het adres van de aan het symbool gekoppelde functiedefinitie is, als die er is, en een derde daarvan dat het adres van het eerste paar adresvakjes voor de lijst `roos viool boterbloem` is, enzovoorts. Hier tonen we dat het derde adresvakje van het symbool naar het eerste paar van de adresvakjes van de lijst wijst.)

Wanneer een symbool naar de CDR van de lijst wordt gezet, verandert de lijst zelf niet. Het symbool heeft eenvoudig een adres verderop in de lijst. (In het jargon: CAR en CDR zijn niet-destructief). Dus de evaluatie van de volgende expressie

```
(setq bloemen (cdr boeket))
```

produceert dit:



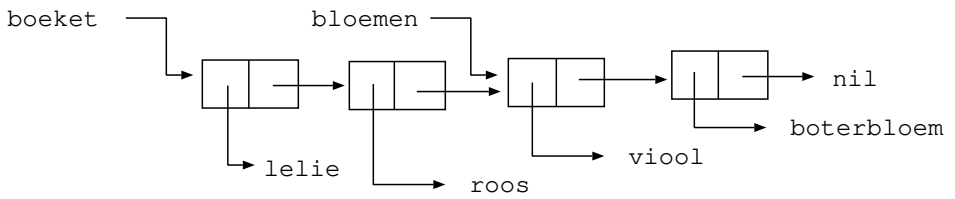
De waarde van `bloemen` is `(viool boterbloem)`, dat wil zeggen, het symbool `bloemen` bevat het adres van het paar adresvakjes, waarvan de eerste het adres van `viool` bevat en de tweede het adres van `boterbloem` bevat.

Een paar adresvakjes heet een *cons-cel* of *dotted pair*. Zie Sectie "Cons Cell and List Types" in *The GNU Emacs Lisp Reference Manual*, en Sectie "Dotted Pair Notation" in *The GNU Emacs Lisp Reference Manual*, voor meer informatie over cons-cellen en dotted pairs.

De functie `cons` voegt een nieuw paar adressen aan de voorkant van de reeks van adressen zoals hierboven getoond. Bijvoorbeeld, het evalueren van de expressie

```
(setq boeket (cons 'lelie boeket))
```

produceert:



Dit verandert echter niet de waarde van het symbool `bloemen`, zoals je kunt zien door het volgende te evalueren:

```
(eq (cdr (cdr boeket)) bloemen)
```

wat `t` voor waar teruggeeft.

Totdat het wordt gereset, heeft `bloemen` nog steeds de waarde `(viool boterbloem)`, dat wil zeggen, het heeft het adres van de cons-cel wiens eerste adres dat van `viool` is. Ook verandert dit geen van de reeds bestaande cons-cellen, die zijn er nog steeds.

Dus om in Lisp de CDR van een lijst te krijgen, krijg je alleen het adres van de volgende cons-cel in de reeks. Om de CAR van een lijst te krijgen, krijg je het adres van het eerste element van de lijst. Om een nieuw element aan een lijst te `cons`-en, voeg je een nieuwe cons-cel aan de voorkant van de lijst toe. Dat is alles! De onderliggende structuur van Lisp is briljant eenvoudig!

En waar refereert het laatste adres in een reeks van cons-cellen naar toe? Het is het adres van de lege lijst, van `nil`.

Samengevat, wanneer een Lisp variabele op een waarde wordt ingesteld, dan krijgt die het adres van de lijst waar de variabele naar refereert.

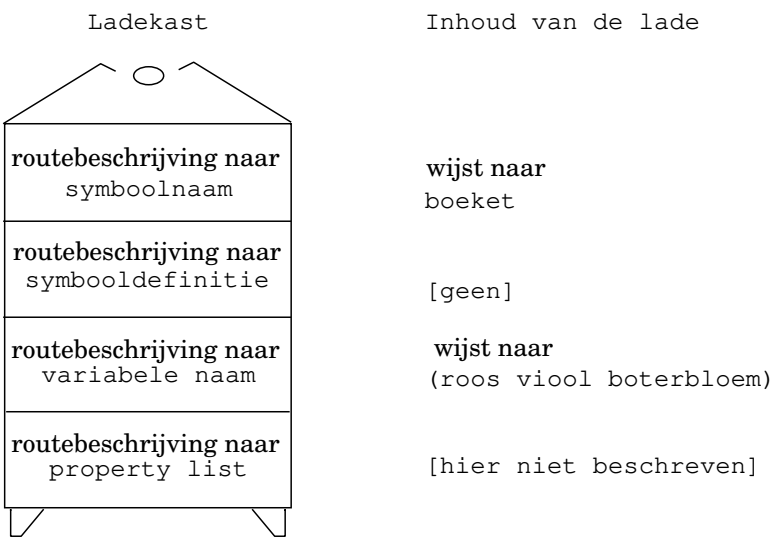
9.1 Symbolen als een ladekast

In een eerdere sectie suggereerde ik dat je een symbool kunt voorstellen als een ladekast. De functiedefinitie wordt in een lade gestopt, de waarde in een andere, enzovoorts. Wat in de lade die de waarde bevat gestopt is, kan worden gewijzigd zonder de inhoud van de waarde die de functiedefinitie bevat te wijzigen, en andersom.

Wat eigenlijk in elke lade gestopt wordt, is het adres van de waarde of de functiedefinitie. Het is alsof je een oude kast op de zolder gevonden hebt, en in een van de lades een kaart vond die de route beschrijft naar waar de begraven schat ligt.

Naast zijn naam, symbooldefinitie en variabele naam, heeft een symbool een lade voor een *property list* die kan worden gebruikt om andere informatie op te slaan. Property lists bespreken we hier niet, zie Sectie “Property Lists” in *The GNU Emacs Lisp Reference Manual*.)

Hier is een fantasierijke weergave:



9.2 Oefening

Stel `bloemen` in op `viool` en `boterbloem`. Cons twee andere bloemen aan deze lijst en zet deze lijst op `meer-bloemen`. Zet de CAR van bloemen op een vis. Wat bevat de lijst `meer-bloemen` nu?

10 Tekst terug yanken

Wanneer je in GNU Emacs met een kill commando tekst uit een buffer verwijdert dan kun je die terughalen met een yank commando. De tekst die uit het buffer verwijderd is, wordt in de killring gestopt en het yank commando voegt de juiste inhoud van de killring terug in een buffer (niet noodzakelijkerwijs het oorspronkelijke buffer).

Een eenvoudig *C-y* (`yank`) commando voegt het eerste element van de killring in het huidige buffer. Wanneer het *C-y* commando direct gevolg wordt door *M-y*, wordt het eerste element vervangen door het tweede element. Herhaaldelijke *M-y* commando's vervangen het teede element met het derde, vierde of vijfde element, enzovoorts. Wanneer het laatste element in de killring is bereikt, wordt het vervangen door het eerste en wordt de cyclus herhaald. (Daarom heet de killring een "ring" in plaats van een "lijst". De werkelijke datastructuur die de tekst bevat is echter een lijst. Zie Appendix B "De killring hanteren", pagina 230, voor de details hoe de lijst wordt gehanteerd als ring.)

10.1 Killring overzicht

De killring is een lijst van tekstuele strings. Dit is hoe die er uit ziet:

```
("wat tekst" "een ander stuk tekst" "nog meer tekst")
```

Wanneer dit de inhoud van mijn killring is en ik sla *C-y* aan, dan wordt de karakterstring '`wat tekst`' in dit buffer op de cursorlokatie ingevoegd.

Het `yank` commando wordt ook gebruikt om tekst te dupliceren door het te kopiëren. De gekopieerde tekst wordt niet uit het buffer verwijderd, maar een kopie er van is in de killring geplaatst en onmiddellijk ingevoegd door het terug te yanken.

Drie functies worden gebruikt om de tekst uit de killring terug te halen: `yank`, dat normaalgesproken gebonden is aan *C-y*. `yank-pop`, dat normaalgesproken gebonden is aan *M-y*, en `rotate-yank-pointer` dat door de twee andere functies gebruikt wordt.

Deze functies refereren aan de killring via de variabele met de naam `kill-ring-yank-pointer`. Inderdaad, de invoegcode van zowel de functie `yank` als `yank-pop` is:

```
(insert (car kill-ring-yank-pointer))
```

(Nou, niet meer. In GNU Emacs 22 is de functie vervangen door `insert-for-yank`, die herhaaldelijk `insert-for-yank-1` aanroept voor elk `yank-handler` segment. `insert-for-yank-1`, op zijn beurt, stript de tekst eigenschappen van de ingevoegde tekst in overeenstemming met `yank-excluded-properties`. Anders dat dat, is het gewoon als `insert`. Wij houden het bij de `insert`, dit is makkelijker te begrijpen.)

Om te beginnen te begrijpen hoe `yank` en `yank-pop` werken, is het eerst nodig naar de variabele `kill-ring-yank-pointer` te kijken.

10.2 De variabele `kill-ring-yank-pointer`

`kill-ring-yank-pointer` is een variabel, net zoals `kill-ring` een variabele is. Het wijst naar iets dat is gebonden aan de waarde van waar het naar wijst, net als elke andere Lisp variabele.

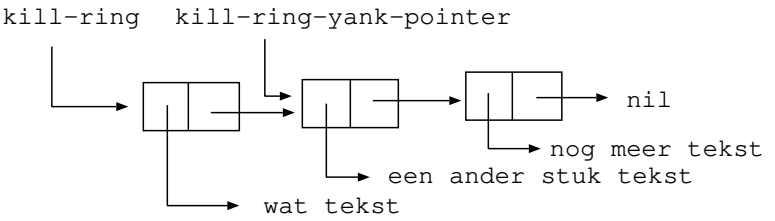
Dus, als de waarde van de killring is:

```
("wat tekst" "een ander stuk tekst" "nog meer tekst")
```

en de `kill-ring-yank-pointer` wijst naar de tweede clausule, dan is de waarde van `kill-ring-yank-pointer`:

```
("een ander stuk tekst" "nog meer tekst")
```

Zoals in het vorige hoofdstuk (zie Hoofdstuk 9 “Implementatie van lijsten”, pagina 108) uitgelegd, bewaart de computer geen twee verschillende exemplaren van de tekst waar zowel de `kill-ring` als de `kill-ring-yank-pointer` naar wijzen. De woorden “een ander stuk tekst” en “nog meer tekst” worden niet gedupliceerd. In plaats daarvan wijzen de twee Lisp variabelen naar hetzelfde stuk tekst. Hier is een diagram:



Zowel de variabele `kill-ring` als de variabele `kill-ring-yank-pointer` zijn pointers. Maar de killring zelf wordt meestal beschreven alsof het is wat het van gemaakt is. De killring wordt beschreven alsof het een lijst is in plaats van dat het naar de lijst wijst. Omgekeerd, de `kill-ring-yank-pointer` is beschreven alsof het wijst naar een lijst.

Deze twee manieren om over het zelfde ding te spreken klinken op het eerste gezicht verwarrend maar zijn bij nader inzien logisch. De killring wordt algemeen beschouwd als de complete structuur van data die de informatie bevat van wat recentelijk uit de buffers is geknipt. De `kill-ring-yank-pointer` anderzijds dient om aan te geven—dat wil zeggen, te wijzen naar—dat deel van de killring waarvan het eerste element (the CAR) zal worden ingevoegd.

10.3 Oefeningen met `yank` en `nthcdr`

- Kijk met *C-h v* (`describe-variable`) naar je killring. Voeg verschillende items aan je killring toe en kijk opnieuw naar zijn waarde. Ga met *M-y* (`yank-pop`) helemaal rond in de killring. Hoeveel items waren in je killring? Vind de waarde van `kill-ring-max`. Was je killring vol, of had je nog meer blokken tekst kunnen bewaren?
- Construeer met `nthcdr` en `car` een reeks expressies om het eerste, tweede, derde en vierde element van een lijst terug te geven.

11 Loops en recursie

Emacs Lisp heeft twee primaire manieren om een expressie of een reeks van expressies herhaaldelijk te laten evalueren: een met gebruik van een `while` loop en een met gebruik van *recursie*.

Herhaling kan zeer waardevol zijn. Bijvoorbeeld om vier zinnen naar voren te gaan, hoef je alleen een programma te schrijven dat een zin naar voren gaat, en dat dan vier keer te herhalen. Omdat een computer niet verveeld of vermoeid raakt, heeft zo'n herhalende actie geen schadelijke effecten dat overdreven of verkeerde manier van herhalen op mensen kan hebben.

Men schrijft meestal Emacs Lisp functies met `while` loops en hun soortgenoten. maar je kunt ook recursie gebruiken, wat een erg krachtige manier biedt om over problemen te denken en op te lossen[1]

11.1 `while`

De speciale vorm `while` test of de waarde teruggeven door het eerste argument te evalueren waar of onwaar is. Dit is vergelijkbaar met wat de Lisp interpreter doet met een `if`. Wat de interpreter daarna doet is echter anders.

Wanneer in een `while` expressie de waarde teruggeven door het eerste argument te evalueren onwaar is, slaat de Lisp interpreter de rest van de expressie (de *body* van de expressie) over en evalueert die niet. Wanneer echter de waarde waar is, can evalueert de Lisp interpreter de body van de expressie en test opnieuw of het eerste argument van `while` waar is of onwaar. Wanneer de waarde teruggegeven door het eerste argument te evalueren opnieuw waar is, evalueert de Lisp interpreter opnieuw de body van de expressie.

Het sjabloon voor een `while` expressie ziet er zo uit:

```
(while waar-of-onwaar-test
  body...)
```

Zolang de waar-of-onwaar-test van de `while` expressie een waarde waar teruggeeft wanneer die wordt geëvalueerd, wordt de body herhaaldelijk geëvalueerd. Dit proces wordt een loop genoemd, omdat de Lisp interpreter het zelfde ding opnieuw en opnieuw herhaalt, zoals een vliegtuig een looping maakt. Wanneer het resultaat van het evalueren van de waar-of-onwaar-test onwaar is, evalueert de Lisp interpreter de rest van de `while` expressie niet, en verlaat die de loop.

Het is duidelijk dat wanneer het evalueren van het eerst argument van de `while` altijd waar teruggeeft, de daarop volgende body wordt opnieuw, opnieuw . . . en opnieuw . . . voor altijd herhaald. Omgekeerd, wanneer de teruggegeven waarde nooit waar is, de expressies in de body nooit geëvalueerd zullen worden. Het kunst van het

[1] Je kunt recursieve functies schrijven die zuinig of verspillend omgaan met mentale- of computermiddelen. Methoden die mensen makkelijk vinden—die zuinig met mentale middelen omgaan—kunnen soms aanzienlijke computermiddelen gebruiken. Emacs was ontworpen om op machines te draaien die we nu beperkt vinden en de standaard instellingen zijn terughoudend. Je wilt misschien de instelling van `max-lisp-eval-depth` verhogen. In mijn `.emacs` bestand heb ik het op 30 keer de standaard waarde ingesteld.

schrijven van `while` loops bestaat uit het kiezen van zo'n mechanisme dat de waar-of-onwaar-test precies zo veel keer waar teruggeeft als je wilt dat de daaropvolgende expressies worden geëvalueerd en daarna de test onwaar laat teruggeven.

De waarde die het evalueren van een `while` teruggeeft, is de waarde van de waar-of-onwaar-test. Een interessant gevolg hiervan is dat een `while` loop die zonder fouten evalueert een `nil` of onwaar teruggeeft, ongeacht of het 1 of 100 keer heeft geloopt, of helemaal nooit. Een `while` loop die succesvol evalueert geeft nooit een waarde waar terug! Dit betekent dat `while` altijd geëvalueerd wordt voor zijn zij-effecten, met andere woorden, de consequenties van het evalueren van de expressies in de body van de `while` loop. Dat is logisch. Het is niet slechts het loopen dat gewest is, maar de consequenties van wat gebeurt wanneer de expressies in de loop herhaaldelijk worden geëvalueerd.

11.1.1 Een `while` loop en een lijst

Een gebruikelijke manier om een `while` loop te beheersen is testen of een lijst elementen heeft. Wanneer dat zo is, wordt de loop herhaald, maar wanneer dat niet zo is, wordt de herhaling beëindigd. Omdat dit een belangrijke techniek is, maken we hier een klein voorbeeld om het te illustreren.

Een simpele manier om te testen of een lijsten elementen heeft is de lijst te evalueren: wanneer die geen elementen heeft is het een lege lijst en geeft het een lege lijst terug, `()`, wat een synoniem is voor `nil` of onwaar. Anderzijds geeft een lijst met elementen die elementen terug wanneer die wordt geëvalueerd. Omdat Emacs Lisp elke waarde ongelijk aan `nil` als waar beschouwt, en lijst die elementen teruggeeft zal als waar testen in een `while` loop.

Als voorbeeld kun je de variabele `lege-lijst` op `nil` zetten met het evalueren van de volgende `setq` expressie:

```
(setq lege-lijst ())
```

Na het evalueren van de `setq` expressie, kan je de variabele `lege-lijst` op de gebruikelijke manier evalueren, door de cursor achter het symbool te plaatsen en *`C-x C-e`* te typen. In het echogebied verschijnt `nil`.

```
lege-lijst
```

Anderzijds, als je een variabele instelt op een lijst met elementen, verschijnt de lijst wanneer je de variabele evalueert. Dit kan je zien door de volgende expressies te evalueren:

```
(setq dieren '(gazelle giraf leeuw tijger))

dieren
```

Dus om een `while` loop te creëren die test of er elementen in de lijst `dieren` zijn, is het eerste deel van de loop als volgt:

```
(while dieren
       ...
```

Wanneer de `while` het eerste argument test, wordt de variabele `dieren` geëvalueerd. Dit geeft een lijst terug. Zo lang de lijst elementen heeft, beschouwt de `while` het resultaat van de test als waar, maar wanneer de lijst leeg is, dan beschouwt die het resultaat van de test als onwaar.

Om te voorkomen dat de `while` loop voor altijd blijft lopen, is een mechanisme nodig dat uiteindelijk een lege lijst geeft. Een vaak gebruikte techniek is om een van de opvolgende vormen in de `while` expressie de waarde van de lijst op de CDR van de lijst te zetten. Elke keer dat de functie CDR wordt geëvalueerd wordt de lijst korter, totdat uiteindelijk alleen de lege lijst overblijft. Op dit punt geeft de test in de `while` loop onwaar terug, en de argumenten van de `while` worden niet langer meer geëvalueerd.

De lijst van dieren bijvoorbeeld, gebonden aan de variabele `dieren` kan op de CDR van de oorspronkelijke lijst gezet worden met de volgende expressie:

```
(setq dieren (cdr dieren))
```

Wanneer je de voorgaande expressies geëvalueerd hebt en vervolgens deze expressie, dan zie je `(giraf leeuw tijger)` in het echogebied verschijnen. Wanneer je de expressie nog een keer evalueert, dan verschijnt `(leeuw tijger)` in het echogebied. Wanneer je het nog een keer en daarna nog een keer evalueert, dan verschijnt `tijger` en daarna de lege lijst, getoond met `nil`.

Een sjabloon voor de `wile` loop dat de functie `cdr` herhaaldelijk gebruikt om de waar-of-onwaar-test uiteindelijk als onwaar te testen, ziet er zo uit:

```
(while test-of-lijst-leeg-is
  body...
  zet-lijst-op-cdr-van-lijst)
```

Deze test en het gebruik van `cdr` kunnen samengevoegd worden in een functie die door een lijst gaat en elk element van de lijst op een regel op een eigen regel toont.

11.1.2 Een voorbeeld: `toon-elementen-van-lijst`

De `toon-elementen-van-lijst` functie illustreert een `while` loop met een lijst.

Deze functie benodigd meerdere regels voor de output. Wanneer je dit in een recente versie van GNU Emacs leest, kan je de volgende expressies in Info evalueren, zoals gebruikelijk.

Wanneer je een eerdere versie van Emacs gebruikt, kan het nodig zijn eerst de expressies te kopiëren naar je `*scratch*` buffer en ze daar te evalueren. Dit is omdat het echogebied in eerdere versies meer een regel bevat.

Je kunt de expressies kopiëren door het begin van de region te markeren met *C*-SPC (`set-mark-command`), vervolgens de cursor naar het eind van de region te verplaatsen en dan de region met *M*-`w` (`kill-ring-save` te kopiëren (die `copy-region-as-kill` aanroept en visuele feedback geeft). In het `*scratch*` buffer krijg je de expressies terug door *C*-`y` (`yank`) te typen.

Nadat je de expressies in het `*scratch*` buffer gekopieerd hebt, evalueer je elke expressie op zijn beurt. Zorg dat je de laatste expressie `toon-elementen-van-lijst dieren` met *C*-`u` *C*-`x` *C*-`e` evalueert, oftewel door een argument aan `eval-last-sexp` te geven. Hierdoor wordt het resultaat van de evaluatie in het `*scratch*` buffer getoond, in plaats van in het echogebied. (Anders zie je iets als dit in het echogebied: `^Jgazelle^J^Jgiraf^J^Jleeuw^J^Jtijger^Jnil`, waarbij elk '`^J`' voor een nieuwe regel staat.)

Je kunt de volgende expressie direct in het Info buffer evalueren en het echogebied groeit om de resultaten te tonen.

```
(setq dieren '(gazelle giraf leeuw tijger))

(defun toon-elementen-van-lijst (lijst)
  "Toon elk element van LIJST op een eigen regel."
  (while lijst
    (print (car lijst))
    (setq lijst (cdr lijst))))
(toon-elementen-van-lijst dieren)
```

Wanneer je de drie expressies achtereenvolgens evalueert, zie je dit:

```
gazelle

giraf

leeuw

tijger
nil
```

Elk element van de lijst wordt op een eigen regel getoond (dat is wat de functie `print` doet) en daarna wordt de waarde die de functie teruggeeft getoond . Omdat de laatste expressie van de functie de `while` loop is, en omdat een `while` loop altijd `nil` teruggeeft, wordt `nil` getoond na het laatste element van de lijst.

11.1.3 Een loop met een incrementele teller

Een loop is niet bruikbaar tenzij die stopt wanneer dat moet. Naast het beheersen van een loop met een lijst is een gebruikelijke manier om een loop te stoppen door het eerste argument te schrijven als een test die onwaar teruggeeft wanneer het juiste aantal herhalingen is uitgevoerd. Dat betekent dat de loop een teller moet hebben—een expressie die telt hoeveel maal de loop zichzelf herhaald heeft.

De test voor een loop met een incrementele teller kan een expressie zijn zoals `(< telling gewenste-aantal)` die `t` voor waar teruggeeft wanneer de waarde van `telling` kleiner is dan het `gewenste-aantal` herhalingen, en `nil` voor onwaar wanneer de waarde van `telling` groter of gelijk is aan het `gewenste-aantal`. De expressie die de telling verhoogt kan een simpele `setq` zijn, zoals `(setq telling (1+ telling))`. waar `1+` een in Emacs Lisp een ingebouwde functie is die 1 bij zijn argument optelt. (De expressie `(1+ telling)` heeft hetzelfde resultaat als `(+ telling 1)` maar is voor mensen makkelijker te lezen.)

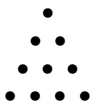
Het sjabloon voor een `while` loop beheerst met een incrementele teller ziet er zo uit:

```
zet-telling-op-initiele-waarde
(while (< telling gewenste-aantal)          ; waar-of-onwaar-test
  body...
  (setq telling (1+ telling)))               ; incrementer
```

Merk op dat je de initiële waarde van `telling` moet zetten, gebruikelijk is die op 1 te zetten.

Voorbeeld met een incrementele teller

Stel je voor dat je op het strand aan spelen bent en besluit om een driehoek van steentjes te maken, met één steentje op de eerste rij, twee op de tweede rij, drie op de derde rij, enzovoorts, zoals dit:



(Ongeveer 2500 jaar geleden ontwikkelden Pythagoras en anderen het begin van getaltheorie door vragen zoals deze te beschouwen.)

Stel je voor dat je wilt weten hoeveel steentjes je nodig hebt om een driehoek met 7 rijen te maken?

Het is duidelijk, wat we moeten doen is de getallen 1 tot en met 7 op te tellen. Er zijn twee manieren om dit te doen: beginnen met het kleinste nummer, een, en dan de lijst opvolgend optellen, 1, 2, 3, 4, enzovoorts. Of beginnen met het grootste getal en dan de lijst aflopend optellen, 7, 6, 5, 4 enzovoorts. Omdat beide mechanismen gebruikelijke manieren om `while` loops te schrijven illustreren, maken we twee voorbeelden, eentje oplopend tellen en eentje aflopend tellen. In het eerste voorbeeld beginnen we met 1, en tellen dan 2, 3, 4 enzovoorts op.

Wanneer je slechts een korte lijst getallen optelt, is de makkelijkste manier daarvoor om alle getallen in een keer op te tellen. Maar wanneer je van te voren niet weet hoeveel getallen de lijst bevat, of als je voorbereid wilt zijn op een erg lange lijst, dan moet je de optelling zo ontwerpen dat je een eenvoudig proces veel keren herhaalt, in plaats van het in een keer uitvoeren van een meer complex proces.

Bijvoorbeeld in plaats van alle steentjes in een keer bij elkaar op te tellen, kun je het aantal steentjes in de eerste rij, 1, optellen bij het aantal in de tweede rij, 2, daarna tel je het totaal van die twee rijen op bij de derde rij, 3. Daarna tel je het aantal van de vierde rij, 4, bij het totaal van de eerste drie rijen, enzovoorts.

De kritieke eigenschap van het proces is dat elke terugkerende actie eenvoudig is. In dit geval tellen we bij elke stap slechts twee getallen op, het aantal steentjes in de rij en het totaal dat we al hadden. Dit proces van het optellen van twee getallen wordt opnieuw herhaald en opnieuw, totdat de laatste rij is opgeteld bij het totaal van de voorgaande rijen. In een meer complexe loop kan de terugkerende actie minder eenvoudig zijn, maar het is eenvoudiger dan alles in een keer doen.

De onderdelen van de functiedefinitie

De voorgaande analyse geeft ons het skelet van onze functiedefinitie: als eerste hebben we een variabele nodig die we `totaal` noemen, dat het totaal van het aantal steentjes gaat worden. Dit is de waarde de de functie gaat teruggeven.

Ten tweede weten we dat de functie een argument nodig heeft, dit argument wordt het totaal aantal rijen in de driehoek. Het kan `aantal-rijen` heten.

Tenslotte hebben we een variabele als teller nodig. We kunnen deze variabele `teller` noemen, maar een betere naam is `rij-nummer`. Dat is omdat de teller in deze functie de rijen telt, en een programma moet zo begrijpelijk mogelijk worden geschreven.

Wanneer de Lisp interpreter voor het eerst start met het evalueren van de expressies in de functie moet de waarde van `totaal` op nul staan, omdat we er nog niets bij opgeteld hebben. Daarna moet de functie het aantal steentjes in de eerste rij bij het totaal optellen en daarna het aantal steentjes in de tweede rij bij het totaal optellen, en dan het aantal steentjes in de derde rij, enzovoorts, totdat er geen rijen meer over zijn om er bij te tellen.

`totaal` en `rij-nummer` worden beide alleen binnen de functie gebruikt, dus kan `let` ze als lokale variabelen declareren en ze een initiële waarde geven. Het is duidelijk dat de initiële waarde voor `totaal` 0 moet zijn. De initiële waarde voor `rij-nummer` moet 1 zijn, omdat we beginnen met de eerste rij. Dit betekent dat de `let` statement er zo uit ziet:

```
(let ((totaal 0)
      (rij-nummer 1))
  body...)
```

Nadat de interne variabelen zijn gedeclareerd en gebonden aan hun initiële waarden, kunnen we met de `while` loop beginnen. De expressie die dient als test moet een waarde van `t` voor waar teruggeven zolang het `rij-nummer` kleiner of gelijk is aan het `aantal-rijen`. (Wanneer de expressie alleen waar test zolang als het rijnummer kleiner is dan het aantal rijen in de driehoek, dan wordt de laatste rij nooit bij het totaal aantal geteld, daarom moet het rijnummer kleiner of gelijk aan het aantal rijen zijn.)

Lisp biedt de functie `<=` die waar teruggeeft indien de waarde van het eerste argument is kleiner of gelijk aan de waarde van het tweede argument, zo niet dan geeft het onwaar terug. De expressie die de `while` als test evalueert moet er zo uitzien:

```
(<= rij-nummer aantal-rijen)
```

Het totale aantal steentjes kan worden bepaald door herhaaldelijk het aantal steentjes in een rij op te tellen bij het reeds getelde totaal aantal. Omdat het aantal steentjes in een rij gelijk is aan het rij-nummer, kan het totaal bepaald worden door het rij-nummer bij het totaal op te tellen. (In een meer complexe situatie kan het aantal steentjes in een rij op een meer gecompliceerde manier gerelateerd zijn aan het rij-nummer. Wanneer dat het geval is dan zou het rij-nummer vervangen worden door een passende expressie.)

```
(setq totaal (+ totaal rij-nummer))
```

Dit zet de nieuwe waarde van `totaal` gelijk aan de som van het aantal steentjes in de rij en het vorige totaal.

Na het zetten van de waarde van `totaal` moeten de condities worden ingesteld voor de volgende herhaling van de loop, als die er is. Dit gebeurt door de waarde van de variabele `rij-nummer` te verhogen, die als teller dient. Nadat de variabele `rij-nummer` is verhoogd, test de waar-of-onwaar-test aan het begin van de `while` loop of de zijn waarde nog steeds kleiner of gelijk is aan de waarde van de `aantal-rijen` en als dat zo is, telt het de nieuwe waarde van de variable `rij-nummer` bij de `totaal` van de vorige herhaling van de loop.

De in Emacs Lisp ingebouwde functie `1+` telt 1 bij een getal op, dus de variabele `rij-nummer` kan met deze expressie worden verhoogd:

```
(setq rij-nummer (1+ rij-nummer))
```

De functiedefinitie samenstellen

We hebben de onderdelen van de functiedefinitie gecreëerd, nu moeten we ze samenstellen.

Ten eerste de inhoud vna de `while` expressie:

```
(while (<= rij-nummer aantal-rijen)    ; waar-of-onwaar-test
  (setq totaal (+ total rij-nummer))
  (setq rij-nummer (1+ rij-nummer)))     ; incrementer
```

Samen met de `let` expressie varlist maakt dit de body van de functie bijna helemaal compleet. Het heeft echter nog een laatste element nodig, waarvan de noodzaak wat subtiel is.

Het puntje op de i is om de variabele `totaal` na de `while` expressie op een eigen regel te plaatsen. Zo niet dan is de waarde die de hele functie terugggeeft de waarde van de laatste expressie die is geëvalueerd in de body van de `let`, dat is de waarde teruggeven door de `while` en die is altijd `nil`.

Dit kan op het eerste gezicht niet vanzelfsprekend zijn. Het lijkt bijna of de verhogende expressie de laatste expressie van de hele functie is. Maar die expressie is onderdeel van de body van de `while`. Bovendien is de hele `while` loop een lijst binnen de body van de `let`.

Een overzicht van de functie ziet er zo uit:

```
(defun naam-van-functie (argument-lijst)
  "documentatie..."
  (let (varlist)
    (while (waar-of-onwaar-test)
      body-van-while... )
    ... ))                          ; Hier is een laatste expressie nodig.
```

Het resultaat van het evalueren van de `let` is wat de `defun` gaat teruggeven, omdat de `let` niet ingebed is een lijst, behalve binnen de `defun` als geheel. Echter, als de `while` het laatste element is van de `let` expressie, geeft de functie altijd `nil` terug. Dit is niet wat we willen! In plaats daarvan willen we dat de functie de waarde van de variabele `totaal` teruggeeft. Dit wordt teruggegeven door eenvoudig dat symbool als laatste element in de lijst te zetten die begint met `let`. Het wordt geëvalueerd nadat de voorgaande elementen van de lijst zijn geëvalueerd, wat betekent dat het wordt geëvalueerd nadat het de correcte waarde voor het totaal heeft gekregen.

Het is eenvoudiger te zien wanneer we de lijst die begint met `let` in zijn geheel op een regel tonen. Dit formaat maakt het vanzelfsprekend dat de varlist en de `while` expressie het tweede en derde element van de lijst die begint met `let` zijn, en dat `totaal` het laatste element is.

```
(let (varlist) (while (waar-of-onwaar-test) body-of-while... ) totaal)
```

Als we alles samenvoegen, ziet de functie `driehoek` er zo uit:

```
(defun driehoek (aantal-rijen)     ; Versie met
                                   ;   oplopende teller.
  "Tel het aantal steentjes in een driehoek op.
De eerste rij heeft een steentje, de tweede rij twee steentjes.
De derde rij heeft drie steentjes, enzovoorts.
Het argument is AANTAL-RIJEN."
  (let ((totaal 0)
        (rij-nummer 1))
    (while (<= rij-nummer aantal-rijen)
      (setq totaal (+ totaal rij-nummer))
      (setq rij-nummer (1+ rij-nummer)))
    totaal))
```

Wanneer je `driehoek` hebt geïnstalleerd door het evalueren van de functie, kun je hem uitproberen. Hier zijn twee voorbeelden:

```
(driehoek 4)

(driehoek 7)
```

De som van de eerste vier getallen is 10 en de som van de eerste zeven getallen is 28.

11.1.4 Loop met een decrementele teller

Een andere gebruikelijke manier om een `while` loop te schrijven is met een test die vaststelt of een teller groter dan nul is. Zolang de teller groter dan nul is wordt de loop herhaald. Maar wanneer de teller kleiner of gelijk is aan nul, wordt de loop gestopt. Om dit zo te laten werken moet de teller groter dan nul starten en dan kleiner en kleiner gemaakt worden door een vorm die herhaaldelijk wordt geëvalueerd.

De test is een expressie zoals `(> teller 0)` die `t` voor waar teruggeeft wanneer de waarde van `teller` groter is dan nul, en `nil` voor onwaar wanneer de waarde van `teller` kleiner of gelijk is aan nul. De expressie die het getal kleiner en kleiner makat kan een eenvoudige `setq` zijn zoals `(setq teller (1- teller))`, waar `1-` een in Emacs Lisp ingebouwde functie die 1 van zijn argument aftrekt.

Het slabloon voor een decrementele `while` loop ziet er zo uit:

```
(while (> teller 0)                    ; waar-of-onwaar-test
  body...
  (setq teller (1- teller)))           ; decrementer
```

Voorbeeld met een aflopende teller

Om een loop met een aflopende teller te laten zien, gaan we de functie `driehoek` zo herschrijven dat de teller naar nul vermindert.

Dit is het omgekeerde van de eerdere versie van de functie. In dit geval om te ontdekken hoeveel steentjes nodig zijn om een driehoek met 3 rijen te maken, tel het aantal steentjes in de derde rij, 3, bij het aantal in de voorgaande rij, 2, en daarna tel het totaal van die rijen bij de rij die aan ze voorafgaat, die 1 is.

Om op dezelfde manier het aantal steentjes in een driehoek met 7 rijen te ontdekken, tel het aantal steentjes in de zevende rij, 7, bij het aantal van de voorgaande rij, dat 6 is, en tel dan het totaal van deze twee rijen bij de rij die aan ze voorafgaat, dat 5 is, enzovoorts. Net als in het vorige voorbeeld betreft elke optelling twee getallen het totaal van de rijen die al geteld zijn en het aantal steentjes in de rij die wordt opgeteld bij het totaal. Dit proces van twee getallen optellen wordt opnieuw en opnieuw herhaald totdat er geen steentjes meer zijn op op te tellen.

We weten met hoeveel steentjes we moeten beginnen: het aantal steentjes in de laatste rij is gelijk aan het aantal rijen. Wanneer de driehoek zeven rijen heeft, dan is het aantal steentjes in de laatste rij 7. Op dezelfde manier weten we hoeveel steentjes in de daaraan voorafgaande rij zijn: het is een minder dan het aantal in de rij.

De onderdelen van de functiedefinitie

We starten met drie variabelen: het totaal aantal rijen in de driehoek, het aantal steentjes in een rij en het totaal aantal steentjes, wat we gaan uitrekenen. Deze variabelen noemen we resp. `aantal-rijen`, `aantal-steentjes-in-rij` en `totaal`.

Zowel `totaal` als `aantal-steentjes-in-rij` worden alleen binnen de functie gebruikt en worden gedeclareerd met `let`. De initiële waarde van `totaal` moet natuurlijk nul zijn. De initiële waarde van `aantal-steentjes-in-rij` moet gelijk zijn aan het aantal rijen in de driehoek, omdat de optelling met de langste rij begint.

Dit betekent dat we met een `let` expressie beginnen die er zo uit ziet:

```
(let ((totaal 0)
      (aantal-steentjes-in-rij aantal-rijen))
  body...)
```

Het totaal aantal steentjes kan worden gevonden door herhaaldelijk het aantal steentjes van een rij bij het totaal al gevonden op te tellen, dat wil zeggen herhaaldelijk de volgende expressie evalueren:

```
(setq totaal (+ totaal aantal-steentjes-in-rij))
```

Na het optellen van `aantal-steentjes-in-rij` bij `totaal`, het `aantal-steentjes-in-rij` moet met een verminderd worden, omdat de volgende keer dat de loop herhaalt, de voorafgaande rij opgeteld wordt bij het totaal.

Het aantal steentjes in een voorafgaande rij is een minder dan het aantal steentjes in een rij, waardoor de in Emacs Lisp ingebouwde functie `1-` kan worden gebruikt om het aantal steentjes in de voorafgaand rij te berekenen. Dit kan met de volgende expressie:

```
(setq aantal-steentjes-in-rij      (1- aantal-steentjes-in-rij))
```

Tenslotte weten we dat `while` loop de herhalende optellingen moet stoppen wanneer er geen steentjes in een rij zijn. Daarom is de test voor de `while` loop eenvoudig:

```
(while (> aantal-steentjes-in-rij 0)
```

De functiedefinitie samenstellen

We kunnen deze expressies samenvoegen om een functiedefinitie te creëren die werkt. Bij onderzoek blijkt echter dat een van de lokale variabelen niet nodig is!

De functiedefinitie ziet er zo uit:

```
;;; Eerste aflopende versie.
(defun driehoek (aantal-rijen)
  "Tel het aantal steentjes in een driehoek."
  (let ((totaal 0)
        (aantal-steentjes-in-rij aantal-rijen))
    (while (> aantal-steentjes-in-rij 0)
      (setq totaal (+ totaal aantal-steentjes-in-rij))
      (setq aantal-steentjes-in-rij
            (1- aantal-steentjes-in-rij)))
    totaal))
```

Zoals gezegd, deze functie werkt.

Maar we hebben `aantal-steentjes-in-rij` niet nodig.

Wanneer de functie `driehoek` wordt geëvalueerd wordt het symbool `aantal-rijen` gebonden aan een getal, om het een initiële waarde te geven. Dat getal kan in de body van de functie gewijzigd worden alsof het een lokale variabele is, zonder enige angst dat zo'n wijziging effect heeft op de waarde van de variabele buiten de functie. Dit is een erg zinvolle karakteristiek van Lisp, het betekent dat de variabele `aantal-rijen` overal in de functie kan worden gebruikt waar `aantal-steentjes-in-rij` is gebruikt.

Hier is de tweede versie van de functie, die netter geschreven is.

```
(defun driehoek (getal)                 ; Second version.
  "Geef som van de getallen 1 tot en met GETAL."
  (let ((totaal 0))
    (while (> getal 0)
      (setq totaal (+ totaal getal))
      (setq getal (1- getal)))
    totaal))
```

In het kort, een goed geschreven `while` loop bestaat uit drie delen:

1. Een test die onwaar teruggeeft nadat de loop het juiste aantal keer zichzelf herhaald heeft.
2. Een expressie waarvan de evaluatie de gewenste waarde teruggeeft nadat die herhaaldelijk geëvalueerd is.
3. Een expressie die de waarde wijzigt die aan de waar-of-onwaar-test wordt doorgegeven zodat de test onwaar teruggeeft nadat de loop zichzelf het juiste aantal keren heeft herhaald.

11.2 Bespaar je tijd: `dolist` en `dotimes`

Naast `while` bieden `dolist` en `dotimes` looping. Soms zijn ze sneller te schrijven dan een equivalente `while` loop. Beide zijn Lisp macros. (Zie Sectie "Macros" in *The GNU Emacs Lisp Reference Manual*.)

`dolist` werkt net als een `while` loop die door een lijst CDRt: `dolist` verkort elke keer dat het loopt automatisch de lijst—neemt de CDR van de lijst—en bindt de CAR van elke kortere versie van de lijst aan het eerste van zijn argumenten.

`dotimes` loopt een specifiek aantal keren: je specificeert hoeveel keer.

De `dolist` macro

Stel bijvoorbeeld dat je een lijst wilt omdraaien, zodat “eerste” “tweede” “derde” wordt “derde” “tweede” “eerste”.

In de praktijk gebruik je de functie `reverse` zoals hier:

```
(setq dieren '(gazelle giraf leeuw tijger))

(reverse dieren)
```

Hier is hoe je de lijst kan omdraaien met een `while` loop:

```
(setq dieren '(gazelle giraf leeuw tijger))

(defun reverse-lijst-met-while (lijst)
  "Met while, draai de volgorde van LIJST om."
  (let (waarde)  ; zorg dat de lijst leeg begint
    (while lijst
      (setq waarde (cons (car lijst) waarde))
      (setq lijst (cdr lijst)))
    waarde))

(reverse-lijst-met-while dieren)
```

En hier is hoe je het met een `dolist` macro kunt doen:

```
(setq dieren '(gazelle giraf leeuw tijger))

(defun reverse-lijst-met-dolist (lijst)
  "Met dolist, draai de volgorde van LIJST om."
  (let (waarde)  ; zorg dat de lijst leeg begint
    (dolist (element lijst waarde)
      (setq waarde (cons element waarde)))))

(reverse-lijst-met-dolist dieren)
```

In Info plaats je de cursor achter het haakje sluiten van elke expressie en je typt `C-x C-e`. In beide gevallen zou je het volgende moeten zien

```
(tijger leeuw giraf gazelle)
```

in het echogebied

Voor dit voorbeeld is bestaande functie `reverse` duidelijk het beste. De `while` loop is net als ons eerste voorbeeld (zie Sectie 11.1.1 "Een `while` loop en een lijst", pagina 117). De `while` checkt eerst of de lijst elementen heeft. Zo ja, dan construeert het een nieuwe lijst door het eerste element van de lijst aan de bestaande lijst toe te voegen (die in de eerste iteratie van de loop `nil` is.) Omdat het tweede element aan de voorkant het eerste element, en het derde element aan de voorkant van het tweede element wordt ingevoegd, wordt de lijst omgedraaid.

In de expressie met een `while` loop maakt de `(setq list (cdr list))` de lijst korter, waardoor de `loop` uiteindelijk stopt. Ook geeft die bij elke herhaling van de loop de `cons` expressie een nieuw eerste element en maakt het een nieuwe en kortere lijst.

De `dolist` expressie doet vrijwel hetzelfde als de `while` expressie, behalve dat de `dolist` macro wat werk voor je doet wanneer je een `while` expressie schrijft.

Net als een `while` loop, voert een `dolist` loopings uit. Het verschil is dat het automatisch de lijst korter maakt elke dat het de loop herhaalt—het CDR-t zelf door de lijst—en het bindt automatisch de CAR van elke kortere versie van de lijst aan zijn eerste argument.

In het voorbeeld wordt naar de CAR van elke kortere versie van de lijst verwezen met het symbool '`element`', de lijst zelf wordt '`lijst`' genoemd. De rest van de `dolist` expressie staat in de body.

De `dolist` expressie bindt de CAR van elke kortere versie van de lijst aan `element` en evalueert de body van de expressie, en herhaalt de loop. Het resultaat wordt teruggeven in `value`.

De `dotimes` macro

De `dotimes` macro is vergelijkbaar met `dolist`, behalve dat het een specifiek aantal keer de loop uitvoert.

Het eerste argument van `dotimes` krijgt rondom de loop elke keer de getallen 0, 1, 2 enzovoorts. Je moet de waarde van het tweede argument opgeven, het aantal keer dat de macro een loop moet doen.

Bijvoorbeeld et volgende bindt de getallen vanaf 0 tot, maar niet tot en met, het getal 3 aan het eerste argument, *getal*, en maakt dan een lijst van de drie getallen. (Het eerste getal is 0, het tweede getal is 1 en het derde getal is 2, dit zijn drie getallen in totaal, beginnend met 0 als het eerste getal.)

```
(let (waarde)      ; anders is waarde een void variabele
  (dotimes (getal 3)
    (setq waarde (cons getal waarde)))
  waarde)

⇒ (2 1 0)
```

De manier om `dotimes` te gebruiken is om een expressie een `aantal` aantal keer uit te voeren en dan het resultaat terug te geven, als een lijst of als een atoom.

Hier is een voorbeeld van een `defun` dat `dotimes` gebruikt om het aantal steetjes in een driehoek op te tellen.

```
(defun driehoek-met-dotimes (aantal-rijen)
  "Tel het aantal steentjes met `dotimes' in een driehoek op."
(let ((totaal 0))  ; anders is totaal een void variabele
  (dotimes (aantal aantal-rijen)
    (setq totaal (+ totaal (1+ aantal))))
  totaal))

(driehoek-met-dotimes 4)
```

11.3 Recursie

Een recursieve functie bevat code die de Lisp interpreter vertelt een programma aan te roepen dat exact hetzelfde als zichzelf loopt, maar met iets verschillende argumenten. De code loopt hetzelfde omdat het dezelfde naam heeft. Hoewel het programma dezelfde naam is, is het echter niet dezelfde entiteit. Het is verschillend. In het jargon heet het een andere “instantie”.

Als het programma correct geschreven is dan gaan de iets verschillende argumenten genoeg van het eerste argument verschillen dat de laatste instantie stopt.

11.3.1 Robots bouwen: de metafoor uitbreiden

Soms is het nuttig om een draaiend programma te zien als een robot die zijn werk doet. Tijdens het uitvoeren van het werk vraagt een recursieve functie een tweede robot om te helpen. De tweede robot is op elke manier identiek aan de eerste, behalve dat de tweede robot de eerste helpt en dat het andere argumenten krijgt dan de eerste.

In een recursieve functie kan een tweede robot een derde roepen, en een derde een vierde, enzovoort. Elk van deze is een aparte entiteit, maar het zijn allemaal klonen.

Omdat elke robot iets verschillende instructies heeft—de argumenten verschillen van de ene robot tot de volgende—moet de laatste robot weten wanneer te stoppen.

Laten we de metafoor van een computerprogamma als robot uitbreiden.

Een functiedefinitie bevat de blauwdruk voor de robot. Wanneer je een functiedefinitie installeert, dat wil zeggen, wanneer je een `defun` macro evalueert, in-

stalleer je de benodigde apparatuur om robots te bouwen. Het is net alsof je in een fabriek een assemblagelijn opzet. Robots met dezelfde naam worden volgens dezelfde blauwdruk gebouwd. Dus ze hebben hetzelfde modelnummer, maar een verschillend serienummer.

We zeggen vaak dat een recursieve functie "zichzelf aanroept". Wat we willen zeggen is dat de instructies in een recursieve functie de Lisp interpreter een andere functie laten aanroepen, die dezelfde naam heeft en hetzelfde werk doet als de eerste, maar met verschillende argumenten.

Het is belangrijk dat de argumenten tussen de ene instantie en de volgende verschillend zijn, anders zal het proces nooit stoppen.

11.3.2 De onderdelen van een recursieve definitie

Een recursieve functie bevat typisch een conditionele expressie die drie onderdelen heeft:

1. Een waar-of-onwaar-test die vaststelt of de functie opnieuw aangeroepen is, hier de *doe-opnieuw-test* genoemd.
2. De naam van de functie. Wanneer de naam wordt aangeroepen, wordt een nieuwe instantie van de functie—een nieuwe robot als het ware—gecreëerd en verteld wat te doen.
3. Een expressie die elke keer dat de functie word aangeroepen een andere waarde teruggeeft, hier de *volgende-stap-expressie* genoemd. Daardoor zal het aan de nieuwe instantie doorgeven argument (of argumenten) verschillen van wat aan de vorige instantie was doorgegeven. Dit zorgt dat de conditionele expressie, de *doe-opnieuw-test*, na het correcte aantal herhalingen naar onwaar testen.

Recursieve functies kunnen veel eenvoudiger zijn dan elk ander soort functie. Inderdaad, wanneer mensen ze beginnen te gebruiken, lijken ze vaak zo mysterieus eenvoudig dat ze onbegrijpelijk zijn Net als op een fiets rijden, vereist het lezen van een recursieve functie een bepaalde kundigheid die in het begin moeilijk is maar later eenvoudig lijkt.

Er zijn verschillende gebruikelijke recursieve patronen. Een erg eenvoudig patroon ziet er zo uit:

```
(defun naam-van-recursieve-functie (argument-list)
  "documentatie..."
  (if doe-opnieuw-test
    body...
    (naam-van-recursieve-functie
         volgende-stap-expressie)))
```

Elke keer dat een recursieve functie wordt geëvalueerd worden een nieuwe instantie er van gecreëerd en verteld wat te doen. De argumenten vertellen de instantie wat te doen.

Een argument wordt gebonden aan de waarde van de volgende-stap-expressie. Elke instantie draait met een andere waarde van de volgende-stap-expressie.

De waarde van de volgende-stap-expressie wordt gebruik in de doe-opnieuw-test.

De waarde die de volgende-stap-expressie teruggeeft wordt doorgegeven aan de nieuwe instantie van de functie, die het evalueert (of een transformatie er van) om

vast te stellen of het moet doorgaan of stoppen. De volgende-stap-expressie is zo ontworpen dat de doe-opnieuw-test onwaar teruggeeft wanneer de functie niet langer meer moet worden herhaald.

De doe-opnieuw-test wordt soms de *stop conditie* genoemd, omdat het de herhalingen stopt wanneer het als onwaar test.

11.3.3 Recursie met een lijst

Het voorbeeld van een `while` loop dat de elementen van een lijst met getallen kan recursief geschreven worden. Hier is de code inclusief een expressie die de waarde van de variabele `dieren` naar een lijst zet.

Wanneer je dit in Info in Emacs leest, kan je expressie direct in Info evalueren. Zo dan, dan moet je het voorbeeld kopiëren in het `*scratch*` buffer en het daar evalueren. Gebruik `C-u C-x C-e` om de `(print-elementen-recursief dieren)` te evalueren zodat de resultaten in het buffer getoond worden, zo niet dan probeert de Lisp interpreter de resultaten in een enkele regel in het echogebied samen te knijpen.

Plaats de cursor direct achter het laatste haakje sluiten van de functie `print-elementen-recursief`, voor het commentaar. Anders probeert de Lisp interpreter het commentaar te evalueren.

```
(setq dieren '(gazelle giraf leeuw tijger))

(defun print-elementen-recursief (lijst)
  "Toon elk element van LIJST op een eigen regel.
Gebruikt recursie."
  (when lijst                              ; doe-opnieuw-test
        (print (car lijst))                ; body
        (print-elementen-recursief         ; recursieve aanroep
         (cdr lijst))))                    ; volgende-stap-expressie

(print-elementen-recursief dieren)
```

De functie `print-elementen-recursief` test eerst of er inhoud in de lijst is. Wanneer dat zo is, toont de functie het eerste element van de lijst, de CAR van de lijst. Vervolgens roept de functie zichzelf aan maar geeft zichzelf niet de hele lijst maar het tweede en volgende elementen van de lijst, de CDR van de lijst.

Anders gezegd, wanneer de lijst niet leeg is, start de functie een nieuwe instantie van code die vergelijkbaar is met de initiële code, maar is een andere uitvoeringsthread, met andere argumenten dan de eerste instantie.

Weer anders gezegd, wanneer de lijst niet leeg is, bouwt de eerste robot een tweede robot en zegt die wat de doen. De tweede robot is een ander individu dan de eerste, maar is het zelfde model.

Wanneer de tweede evaluatie plaatsvindt wordt de `when` expressie geëvalueerd en als die waar is toont die het eerste element van de lijst die het als zijn argument had gekregen (die het tweede element is van de originele lijst is). Hierna roept de functie zichzelf aan met de CDR van de lijst waarmee het aangeroepen is wat (de tweede ronde) de CDR van de CDR van de originele lijst is.

Merk op dat hoewel we zeggen dat de functie "zichzelf aanroept" we bedoelen dat de Lisp interpreter een nieuwe instantie van het programma maakt en instrueert. De nieuwe instantie is een kloon van de eerste, maar is een separaat individu.

Elke keer dat de functie zichzelf aanroept doet het dat met een kortere versie van de originele lijst. Het creëert een nieuwe instantie die werkt met een kortere lijst.

Uiteindelijk roept de functie zichzelf aan met een lege lijst. Dit creëert een nieuwe instantie wiens argument `nil` is. De conditionele expressie test de waarde van `lijst`. Omdat de waarde van `nil` `nil` is, test de `when` expressie onwaar en dus wordt het dan-deel niet geëvalueerd. De functie als geheel geeft `nil` terug.

Wanneer je de expressie `(print-elementen-recursief dieren)` in het `*scratch*` buffer evalueert, zie je dit resultaat:

```
gazelle

giraf

leeuw

tijger
nil
```

11.3.4 Recursie in plaats van een teller

De in een eerdere beschreven functie `driehoek` kan ook recursief worden geschreven. Dit ziet er zo uit:

```
(defun driehoek-recursief (aantal)
  "Geef de som van de getallen 1 tot en met AANTAL terug.
Met recursie."
  (if (= aantal 1)                    ; doe-opnieuw-test
      1                               ; dan-deel
    (+ aantal                         ; anders-deel
       (driehoek-recursief            ; recursieve aanroep
        (1- aantal)))))               ; volgende-stap-expressie

(driehoek-recursief 7)
```

Je kunt deze functie installeren door het te evalueren en dan uitproberen door `(driehoek-recursief 7)` te evalueren. (Denk er aan de cursor direct achter het laatste haakje sluiten van de functiedefinitie te plaatsen, voor het commentaar.)

Om te begrijpen hoe deze functie werkt, beschouwen we wat gebeurt in de verschillende gevallen wanneer de functie 1, 2, 3 of 4 als waarde van zijn argument krijgt doorgegeven.

Ten eerste wat gebeurt als de waarde van het argument 1 is?

De functie heef een `if` expressie na de documentatiestring. Die test of de waarde van `aantal` gelijk is aan 1. Wanneer dat zo is, evalueert Emacs het dan-deel van de `if` exprissie, die 1 teruggeeft als waarde van de functie. (Een driehoek met één rij bevat één steentje.)

Stel echter dat de waarde van het argument 2 is. In dat geval evalueert Emacs het anders-deel van de `if` expressie.

Het anders-deel bestaat uit een optelling, de recursieve aanroep van `driehoek-recursief` en een aflopende actie, en ziet er zo uit:

```
(+ aantal (driehoek-recursief (1- aantal)))
```

Wanneer Emacs deze expressie evalueert, wordt de binnenste expressie het eerste geëvalueerd, daarna de andere delen opvolgend. Hier zijn de stappen in detail:

Stap 1 *Evalueer de binnenste expressie*
De binnenste expressie is `1- aantal`, dus Emacs verlaagt de waarde van `aantal` van 2 naar 1.

Stap 2 *Evalueer de functie* `driehoek-recursief`.
De Lisp interpreter creëert een individuele instantie van `driehoek-recursief`. Het maakt niet uit dat deze functie binnen zichzelf staat. Emacs geeft het resultaat Stap 1 als argument gebruikt door deze instantie van de functie `driehoek-recursief`.

In dit geval evalueert Emacs `driehoek-recursief` met een argument 1. Dit betekent dat deze evaluatie van `driehoek-recursief` 1 teruggeeft.

Stap 3 *Evalueer de waarde van* `aantal`.
De variabele `aantal` is het tweede element van de lijst die start met `+`. De waarde is 2.

Stap 4 *Evalueer de* `+` *expressie.*
De `+` expressie ontvangt twee argumenten, de eerste van het evalueren van `aantal` (Stap 3) en het tweede van de evaluatie van `driehoek-recursief` (Stap 2).

Het resultaat van de optelling is de som van 2 plus 1 en het getal 3 wordt teruggegeven. dit is correct. Een driehoek met twee rijen bevat drie steentjes.

Een argument van 3 of 4

Stel dat `driehoek-recursief` wordt aangeroepen met een argument van 3.

Stap 1 *Evaluaeer de doe-opnieuw-test.*
De `if` expressie wordt eerst geëvalueerd. Dit is de doe-opnieuw-test en geeft onwaar terug, dus het anders-deel van de `if` expressie wordt geëvalueerd. (Merk op dat in dit voorbeeld de doe-opnieuw-test zorgt dat de functie zichzelf aanroept wanneer die onwaar test, en niet wanneer die waar test.)

Stap 2 *Evalueer de binnenste expressie van het anders-deel.*
De binnenste expressie van het anders-deel wordt geëvalueerd, die 3 verlaagt naar 2. Dit is de volgende-stap-expressie.

Stap 3 *Evalueer de functie* `driehoek-recursief`.
Het getal 2 is doorgegeven aan de functie `driehoek-recursief`.

We weten al wat gebeurt wanneer Emacs de `driehoek-recursief` evalueert met een argument van 2. Nadat het door de eerder

beschreven reeks acties heen is gegaan, geeft het een waarde 3 terug. Dus dat is wat hier gebeurt.

Stap 4 *Evalueer de optelling.*
3 wordt doorgegeven als een argument van de optelling en wordt opgeteld bij het getal waarmee de functie was aangeroepen, dat 3 is.

De waarde teruggeven door de functie als geheel is 6.

Nu we weten wat gebeurt wanneer `driehoek-recursief` wordt aangeroepen met een argument van 3, ligt het voor de hand wat gebeurt als het wordt aangeroepen met een argument van 4.

In de recursieve aanroep, geeft de evaluatie van
```
(driehoek-recursief (1- 4)
```
de waarde van het evalueren van
```
(driehoek-recursief 3)
```
wat 6 is en deze waarde wordt opgeteld bij 4 door de optelling in de derde regel.

De waarde teruggeven door de functie als geheel is 10.

Elke keer dat `driehoek-recursief` wordt geëvalueerd, evalueert het een versie van zichzelf—een andere instantie van zichzelf—met een kleiner argument, totdat het argument klein genoeg is en het niet zichzelf evalueert.

Merk op dat dit specifieke ontwerp voor een recursieve functie vereist dat operaties kunnen worden uitgesteld.

Voordat `(driehoek-recursief 7)` een antwoord kan berekenen, moet het `(driehoek-recursief 6)` aanroepen, en voordat `(driehoek-recursief 6)` een antwoord kan berekenen, moet het `(driehoek-recursief 5)` aanroepen, enzovoorts. Met andere woorden, de berekening die `(driehoek-recursief 7)` maakt moet worden uitgesteld totdat `(driehoek-recursief 6)` een berekening gemaakt heeft, en `(driehoek-recursief 6)` moet worden uitgesteld totdat `(driehoek-recursief 5)` klaar is, enzovoorts.

Wanneer al deze instanties van `driehoek-recursief` beschouwd worden als verschillende robots, dan moet de eerste robot wachten tot de tweede robot zijn werk gedaan heeft, en die moet weer wachten tot de derde robot klaar is, enzovoorts.

Er is een manier om dit soort uitstel te omzeilen, die we bespreken in Sectie 11.3.7 “Recursie zonder uitstel”, pagina 138.

11.3.5 Recursie voorbeeld met `cond`

De eerder beschreven versie van `driehoek-recursief` is geschreven met de speciale vorm `if`. Het kan ook worden geschreven met een andere speciale vorm met de naam `cond`. De naam van de speciale vorm `cond` is een afkorting van het woord `conditioneel`.

Alhoewel de speciale vorm `cond` niet zo vaak in de Emacs Lisp broncode als `if`, wordt het vaak genoeg gebruikt om het uit te leggen.

Het sjabloon voor een `cond` expressie ziet er zo uit:

```
(cond
 body...)
```

waar de *body* een reeks lijsten is.

Meer volledig ziet het sjabloon er zo uit:

```
(cond
 (eerste-waar-of-onwaar-test eerste-consequentie)
 (tweede-waar-of-onwaar-test tweede-consequentie)
 (derde-waar-of-onwaar-test derde-consequentie)
  ...)
```

Wanneer de Lisp interpreter de `cond` expressie evalueert, evalueert die het eerste element (de CAR van de waar-of-onwaar-test) van de expressie in een reeks van expressies binnen de body van de `cond`.

Wanneer de waar-of-onwaar-test `nil` teruggeeft, wordt de rest van de expressie, de consequentie, overgeslagen en wordt de volgende expressie geëvalueerd. Wanneer een expressie gevonden wordt wiens waar-of-onwaar-test een waarde ongelijk aan `nil` teruggeeft, wordt de consequentie van die expressie geëvalueerd. De consequentie kan een of meer expressies zijn. Wanneer de consequentie meer dan een expressie bevat worden de expressie achtereenvolgens geëvalueerd en de waarde van de laatste wordt teruggegeven. Wanneer de expressie geen consequentie heeft, dan wordt de waarde van de waar-of-onwaar-test teruggegeven.

Wanneer geen van de waar-of-onwaar-tests waar test, dan geeft de `cond` expressie `nil` terug.

De functie `driehoek` met een `cond` ziet er zo uit:

```
(defun driehoek-met-cond (getal)
  (cond ((<= getal 0) 0)
        ((= getal 1) 1)
        ((> getal 1)
         (+ getal (driehoek-met-cond (1- getal))))))
```

In dit voorbeeld geeft de `cond` 0 terug wanneer het getal kleiner of gelijk aan 0 is, het geeft 1 terug wanneer het getal 1 is, en het evalueert `(+ getal (driehoek-met-cond (1- getal)))` wanneer het getal groter dan 1 is.

11.3.6 Recursieve patronen

Here are three common recursive patterns. Each involves a list. Recursion does not need to involve lists, but Lisp is designed for lists and this provides a sense of its primal capabilities.

Recursief patroon: *elke*

In het `elke` recursieve patroon wordt een actie uitgevoerd op elk element van de lijst.

Het basispatroon is:

- Als een lijst leeg is, geef `nil` terug.
- Else, act on the beginning of the list (the CAR of the list)
  - through a recursive call by the function on the rest (the CDR) of the list,
  - and, optionally, combine the acted-on element, using `cons`, with the results of acting on the rest.

Hier is een voorbeeld:

```
(defun kwadrateer-elke (getallen-lijst)
  "Kwadrateer elk van een GETALLEN Lijst, recursief."
  (if (not getallen-lijst)                ; doe-opnieuw-test
      nil
    (cons
     (* (car getallen-lijst) (car getallen-lijst))
     (kwadrateer-elke (cdr getallen-lijst))))) ; volgende-stap-expressie

(kwadrateer-elke '(1 2 3))
    ⇒ (1 4 9)
```

Wanneer `getallen-lijst` leeg is, doe niets. Maar als die inhoud heeft, construeer een lijst door het kwadraat van het eerste getal te combineren met een lijst met het resultaat van de recursieve aanroep.

(Het voorbeeld volgt exact dit patroon: `nil` wordt teruggeven wanneer de getallen-lijst leeg is. In de praktijk schrijf je conditional zo dat het de actie uitvoert wanneer de lijst met getallen niet leeg is.)

De functie `print-elementen-recursief` (zie Sectie 11.3.3 “Recursie met een lijst”, pagina 131) is een ander voorbeeld van een `elke` patroon, behalve vat in dit geval, in plaats van het samenbrengen van de resultaten met `cons`, we elk output-element tonen.

De functie `print-elementen-recursief` ziet er zo uit:

```
(setq dieren '(gazelle giraf leeuw tijger))

(defun print-elementen-recursief (lijst)
  "Toon elk element van LIJST op een eigen regel.
Gebruikt recursie."
  (when lijst                           ; doe-opnieuw-test
        (print (car lijst))             ; body
        (print-elementen-recursief      ; recursieve aanroep
         (cdr lijst))))                 ; volgende-stap-expressie

(print-elementen-recursief dieren)
```

Het patroon voor `print-elementen-recursief` is:

- Wanneer de lijst leeg is, doe niets.
- But when the list has at least one element,
  - act on the beginning of the list (the CAR of the list),
  - and make a recursive call on the rest (the CDR) of the list.

Recursief patroon: *accumuleren*

Een ander recursief patroon heet het `accumuleren` patroon. In het `accumuleren` recursieve patroon wordt een actie uitgevoerd op elk element van een lijst en het resultaat van die actie wordt geaccumuleerd met de resultaten van het uitvoeren van die actie op de andere elementen.

Dit lijkt erg op het `elke` patroon met `cons`, behalve dat `cons` niet wordt gebruikt, maar een andere combinerende functie.

Het patroon is:

- Als een lijst leeg is, geef nul terug of een andere constante.
- Else, act on the beginning of the list (the CAR of the list),
  - and combine that acted-on element, using `+` or some other combining function, with
  - a recursive call by the function on the rest (the CDR) of the list.

Hier is een voorbeeld:

```
(defun sommeer-elementen (getallen-lijst)
  "Tel de elementen van GETALLEN-LIJST bij elkaar op."
  (if (not getallen-lijst)
      0
    (+ (car getallen-lijst) (sommeer-elementen (cdr getallen-lijst)))))

(sommeer-elementen '(1 2 3 4))
    ⇒ 10
```

Zie Sectie 14.9.2 “Een lijst met bestanden maken”, pagina 183, voor een uitleg van het accumuleren patroon.

Recursieve patronen: *bewaren*

Een derde recursief patroon het het `bewaren` patroon. In het `bewaren` recursieve patroon wordt elk element van de lijst getest, het element wordt behandeld en de resultaten worden alleen bewaard wanneer het element aan een criterium voldoet.

Opnieuw lijkt dit erg op het `elke` patroon, behalve dat elementen worden overgeslagen tenzij ze aan een criterium voldoen.

Het patroon heeft drie elementen:

- Als een lijst leeg is, geef `nil` terug.
- Else, if the beginning of the list (the CAR of the list) passes a test
  - act on that element and combine it, using `cons` with
  - a recursive call by the function on the rest (the CDR) of the list.
- Otherwise, if the beginning of the list (the CAR of the list) fails the test
  - skip on that element,
  - and, recursively call the function on the rest (the CDR) of the list.

Hier is een voorbeeld met `cond`:

```
(defun bewaar-drie-letter-woorden (woorden-lijst)
  "Bewaar drie-letter woorden in WOORDEN-LIJST."
  (cond
   ;; Eerst doe-opnieuw-test: stop-conditie
   ((not woorden-lijst) nil)

   ;; Tweede doe-opnieuw-test: wanneer te behandelen
   ((eq 3 (length (symbol-name (car woorden-lijst))))
    ;; combineer behandelde elementen met recursieve aanroep op kortere lijst
    (cons (car woorden-lijst) (bewaar-drie-letter-woorden (cdr woorden-lijst))))

   ;; Derde doe-opnieuw-test: wanneer element over te slaan;
   ;;   recursief aanroepen kortere lijst met volgende-stap-expressie
   (t (bewaar-drie-letter-woorden (cdr woorden-lijst)))))

(bewaar-drie-letter-woorden '(een twee drie vier vijf zes))
    ⇒ (een zes)
```

Het spreekt voor zich dat je `nil` als test voor wanneer te stoppen niet hoeft te gebruiken, je kunt natuurlijk deze patronen combineren.

11.3.7 Recursie zonder uitstel

Laten we nog een keer de functie `driehoek-recursief` beschouwen. We zullen zien dat de tussenliggende berekeningen worden uitgesteld tot ze allemaal uitgevoerd kunnen worden.

Hier is de functiedefinitie:

```
(defun driehoek-recursief (aantal)
  "Geef de som van de getallen 1 tot en met AANTAL terug.
Met recursie."
  (if (= aantal 1)                    ; doe-opnieuw-test
      1                               ; dan-deel
    (+ aantal                         ; anders-deel
       (driehoek-recursief            ; recursieve aanroep
        (1- aantal)))))               ; volgende-stap-expressie
```

Wat gebeurt er wanneer de deze functie met een argument van 7 aanroepen?

De eerste instantie van de functie `driehoek-recursief` telt het getal 7 bij de waarde die de tweede instantie van `driehoek-recursief` teruggeeft, een instantie

de een argument van 6 krijgt doorgestuurd. Dat wil zeggen dat de eerste berekening is:

```
(+ 7 (driehoek-recursief 6))
```

De eerste instantie van `driehoek-recursief`—je kunt het zien als een kleine robot—kan zijn werk niet afmaken. Het moet de berekening voor `(driehoek-recursief 6)` overdragen aan een tweede instantie van het programma, aan een tweede robot. De tweede individu is compleet verschillend van het eerste, het is, in het jargon, een "andere instantie". Of, anders gezegd, een andere robot. Die is hetzelfde model als de eerste, maar met een verschillende serienummer.

En wat geeft `(driehoek-recursief 6)` terug? Het geeft het getal 6 opgeteld bij de waarde teruggeven door het evalueren van `driehoek-recursief` met een argument van 5. Met het robot metafoor, het vraagt een andere robot om hulp.

Nu is het totaal:

```
(+ 7 6 (driehoek-recursief 5))
```

En wat gebeurt hierna?

```
(+ 7 6 5 (driehoek-recursief 4))
```

Elke keer dat `driehoek-recursief` wordt aangeroepen, behalve de laatste keer, maakt het een nieuwe instantie van het programma—een andere robot—en vraagt die een berekening te maken.

Uiteindelijk is de volledige berekening opgezet en uitgevoerd:

```
(+ 7 6 5 4 3 2 1)
```

Dit ontwerp van de functie stelt de berekening van eerste uit tot de tweede kan worden gedaan, dan stelt die uit tot de derde kan worden gedaan, enzovoorts. Elk uitstel betekent dat de computer moet onthouden waarop gewacht wordt. Dat is geen probleem wanneer er maar een stappen zijn, zoals in dit voorbeeld. Maar het kan een probleem worden wanneer er meer stappen zijn.

11.3.8 Zonder uitstel oplossing

De oplossing van het probleem van de uitgestelde operaties is het op een manier te schrijven die de operaties niet uitstelt[2]. Dit vereist het schrijven volgens een ander patroon, een die vaak het schrijven van twee functiedefinities inhoudt, een initialisatiefunctie en een hulpfunctie.

De initialisatiefunctie zet de klus op, de hulpfunctie doet het werk.

[2] De uitdrukking *tail-recursie* wordt gebruikt om zo'n proces te beschrijven, een die constante ruimte gebruikt.

Hier zijn twee functiedefinities voor het optellen van getallen. Zij zijn zo eenvoudig dat ik moeilijk vind ze te begrijpen.

```
(defun driehoek-initialisatie (aantal)
  "Geef de som van de getallen 1 tot en met AANTAL.
Dit is the initialisatiecomponent van een duo van twee functies
met recursie ."
  (driehoek-recursieve-helper 0 0 aantal))
(defun driehoek-recursieve-helper (som teller aantal)
  "Geef SOM terug, met TELLER, tot en met AANTAL.
Dit is de hulpcomponent van een duo van twee functies
met recursie."
  (if (> teller aantal)
      som
    (driehoek-recursieve-helper (+ som teller) ; som
                                (1+ teller)    ; teller
                                aantal)))      ; aantal
```

Installeer beide functiedefinities door ze te evalueren en roep dan `driehoek-initialisatie` met twee rijen aan:

```
(driehoek-initialisatie 2)
    ⇒ 3
```

De initialisatiefunctie roept de eerste instantie van de hulpfunctie met drie argumenten aan: nul, nul, een getal dat het aantal rijen van de driehoek is.

De eerste twee aan de hulpfunctie doorgegeven argumenten zijn initialisatiewaarden. Deze waarden worden aangepast wanneer `driehoek-recursieve-helper` nieuwe instanties aanroept.[3]

Laten we kijken wat gebeurt wanneer we een driehoek met een rij hebben. (Deze driehoek bevat één steentje!)

`driehoek-initialisatie` roept zijn hulpfunctie aan met de argumenten `0 0 1`. Die functie voert de conditionele test uit of `(> teller aantal)`:

```
(> 0 1)
```

een ziet dat het resultaat onwaar is, en dus roept deze het anders-deel van de `if` clausule aan:

```
(driehoek-recursieve-helper
 (+ som teller)  ; som plus teller ⇒ som
 (1+ teller)     ; verhoog teller ⇒ teller
 aantal)         ; aantal blijft hetzelfde
```

[3] Het jargon is lichtelijk verwarrend: `driehoek-recursieve-helper` gebruikt een proces dat iteratief is in een procedure die recursief is. Dit proces het iteratief omdat de computer maar die waarden hoeft bij te houden: `som`, `teller` en `aantal`. De procedure is recursief omdat de functie zichzelf aanroept. Anderzijds worden beide processen en de door `driehoek-recursief` gebruikte procedure recursief genoemd. Het woord "recursief" heeft verschillende betekenissen in de twee contexten.

wat eerst uitrekent:

```
(driehoek-recursieve-helper (+ 0 0)  ; som
                            (1+ 0)   ; teller
                            1)       ; aantal
```

wat is:

```
driehoek-recursieve-helper 0 1 1
```

Opnieuw wordt `(> teller aantal)` onwaar, en opnieuw gaat de Lisp interpreter de `driehoek-recursieve-helper` evalueren, en maakt een nieuwe instantie met nieuwe argumenten.

Deze nieuwe instantie is:

```
(driehoek-recursieve-helper
 (+ som teller)  ; som plus teller ⇒ som
 (1+ teller)     ; verhoog teller ⇒ teller
 aantal)         ; aantal blijft hetzelfde
```

wat is:

```
(driehoek-recursieve-helper 1 2 1)
```

In dit geval test de `(> teller aantal)` op waar! Dus geeft de instantie de waarde van de som terug, die 1 is, zoals verwacht.

Laten we nu `driehoek-initialisatie` een argument van 2 geven, om te ontdekken hoeveel steentjes een driehoek met twee rijen bevat.

Die functie roept `(driehoek-recursieve-helper 0 0 2)` aan.

In stappen, zijn de aangeroepen instanties:

```
(driehoek-recursieve-helper 0    1       2)

(driehoek-recursieve-helper 1    2       2)

(driehoek-recursieve-helper 3    3       2)
```

Wanneer de laatste instantie wordt aangeroepen, test de de `(> teller aantal)` op waar, dus de instantie geeft de waarde van `som` terug, die 3 is.

Dit soort patronen helpt wanneer je functies schrijft die veel resources in de computer kunnen gebruiken.

11.4 Looping oefening

- Schrijf een functie vergelijkbaar met `driehoek` waar elke rij een waarde heeft die het kwadraat is van het rijnummer. Gebruik een `while` loop.
- Schrijf een functie vergelijkbaar met `driehoek` die de waarden vermenigvuldigt in plaats van optelt.
- Herschrijf deze twee functies recursief. Herschrijf ze met gebruik van `cond`.
- Schrijf een functie voor Texinfo mode die een index-entry creëert aan het begin van een paragraaf voor elke '`@dfn`' in die paragraaf. (In een Texinfo bestand, markeert '`@dfn`' een definitie. Dit boek is geschreven in Texinfo.)

  Veel van de functies die je nodig hebt zijn beschreven in twee voorgaande hoofdstukken, Hoofdstuk 8 "Knippen en opslaan van tekst", pagina 86, en Hoofd-

stuk 10 “Tekst terug yanken”, pagina 113. Wanneer je `forward-paragraph` gebruikt om de index-entry aan het begin van de paragraaf te plaatsten, kun je met *`C-h f`* (`describe-function`) ontdekken hoe je het commando achterwaarts laat werken.

Voor meer informatie, zie “Indicating Definitions, Commands, etc.” in *Texinfo, The GNU Documentation Format*.

12 Reguliere expressie zoekopdrachten

Zoekopdrachten met reguliere expressies worden uitgebreid gebruikt in Emacs. De twee functies `forward-sentence` en `forward-paragraph` illustreren deze zoekopdrachten goed. Zij gebruiken reguliere expressies om uit te zoeken waarheen de point te verplaatsen. De uitdrukking "reguliere expressie" wordt vaak geschreven als "regexp".

Reguliere expressie zoekopdrachten zijn beschreven in Sectie "Regular Expression Search" in *The GNU Emacs Manual*, en ook in Sectie "Regular Expressions" in *The GNU Emacs Lisp Reference Manual*. Tijdens het schrijven van dit hoofdstuk ga ik er vanuit dat je tenminste een beetje met ze kennis gemaakt hebt. Het belangrijkste punt om te onthouden is dat reguliere expressies je in staat stellen naar patronen te zoeken en ook naar letterlijke karakterstrings. De code in `forward-sentence` zoekt bijvoorbeeld naar patronen van mogelijke karakters die het einde van een zin kunnen markeren, en verplaatst point naar die plek.

Voordat we naar de code van de functie `forward-sentence` kijken, is het waardevol te overwegen wat het patroon is dat het einde van een zin moet zijn. Het patroon wordt in de volgende sectie besproken, daarop volgt een beschrijving van de reguliere expressie zoekfunctie, `re-search-forward`. De functie `forward-sentence` wordt in de daaropvolgende sectie besproken. Tenslotte wordt de functie `forward-paragraph` in de laatste sectie van dit hoofdstuk beschreven. `forward-paragraph` is een complexe functie die verschillende nieuwe eigenschappen introduceert.

12.1 De reguliere expressie for `sentence-end`

Het symbool `sentence-end` is gebonden aan het patroon dat het einde van een zin markeert. Wat moet deze reguliere expressie zijn?

Het is duidelijk dat een zin kan worden beëindigd met een punt, een vraagteken of een uitroepteken. In het Engels worden inderdaad alleen clausules die met een van deze drie karakters eindigen beschouwd als het eind van een zin. Dit betekent dat het patroon de volgende karakterset moet bevatten:

```
[.?!]
```

We willen echter niet dat `forward-sentence` gewoon naar een punt, vraagteken of uitroepteken springt, omdat zo'n karakter in het midden van een zin kan worden gebruikt. Een punt wordt bijvoorbeeld gebruikt na een afkorting. Meer informatie is dus benodigd.

Volgens de conventie typ je in het Engels twee spaties achter elke zin, maar alleen één spatie achter een punt, vraagteken of uitroepteken in de body van een zin. Een een punt, vraagteken of uitroepteken gevolgd door twee spaties is daarom een goede indicator van het einde van een zin. In een bestand echter, kan een tab of het einde van regel in plaats van de twee spaties staan. Dit betekent dat de reguliere expressie deze drie alternatieven moet bevatten.

Deze groep van alternatieven zijn er zo uit:

```
\\($\\| \\|  \\)
       ^   ^^
     TAB  SPC
```

Hier geeft '`$`' het einde van de regel aan, en ik heb aangegeven waar de tab en de twee spaties zijn ingevoegd in de expressie. Beide zijn ingevoegd door de betreffende letters in de expressie op te nemen.

Twee backslashes, '`\\`' zijn vereist voor de haakjes en het vertikale streepje (vertical bar): de eerste backslash quote hde volgende backslash in Emacs, en de tweede geeft aan dat het volgende karakter, het haakje of het vertikale streepje, speciaal is.

Een zin kan ook door een of meer carriage returns gevolgd worden, zoals dit:

```
[
]*
```

Zoals tabs en spaties wordt een carriage return ingevoegd in een reguliere expressie door het letterlijk in te voegen. De ster geeft aan dat de `RET` nul of meer keer herhaald kan worden.

Maar het einde van een zin bestaat niet alleen uit een punt, vraagteken of uitroepteken gevolgd door passende ruimte: een aanhalingsteken sluiten, of een soort sluitend haakje kan aan de spatie voorafgaan. Een of meer van deze tekens of haakjes kan aan de spatie voorafgaan. Dit vereist een expressie die er zo uitziet:

```
[]\"')}]*
```

De eerste '`]`' is het eerste karakter in deze expressie, het tweede karakter is '`"`', die voorafgegaan wordt door een '`\`' om Emacs te vertellen dat de '`"`' *niet* speciaal is. De laatste drie karakters zijn '`'`', '`)`' en '`}`'.

Dit alles suggereert ons wat de reguliere expressie voor het markeren van het eind van een zin zou moeten zijn. En wanneer we `sentence-end` evalueren, zien we dat het inderdaad de volgende waarde teruggeeft:

```
sentence-end
     ⇒ "[.?!][]\"')}]*\\($\\|     \\|  \\)[
]*"
```

(Nou, niet in GNU Emacs 22. Dat is omdat gestreefd is het proces simpeler te maken en meer bijzondere karakters en talen te hanteren. Wanneer de waarde van `sentence-end` `nil` is, gebruik dan de waarde gedefinieerd door de functie `sentence-end`. (Hier is een gebruik van het verschil tussen een waarde en een functie in Emacs Lisp.) De functie geeft de waarde terug die geconstrueerd is met de variabelen `sentence-end-base`, `sentence-end-double-space`, `sentence-end-without-period`, and `sentence-end-without-space`. De kritieke variabele `sentence-end-base`. De globale waarde daarvan lijkt op die hierboven beschreven maar bevat twee extra aanhalingstekens. Deze zijn op verschillende manieren gebogen. Wanneer de variabele `sentence-end-without-period` waar is, dan vertelt dat Emacs dat een zin mag eindigen zonder punt, zoals Taise tekst.

12.2 De functie `re-search-forward`

De functie `re-search-forward` lijkt erg op de functie `search-forward`. (Zie Sectie 8.1.3 "De functie `search-forward`", pagina 88.)

`re-search-forward` zoekt naar een reguliere expressie. Wanneer de zoekopdracht geslaagd is, zet het point onmiddellijk achter het laatste karakter in het doel. Wanneer de zoekopdracht achterwaarts is, dan zet het point direct voor het eerste karakter in het doel. Je kunt `re-search-forward` zeggen dat het `t` voor waar moet teruggeven. (Verplaatsen van point is daarom een zij-effect.)

Net als `search-forward` heeft de functie `re-search-forward` vier argumenten.

1. Het eerste argument is de reguliere expressie waar de functie naar zoekt. De reguliere expressie is een string tussen aanhalingstekens.
2. Het optionele tweede argument begrenst tot hoever de functie zoekt, het is een grens, die gespecificeerd wordt als een positie in het buffer.
3. Het optionele derde argument specificeert hoe de functie reageert op falen: `nil` als derde argument laat de functie een fout signaleren (en een boodschap tonen) wanneer de zoekopdracht faalt, een andere waarde zorgt dat die `nil` teruggeeft wanneer de zoekopdracht faalt en `t` wanneer de zoekopdracht slaagt.
4. Het optionele vierde argument is de aantal-herhalingen. Een negatieve aantal-herhalingen laat `re-search-forward` achterwaarts zoeken.

Het slabloon voor `re-search-forward` ziet er zo uit:

```
(re-search-forward "reguliere-expressie"
                   zoeklimiet
                   wat-te-doen-wanneer-zoeken-faalt
                   aantal-herhalingen)
```

Het tweede, derde en vierde argument zijn optioneel. Als je echter een waarde naar een of beide van de laatste twee argumenten wilt doorgeven, moet je ook een waarde aan alle voorgaande argumenten doorgeven. Anders weet de Lisp interpreter niet aan welk argument je een waarde doorgeeft.

In de functie `forward-sentence` is de reguliere expressie de waarde van de variabele `sentence-end`. In een simpele vorm is die:

```
"[.?!][]\"')}]*\\($\\|  \\|  \\)[
]*"
```

De limiet van de zoekopdracht is het einde van de paragraaf (omdat een zin niet voorbij een paragraaf kan gaan). Wanneer de zoekopdracht faalt, geeft de functie `nil` terug en het aantal herhalingen wordt gegeven door het argument van de functie `forward-sentence`.

12.3 `forward-sentence`

Het commando om de cursor een zin vooruit te verplaatsen is een eenvoudig voorbeeld van hoe zoekopdrachten met reguliere expressies in Emacs Lisp te gebruiken. De functie ziet er langer en ingewikkelder uit dan die is, dit is omdat de functie gemaakt is om zowel achterwaarts als voorwaarts te gaan en, optioneel, meer dan één zin. De functie is over het algemeen gebonden aan het toetscommando *`M-e`*.

Hier is de code van `forward-sentence`:

```
(defun forward-sentence (&optional arg)
  "Move forward to next end of sentence.  With argument, repeat.
With negative argument, move backward repeatedly to start of sentence.

The variable `sentence-end' is a regular expression that matches ends of
sentences.  Also, every paragraph boundary terminates sentences as well."
  (interactive "p")
  (or arg (setq arg 1))
  (let ((opoint (point))
        (sentence-end (sentence-end)))
    (while (< arg 0)
      (let ((pos (point))
            (par-beg (save-excursion (start-of-paragraph-text) (point))))
       (if (and (re-search-backward sentence-end par-beg t)
                (or (< (match-end 0) pos)
                    (re-search-backward sentence-end par-beg t)))
           (goto-char (match-end 0))
         (goto-char par-beg)))
      (setq arg (1+ arg)))
    (while (> arg 0)
      (let ((par-end (save-excursion (end-of-paragraph-text) (point))))
       (if (re-search-forward sentence-end par-end t)
           (skip-chars-backward " \t\n")
         (goto-char par-end)))
      (setq arg (1- arg)))
    (constrain-to-field nil opoint t)))
```

De functie ziet er op het eerste gezicht lang uit en het is het beste om eerst naar zijn geraamte te kijken, en daarna naar zijn spieren. De manier om het geraamte te zien is te kijken naar de expressies die aan de meest linkse kant beginnen:

```
(defun forward-sentence (&optional arg)
  "documentatie..."
  (interactive "p")
  (or arg (setq arg 1))
  (let ((opoint (point)) (sentence-end (sentence-end)))
    (while (< arg 0)
      (let ((pos (point))
            (par-beg (save-excursion (start-of-paragraph-text) (point))))
       rest-van-de-body-van-de-while-loop-bij-achterwaarts-bewegen
    (while (> arg 0)
      (let ((par-end (save-excursion (end-of-paragraph-text) (point))))
       rest-van-de-body-van-de-while-loop-bij-voorwaarts-bewegen
    hanteer-vormen-en-equivalenten
```

Dit ziet er veel eenvoudiger uit! De functiedefinitie bestaat uit documentatie, een `interactive` expressie, een `or` expressie, een `let` expressie en een `while` loop.

Laten we elk van de delen beurtelings bekijken.

We merken op dat de documentatie grondig en begrijpelijk is.

De functie heeft een `interactive "p"` declaratie. Dit betekent dat het verwerkte prefix argument, als die er is, doorgegeven wordt aan de functie als een argument.

(Dit is een getal.) Wanneer de functie geen argument krijgt doorgegeven (dit is optioneel) dan wordt het argument `arg` gebonden aan 1.

Wanneer `forward-sentence` non-interactief zonder argument wordt aangeroepen, wordt `arg` gebonden aan `nil`. De `or` expressie handelt dit af. Wat die doet is óf de waarde van `arg` laten zoals die is, maar alleen als `arg` aan een waarde gebonden is, óf het zet de waarde van `arg` op 1, in het geval dat `arg` gebonden is aan `nil`.

Hierna komt een `let`. Die specificeert de waarden van twee lokale variabelen, `opoint` en `sentence-end`. De lokale waarde van point, van voor de zoekopdracht, wordt gebruikt in de functie `constrain-to-field`, die vormen en equivalenten afhandelt. De variabele `sentence-end` wordt gezet door de functie `sentence-end`.

De `while` loops

Hierna volgen twee `while` loops. De eerste `while` heeft een waar-of-onwaar-test dat waar test als het prefix argument voor `forward-sentence` een negatief getal is. Dat is om achterwaarts te gaan. De body van deze loop vergelijkbaar met de body van de tweede `while` clausule, maar is niet exact hetzelfde. We slaan deze `while` loop over en concentreren ons de tweede `while` loop.

De tweede `while` loop is voor het voorwaarts verplaatsen van point. Het geraamte ziet er zo uit:

```
(while (> arg 0)            ; waar-of-onwaar-test
  (let varlist
    (if (waar-of-onwaar-test)
        dan-deel
      anders-deel
  (setq arg (1- arg))))     ; while loop decrementer
```

De `while` loop is van het aflopende type. (Zie Sectie 11.1.4 “Loop met een decrementele teller”, pagina 124.) Het heeft een waar-of-onwaar-test die waar test zolang de teller (in dit geval, de variabele `arg`) groter is dan nul. En het heeft een decrementer die elke keer als de loop herhaalt 1 aftrekt van de waarde van de teller.

Wanneer geen prefix argument aan `forward-sentence` wordt doorgegeven, wat de meest voorkomende manier is om dit commando te gebruiken, de `while` loop draait maar een keer, omdat de waarde van `arg` dan 1 is.

De body van de `while` loop bestaat uit een `let` expressie, die de lokale variabele creëert en bindt, en als body een `if` expressie heeft.

De body van de `while` loop ziet er zo uit:

```
(let ((par-end
       (save-excursion (end-of-paragraph-text) (point))))
  (if (re-search-forward sentence-end par-end t)
      (skip-chars-backward " \t\n")
    (goto-char par-end)))
```

De `let` expressie creëert en bindt de lokale variabele `par-end`. Zoals we zullen zien is deze lokale variabele gemaakt om een grens of limiet aan de reguliere zoekopdracht te geven. Wanneer de zoekopdracht faalt in het vinden van een einde van een zin de in de paragraaf, stopt het bij het bereiken van het einde van de paragraaf.

Maar laten wij eerst onderzoeken hoe `par-end` wordt gebonden aan de waarde van het einde van de paragraaf. Wat gebeurt is dat de `let` de waarde van `par-end` zet op de waarde die de Lisp interpreter teruggeeft wanneer die de expressie evalueert.

```
(save-excursion (end-of-paragraph-text) (point))
```

In deze expressie verplaatst `(end-of-paragraph-text)` point naar het einde van de paragraaf, `(point)` geeft de waarde van point terug en vervolgens herstelt `save-excursion` point naar zijn oorspronkelijke positie. Dus bindt de `let` expressie `par-end` aan de waarde die de `save-excursion` teruggeeft. wat de positie is van het einde van de paragraaf. (De functie `end-of-paragraph-text` gebruikt `forward-paragraph`, die we straks bespreken.)

Vervolgens evalueert Emacs de body van de `let`, wat een `if` expressie is, die er zo uit ziet:

```
(if (re-search-forward sentence-end par-end t) ; if-deel
    (skip-chars-backward " \t\n")              ; dan-deel
  (goto-char par-end)))                        ; anders-deel
```

De `if` test of het eerste argument waar is en zo ja, evalueert het dan-deel, anders evalueert de Emacs Lisp interpreter het anders-deel. De waar-of-onwaar-test van de `if` expressie is de reguliere expressie zoekopdracht.

Het mag vreemd lijken om wat er uit ziet als het echte werk van de functie `forward-sentence` hier ingegraven is, maar dat is de gebruikelijke manier waarop dit soort operaties in Lisp worden uitgevoerd.

De reguliere expressie zoekopdracht

De functie `re-search-forward` zoekt naar het einde van de zin, dat wil zeggen, naar het patroon dat is gedefinieerd met de reguliere expressie `sentence-end`.

1. De functie `re-search-forward` voert een zij-effect uit, dat is het verplaatsen van point naar eind van de gevonden zin.
2. De functie `re-search-forward` geeft een waarde waar terug. Dit is de waarde die de `if` ontvangt, en betekent dat de zoekopdracht succesvol was.

Het zij-effect, de verplaatsing van point, is afgerond voordat de `if` de waarde krijgt overhandigd die teruggegeven is door de succesvolle afronding van de zoekopdracht.

Wanneer de `if` functie de waarde waar ontvangt van een succesvolle aanroep van `re-search-forward` evalueert de `if` het dan-deel, dit is de expressie `(skip-chars-backward " \t\n")`. Deze expressie beweegt achterwaarts over spaties, tabs

of carriage returns totdat een karakter gevonden is en zet dan point achter dat karakter. Omdat point al verplaatst was naar het eind van het patroon dat het einde van een zin markeert, zet deze actie point direct achter het afsluitende karakter van de zin, wat meestal een punt is.

Anderzijds, wanneer de functie `re-search-forward` faalt in het vinden van het patroon dat het einde van een zin markeert, geeft de functie onwaar terug. Deze onwaar zorgt dat de `if` het derde argument evalueert, dit is `(goto-char par-end)`: het verplaatst point naar het einde van de paragraaf.

(En wanneer de tekst een vorm of equivalent is en point kan niet volledig bewegen, dan komt de functie `constrain-to-field` in het spel.)

Reguliere expressie zoekopdrachten zijn uiterst nuttig en het patroon geïllustreerd met `re-search-forward` waarin de zoekopdrachten de test is van een `if` expressie is handig. Je ziet of schrijft vaak code dat die patroon incorporeert.

12.4 `forward-paragraph`: een goudmijn van functies

De functie `forward-paragraph` verplaatst point naar het einde van de paragraaf. Die is meestal gebonden aan *M-}* en maakt gebruik van een aantal functies die op zichzelf belangrijk zijn, waaronder `let*`, `match-beginning` en `looking-at`.

De functiedefinitie van `forward-paragraph` is aanzienlijk langer dan de functiedefinitie van `forward-sentence` omdat het met een paragraaf werkt, waarvan elke regel met een opvul-prefix kan beginnen.

Een opvul-prefix bestaat uit een string van karakters die herhaald wordt aan het begin van elke regel. Bijvoorbeeld in Lisp code is het een conventie om elke regel van een paragraaf-lang commentaar te beginnen met '`;;; `'. In Text-mode bestaat een ander gebruikelijke opvul-prefix uit vier spaties, om een ingesprongen paragraaf te maken. (Zie Sectie "Fill Prefix" in *The GNU Emacs Manual*, voor meer informatie over opvul-prefixes.)

Het bestaan van een opvul-prefix betekent dat naast het kunnen vinden van het einde van de paragraaf wiens regels helemaal links beginnen, de functie `forward-paragraph` ook in staat moet zijn het einde van de paragraaf te vinden wanneer alle of veel van de regels in het buffer met een opvul-prefix beginnen.

Bovendien is het soms praktisch om een bestaand opvul-prefix te negeren, in het bijzonder wanneer lege regels paragrafen scheiden. Dit is een extra complicatie.

In plaats van alles van de functie `forward-paragraph` te tonen, tonen we alleen delen er van. Het kan ontmoedigend zijn de functie zonder voorbereiding te lezen.

Globaal ziet de functie er zo uit:

```
(defun forward-paragraph (&optional arg)
  "documentatie..."
  (interactive "p")
  (or arg (setq arg 1))
  (let*
      varlist
    (while (and (< arg 0) (not (bobp)))      ; achterwaarts-bewegende-code
      ...
    (while (and (> arg 0) (not (eobp)))      ; voorwaarts-bewegende-code
      ...
```

De eerste delen van de functie zijn routine: de argumentlist van de functie bestaat uit een optioneel argument. Documentatie volgt daarna.

De kleine letter '`p`' in de `interactive` declaratie betekent dat het verwerkte prefix argument, als die er is, wordt doorgegeven aan de functie. Dit is een getal, en is het aantal herhalingen van hoeveel paragrafen point wordt verplaatst. De `or` expressie in de volgende regel behandelt de gebruikelijke situatie wanneer geen argument aan de functie wordt doorgegeven, wat gebeurt wanneer de functie wordt aangeroepen vanuit andere code in plaats van interactief. Deze situatie is eerder beschreven (Zie Sectie 12.3 "forward-sentence", pagina 145.) Nu bereiken we het einde van het vertrouwde deel van deze functie.

De `let*` expressie

De volgende regel van de functie `forward-paragraph` begint met een `let*` expressie (zie "`let*` geïntroduceerd", pagina 57), waarin Emacs een totaal van zeven variabelen bindt: `opoint`, `fill-prefix-regexp`, `parstart`, `parsep`, `sp-parstart`, `start` en `found-start`. Het eerste deel van de `let*` expressie ziet er als volgt uit:

```
(let* ((opoint (point))
       (fill-prefix-regexp
        (and fill-prefix (not (equal fill-prefix ""))
             (not paragraph-ignore-fill-prefix)
             (regexp-quote fill-prefix)))
       ;; Remove ^ from paragraph-start and paragraph-sep if they are there.
       ;; These regexps shouldn't be anchored, because we look for them
       ;; starting at the left-margin.  This allows paragraph commands to
       ;; work normally with indented text.
       ;; This hack will not find problem cases like "whatever\\|^something".
       (parstart (if (and (not (equal "" paragraph-start))
                          (equal ?^ (aref paragraph-start 0)))
                     (substring paragraph-start 1)
                   paragraph-start))
       (parsep (if (and (not (equal "" paragraph-separate))
                        (equal ?^ (aref paragraph-separate 0)))
                   (substring paragraph-separate 1)
                 paragraph-separate))
       (parsep
        (if fill-prefix-regexp
            (concat parsep "\\|"
                    fill-prefix-regexp "[ \t]*$")
          parsep))
       ;; This is used for searching.
       (sp-parstart (concat "^[ \t]*\\(?:" parstart "\\|" parsep "\\)"))
       start found-start)
  ...)
```

De variabele `parsep` verschijnt twee keer, ten eerste om instanties van '`^`' te verwijderen en ten tweede om opvul-prefixes te behandelen.

De variabele `opoint` is gewoon de waarde van `point`. Zoals je kunt raden wordt het gebruikt in een `constrain-to-field` expressie, net als in `forward-sentence`.

De variabele `fill-prefix-regexp` wordt gezet op de waarde teruggeven door de volgende lijst te evalueren:

```
(and fill-prefix
     (not (equal fill-prefix ""))
     (not paragraph-ignore-fill-prefix)
     (regexp-quote fill-prefix))
```

Dit is een expressie wiens eerste element de speciale vorm `and` is.

Zoals we eerder (zie "De `kill-new` functie", pagina 98) leerden, evalueert de speciale vorm `and` elk van zijn argumenten tot dat een van de de argumenten een waarde `nil` teruggeeft, in dat geval geeft de `and` een waarde `nil` terug. Echter, wnanneer geen van de arugmenten een waarde `nil` teruggeeft, wordt de waarde die het resultaaat van het evalueren van het laatste argument teruggegeven. (Omdat zo'n waarde niet `nil` is, wordt het als waar beschouwt in Lisp.) Met andere woorden, een `and` expressie geeft alleen een waarde waar terug wanneer al zijn argumenten waar zijn.

In dit geval wordt de variabele `fill-prefix-regexp` uitsluitend gebonden aan een non-`nil` waarde wanneer de volgende vier expressies een waarde waar (dus non-`nil`) produceren wanneer zij worden geëvalueerd. Zo niet, dan wordt `fill-prefix-regexp` gebonden aan `nil`.

`fill-prefix`
: Wanneer deze variabele wordt geëvalueerd wordt de waarde van de fill prefix, als die er is, teruggegeven. Wanneer er geen fill prefix is, dan geeft de variabel `nil` terug.

`(not (equal fill-prefix "")`
: Deze expressie kijkt of een bestaand fill prefix een lege string is, oftewel een string zonder karakters er in. Een lege string is geen nuttige opvul-prefix.

`(not paragraph-ignore-fill-prefix)`
: Deze expressie geeft `nil` terug wanneer de variabele `paragraph-ignore-fill-prefix` aan staat, doordat het op de waarde waar, zoals `t`, is gezet.

`(regexp-quote fill-prefix)`
: Dit is het laatste argument van de speciale vorm `and`. Wanneer al de argumenten van de `and` waar zijn, wordt de waarde die resulteert uit het evalueren van deze expressie teruggeven door de `and` expressie en gebonden aan de variabele `fill-prefix-regexp`.

Het resultaat van een succesvolle evaluatie van deze `and` expressie is dat `fill-prefix-regexp` wordt gebonden aan de waarde van `fill-prefix`, aangepast door de functie `regexp-quote`. Wat `regexp-quote` doet is het lezen van een string en een reguliere expressie teruggeven die exact de string matcht, en niets anders matcht. Dit betekent dat de `fill-prefix-regexp` wordt gezet naar een waarde is exact het opvul-prefix matcht als die bestaat. Zo niet, dan wordt de variabele gezet op `nil`.

De volgende twee lokale variabelen in de `let*` expressie zijn ontworpen om instanties van '`^`' te verwijderen uit `parstart` en `parsep`, de lokale variabelen die de

start van de paragraaf en de paragraaf separator aanduiden. De volgende expressie stelt `parsep` opnieuw. Dit is om opvul-prefixes te behandelen.

Het is dit instellen dat vereist dat de definitie `let*` aanroept in plaats van `let`. De waar-of-onwaar-test van de `if` hangt af van of de variabele `fill-prefix-regexp` naar `nil` evalueert, of naar een andere waarde.

Wanneer `fill-prefix-regexp` geen waarde heeft dan evalueert Emacs het anders-deel van de `if` expressie en bindt `parsep` naar de lokale waarde. (`parsep` is een reguliere expressie die matcht wat paragrafen scheidt.)

Maar wanneer `fill-prefix-regexp` een waarde heeft, evalueert Emacs het dan-deel van de `if` expressie en bindt `parsep` aan een reguliere expressie die de `fill-prefix-regexp` als deel van het patroon bevat.

Specifiek wordt `parsep` gezet op de originele waarde van de paragraaf scheidende reguliere expressie samengevoegd met een alternatieve expressie die bestaat uit de `fill-prefix-regexp` gevolgd door optionele witte ruimte als einde van de regel. (De witte ruimte is gedefinieerd met `"[ \t]*$"`.) De '`\\|`' definieert die deel van de regexp als een alternatief van `parsep`.

Volgens het commentaar in de code wordt de volgende lokale variabele, `sp-parstart` gebruikt om te zoeken en de laatste twee, `start` en `found-start` worden op `nil` gezet.

Nu komen we bij de body van de `let*`. Het eerste deel van de body van de `let*` behandelt het geval wanneer de functie een negatief argument krijgt en daarom achterwaarts beweegt. We slaan dit stuk over.

De voorwaarts bewegende `while` loop

Het tweede deel van de body van de `let*` behandelt de voorwaartse beweging. Het is een `while` loop dit zichzelf herhaalt zolang de waard van `arg` groter is dan nul. In de meest gebruikte manier van de functie is de waarde van het argument 1, en wordt de body van de `while` loop exact een keer geëvalueerd, en verplaatst de cursor een paragraaf voorwaarts.

Dit deel behandelt drie situaties: wanneer point tussen twee paragrafen is, wanneer er een opvul-prefix is en wanneer er geen opvul-prefix is.

De `while` loop ziet er zo uit:

```
;; ga voorwaarts en niet aan het eind van het buffer
(while (and (> arg 0) (not (eobp)))

  ;; tussen paragrafen
  ;; Move forward over separator lines...
  (while (and (not (eobp))
              (progn (move-to-left-margin) (not (eobp)))
              (looking-at parsep))
    (forward-line 1))
  ;;  Dit verlaagt de loop
  (unless (eobp) (setq arg (1- arg)))
  ;; ... and one more line.
  (forward-line 1)
```

```
(if fill-prefix-regexp
    ;; There is a fill prefix; it overrides parstart;
    ;; we go forward line by line
    (while (and (not (eobp))
                (progn (move-to-left-margin) (not (eobp)))
                (not (looking-at parsep))
                (looking-at fill-prefix-regexp))
      (forward-line 1))

  ;; There is no fill prefix;
  ;; we go forward character by character
  (while (and (re-search-forward sp-parstart nil 1)
              (progn (setq start (match-beginning 0))
                     (goto-char start)
                     (not (eobp)))
              (progn (move-to-left-margin)
                     (not (looking-at parsep)))
              (or (not (looking-at parstart))
                  (and use-hard-newlines
                       (not (get-text-property (1- start) 'hard)))))
    (forward-char 1))

  ;; and if there is no fill prefix and if we are not at the end,
  ;;     go to whatever was found in the regular expression search
  ;;     for sp-parstart
  (if (< (point) (point-max))
      (goto-char start))))
```

We kunnen zien dat dit een aflopende teller `while` loop is, met de expressie `(setq arg (1- arg))` als de decrementer. Die expressie is niet ver van de `while` loop, maar is verstopt in een andere Lisp macro, een `unless` macro. Tenzij we aan het eind van het buffer zijn—dat is wat de functie `eobp` vaststelt, het is een afkorting van '`End Of Buffer P`'—verlagen we de waarde van `arg` met één.

(Wanneer we aan het eind van het buffer zijn, kunnen we niet meer voorwaarts en test de volgende loop van de `while` expressie onwaar omdat de test een `and` met `(not (eobp))` is. De functie `not` betekent precies wat je verwacht, het is een andere naam voor `null`, een functie die waar teruggeeft wanneer zijn argument onwaar is.)

Interessant genoeg wordt de loop teller niet eerder verlaagd dan dat we de ruimte tussen paragrafen verlaten, tenzij we aan het eind van het buffer komen of de lokale variabele van de paragraaf separator niet meer zien.

De tweede `while` heeft ook een `(move-to-left-margin)` expressie. De functie spreekt voor zich. Die bevindt zich binnen een `progn` expressie en is niet het laatste element van de body, dus die wordt alleen aangeroepen voor het zij-effect, het verplaatsten van point naar de linker kantlijn op de huidige regel.

De functie `looking-at` spreekt ook voor zich. het geeft waar terug wanneer de tekst achter point matcht met de reguliere expressie die als zijn argument is doorgegeven.

De rest van de body van de loop ziet er op het eerste gezicht moeilijk uit, maar is logisch wanneer je die begint te begrijpen.

Bedenk eerst wat gebeurt wanneer er een opvul-prefix is:

```
(if fill-prefix-regexp
    ;; There is a fill prefix; it overrides parstart;
    ;; we go forward line by line
    (while (and (not (eobp))
                (progn (move-to-left-margin) (not (eobp)))
                (not (looking-at parsep))
                (looking-at fill-prefix-regexp))
      (forward-line 1))
```

De expressie verplaatst point voorwaarts regel voor regel zolang vier condities waar zijn:

1. Point is niet aan het eind van het buffer.
2. We kunnen naar de linker kantlijn van de tekst verplaatsen en zijn niet aan het eind van het buffer.
3. De tekst die volgt op point scheidt geen paragrafen.
4. Het patroon volgend op point is de fill prefix reguliere expressie.

De laatste conditie kan verwarrend zijn, totdat je herinnert dat point naar het begin van de regel is verplaatst in het begin van de functie `forward-paragraph`. Dat betekent dat wanneer de tekst een opvul-prefix heeft, de functie `looking-at` het zal zien.

Bedenk wat gebeurt wanneer er geen opvul-prefix is.

```
(while (and (re-search-forward sp-parstart nil 1)
            (progn (setq start (match-beginning 0))
                   (goto-char start)
                   (not (eobp)))
            (progn (move-to-left-margin)
                   (not (looking-at parsep)))
            (or (not (looking-at parstart))
                (and use-hard-newlines
                     (not (get-text-property (1- start) 'hard)))))
  (forward-char 1))
```

De `while` loop laat ons voorwaarts zoeken naar `sp-parstart`, wat de combinatie is van mogelijke witte ruimte met de lokale waarde van de start van een paragraaf of van een paragraaf separator.

De twee expressies,

```
(setq start (match-beginning 0))
(goto-char start)
```

betekenen ga naar de start van tekst gematcht door de reguliere expressie zoekopdracht.

De `(match-beginning 0)` expressie is nieuw. Het geeft het getal dat de lokatie van de start van de tekst aangeeft die was gematcht door de laatste zoekopdracht.

De functie `match-beginning` wordt hier gebruikt vanwege een karakteristiek van een voorwaartse zoekopdracht: een succesvolle zoekopdracht, ongeacht of het een gewone zoekopdracht of een reguliere expressie zoekopdracht is, verplaatst point naar het einde van de tekst die is gevonden. In dit geval verplaatst een succesvolle zoekopdracht point naar het einde van het patroon voor `sp-parstart`.

Wij willen echter point aan het einde van de huidige paragraaf zetten, niet ergens anders. Omdat de zoekopdracht mogelijk de paragraaf separator bevat, kan point bij het begin van de volgende terecht komen tenzij we een expressie gebruiken die `match-beginning` bevat.

Wanneer het een argument van 0 krijgt, geeft `match-beginning` de positie terug die de start is van de tekst gematcht met de meest recente zoekopdracht. In dit geval zoekt de meest recente zoekopdracht naar `sp-parstart`. De `(match-beginning 0)` expressie geeft de beginposiitie van dat patroon, in plaats van de eindpositie van dat patroon.

(Overigens, wanneer een positief getal als argument wordt doorgegeven, geeft de functie `match-beginning` de lokatie van point bij die tussen haakjes geplaatste expressie in de laatste zoekopdracht, tenzij die tussen haakjes geplaatste expressie begint met `\(?:`. Ik weet niet waarom `\(?:` hier verschijnt, aangezien het argument 0 is.)

De laatste expressie wanneer er geen opvul-prefix is

```
(if (< (point) (point-max))
    (goto-char start))))
```

(Merk op dat dit stukje code ongewijzigd gekopieerd is van de originele code, waardoor de twee extra haakjes sluiten horen bij de voorafgaande `if` en `while`.)

Dit vertelt dat wanneer er geen opvul-prefix is en wanneer wij niet aan het einde zijn, point naar het begin moet verplaatsen van wat gevonden was door de reguliere expressie zoekopdracht naar `sp-parstart`.

De volledige definitie van de functie `forward-paragraph` bevat niet alleen code om voorwaarts te gaan, maar ook code om achterwaarts te gaan.

Wanneer je dit in GNU Emacs leest en je de gehele functie wilt zien, dan kan je *`C-h f`* (`describe-function`) typen, en daarna de naam van de functie. Dit geeft je de functiedocumentatie en de naam van de library die de broncode bevat. Plaats point op de naam van de library en druk op de `RET` toets. Je brengt je direct naar de bron. (Zorg dat je de broncodes installeert! Zonder die ben je als iemand die probeert met de ogen dicht een auto te besturen.)

12.5 Terugblik

Hier is een korte samenvatting van sommige recent geïntroduceerde functies.

`while` Evalueer herhaaldelijk de body van de expressie zolang als het eerste element van de body waar test. Geef daarna `nil` terug. (De expressie wordt uitsluitend voor de zij-effecten geëvalueerd.)

Bijvoorbeeld:

```
(let ((foo 2))
  (while (> foo 0)
    (insert (format "foo is %d.\n" foo))
    (setq foo (1- foo))))

 ⇒      foo is 2.
        foo is 1.
        nil
```

(De functie `insert` voegt het argument bij point in. de functie `format` geeft een geformatteerde string terug op dezelfde manier als `message` de argumenten formatteert. `\n` produceert een nieuwe regel.)

`re-search-forward`
: Zoek voor een patroon, en als het patroon is gevonden, verplaats point direct er achter.

  Verwacht vier argumenten, zoals `search-forward`:

  1. Een reguliere expressie die het zoekpatroon specificeert. (Denk er aan om dubbele aanhalingstekens om dit argument te zetten!)
  2. Optioneel, de limiet van de zoekopdracht.
  3. Optioneel, wat te doen als de zoekopdracht mislukt, geef `nil` terug of een fout.
  4. Optioneel, hoeveel keer de zoekopdracht te herhalen. Wanneer negatief, zoek achterwaarts.

`let*`
: Bind sommige variabelen lokaal aan bepaalde waardes en evalueer dan de overige argumenten en geeft de waarde van de laatste terug. Gebruik de lokale waarden van eerder gebonden variabelen bij het binden van lokale variabelen.

  Bijvoorbeeld:

```
(let* ((foo 7)
       (bar (* 3 foo)))
  (message "`bar' is %d." bar))
    ⇒ ‘bar’ is 21.
```

`match-beginning`
: Geef de positie van de start van de tekst gevonden door de laatste reguliere expressie.

`looking-at`
: Geef `t` voor waar terug wanneer de tekst achter point overeenkomt met het argument, wat een reguliere expressie moet zijn.

`eobp`
: Geef `t` voor waar terug wanneer point aan het einde van het bereikbare deel van een buffer is. Het einde van het bereikbare deel is het einde van het buffer wanneer het buffer niet versmald is. Het is het einde van het versmalde deel wanneer het buffer is versmald.

12.6 Oefeningen met `re-search-forward`

- Schrijf een functie om te zoeken met een reguliere expressie die overeenkomt met twee of meer opeenvolgende lege regels.
- Schrijf een functie die zoekt naar gedupliceerde woorden, zoals "de de". Zie Sectie "Syntax of Regular Expressions" in *The GNU Emacs Manual*, voor informatie hoe je een regexp schrijft (een reguliere expressie) om een string te matchen die uit twee identieke helften bestaat. De functie die ik gebruik is beschreven in de appendix, samen met verschillende regexps. Appendix A "`de-de` Gedupliceerde woorden functie", pagina 228.

13 Tellen via repetitie en regexps

Repetitie en reguliere expressie zoekopdrachten zijn krachtige instrumenten de je vaak gebruikt wanneer je code in Emacs Lisp schrijft. Dit hoofdstuk illustreert het gebruik van reguliere expressie zoekopdrachten bij de constructie van woorden telling commando's, met `while` loops en recursie.

De standaard Emacs distributie bevat functies om het aantal regels en het het aantal woorden in een region te tellen.

Sommige soorten schrijfwerk vragen je woorden te tellen. Zoals wanneer je een essay schrijft kan dat gelimiteerd zijn op 800 woorden. Wanneer je een roman schrijft kan je je zelf opleggen om 1000 woorden per dag te schrijven. Het klinkt raar, maar Emacs had heel lang geen woorden tel commando. Misschien gebruikten mensen Emacs voornamelijk voor code of soorten documentatie die geen woordtelling vereisen, of misschien beperkte zij zich tot het woorden tel commando van het besturingsysteem zoals `wc`. Misschien dat mensen de conventie onder uitgevers volgden en berekenden zij de woord telling door het aantal karaktrs in een document door vijf te delen.

Een commando om woorden te tellen kan op veel verschillende manieren gecompleteerd worden. Hier volgen sommige voorbeelden die je wellicht wilt vergelijken met het standaard Emacs commando `count-words-region`.

13.1 De `tel-woorden-voorbeeld` functie

Een woorden tel commando kan woorden tellen in een regel, paragraaf, region of buffer. Wat zou het commando moeten doen? Je kunt een commando maken dat het aantal woorden telt in een volledig buffer. Echter, de Emacs traditie moedigt flexibiliteit aan—je wil misschien alleen de woorden in een sectie tellen in plaats van het hele buffer. Het is daarom verstandiger om het commando het aantal woorden in een region te laten tellen. Wanneer je eenmaal een commando hebt dat woorden in een region telt, dan kan je als je dat wilt woorden in het hele buffer tellen door het te markeren met *`C-x h`* (`mark-whole-buffer`).

Woorden tellen is duidelijk een repeterende handeling: startend bij het begin van de region tel je het eerste woord, dan het tweede woord, dan het derde, enzovoorts., totdat je het einde van de region bereikt. Dit betekent dat het tellen van woorden uitstekend geschikt is voor recursie of voor een `while` loop.

Eerst gaan we een woord tel commando maken met een `while` loop, daarna met recursie. Het commando is uiteraard interactief.

Het sjabloon voor een interactieve functiedefinitie is, zoals altijd:

```
(defun naam-van-de-functie (argument-list)
  "documentatie..."
  (interactive-expression...)
  body...)
```

Wat we moeten doen is de slots invullen.

De naam van def functie moet zelf verklarend en makkelijk te houden zijn. `count-words-region` is een voor de hand liggende keuze. Maar omdat die naam

gebruikt wordt door het standaard Emacs commando om woorden te tellen, noemen we onze implantatie `tel-woorden-voorbeeld`.

De functie telt woorden in een region. Dat betekent dat de arugmentlist symbolen moet bevatten die gebonden worden aan de twee posities, het begin en einde van de region. Deze twee posities noemen we respectievelijk '`begin`' en '`end`'. De eerste regel van de documentatie met een enkele zin zijn, omdat dit alles wat als documentatie getoond wordt door een commando zoals `apropos`. De interactieve expressie heeft de vorm '`(interactive "r")`', omdat die zorgt dat Emacs het begin en einde van de region doorgeven aan de arugmentlist van de functie. Dit is allemaal routine.

De body van de functie moet drie taken doen: ten eerste de condities opzetten waaronder de `while` loop de woorden kan tellen, ten tweede het runnen van de '`while`' loop, en ten derde, de gebruiker een boodschap geven.

Wanneer een gebruiker `tel-woorden-voorbeeld` aanroept kan point aan het begin of aan het einde van de region staan. Het tel-proces moet echter bij het begin van de region starten. Dit betekent dat we point daar willen zetten wanneer die daar nog niet is. Het uitvoeren van `(goto-char beginning)` verzekert dit. We willen natuurlijk point terug zetten op zijn verwachte positie wanneer de functie zijn werk gedaan heeft. Om deze reden moet de body ingevoegd zijn in een `save-excursion` expressie.

Het centrale deel van de body van de functie bestaat uit een `while` loop waarin een expressie point voorwaarts van woord naar woord laat springen, en een andere expressie deze sprongen telt. De waar-of-onwaar-test van de `while` loop moet naar waar testen zolang point voorwaarts moet springen, en onwaar wanneer point aan het einde van de region is.

We zouden `(forward-word 1)` kunnen gebruiken om point woord voor woord voorwaarts te verplaatsen, maar het is makkelijker om te kijken wat Emacs als "woord" identificeert, wanneer we een reguliere expressie zoekopdracht gebruiken.

Een reguliere expressie zoekopdracht die het patroon vindt waarvoor het zoekt, zet point achter het laatst gematchte karakter. Dit betekent dat opvolgende succesvolle zoekopdrachten point woord voor woord voorwaarts verplaatst.

In de praktijk willen we dat de reguliere expressie zoekopdracht over witte ruimte en leestekens tussen woorden springt en ook over de woorden zelf. Een regexp die weigert over witte ruimte tussen woorden te springen zou nooit over een meer dan een woord springen! Dit betekent dat de regexp de witte ruimte en leestekens volgend op een woord moet bevatten, als die er is, zowel als het woord zelf. (Een woord kan een buffer beëindigen en kan geen opvolgende witte ruimte of leestekens hebben, dus dat deel van de regexp moet optioneel zijn.)

Dus wat we willen is dat regexp een patroon is dat een of meer woord-vormende karakters, optioneel gevolgd door een or meer karakters die niet woord-vormend zijn, definieert.

```
\w+\W*
```

De syntax-table van het buffer stelt vast welke karakters woord-vormend zijn en welke niet. Voor meer informatie over syntax, zie Sectie "Syntax Tables" in *The GNU Emacs Lisp Reference Manual*.

De zoek expressie ziet er zo uit:

```
(re-search-forward "\\w+\\W*")
```

(Merk op dat de paren backslashes aan de '`w`' en '`W`' voorafgaan. Een enkele backslash heeft een speciale betekenis voor de Emacs Lisp interpreter. Het geeft aan dat het er op volgende karakter anders geïnterpreteerd is dan gebruikelijk. Bijvoorbeeld staan de twee karakters '`\n`' voor '`newline`' in plaats van een backslash gevolgd door '`n`'. Twee backslashes achter elkaar staan voor een gewone, niet-speciale backslash, waardoor de Emacs Lisp interpreter uiteindelijk een enkele backslash gevold door een letter ziet. Daardoor weet het dat de letter speciaal is.)

We hebben een teller nodig die telt hoeveel woorden er zijn. Deze variabele moet eerst op 0 gezet worden en daarna elke keer dat Emacs door de `while` loop gaat verhoogd worden.

```
(setq count (1+ count))
```

Tenslotte willen we aan de gebruiker vertellen hoeveel woorden er in de region zijn. De functie `message` is bedoeld om dit soort informatie aan de gebruiker te presenteren. De boodschap moet zo zijn geformuleerd dat die goed is ongeacht hoeveel woorden er in de region zijn: we willen niet zeggen "er zijn 1 woorden in de region". Het conflict tussen enkelvoud en meervoud is grammaticaal onjuist. Wij lossen dit op met een conditionele expressie die verschillende boodschappen evalueert, afhankelijk van het aantal woorden in de region. Er zijn drie mogelijkheden: geen woorden in de region, één woord in de region, en meer dan een woord. Dit betekent dat de speciale vorm `cond` geschikt is.

Dit alles leidt tot de volgende functiedefinitie:

```
;;; Eerste versie; heeft bugs!
(defun tel-woorden-voorbeeld (beginning end)
  "Toon aantal woorden in de region.
Woorden worden gedefinieerd als minstens een woord-vormend
karakter gevolgd door minstens een karakter dat
niet woord-vormend is. De syntax table van het buffer
bepaald welke karakters dat zijn."
  (interactive "r")
  (message "Tel woorden in region ... ")

;;; 1. Opzetten geschikte condities.
  (save-excursion
    (goto-char beginning)
    (let ((count 0))

;;; 2. Run de while loop.
      (while (< (point) end)
        (re-search-forward "\\w+\\W*")
        (setq count (1+ count)))

;;; 3. Toon een boodschap aan de gebruiker.
      (cond ((zerop count)
             (message
              "De region heeft GEEN woorden."))
            ((= 1 count)
             (message
              "De region heeft 1 woord."))
            (t
             (message
              "The region heeft %d woorden." count))))))
```

Zo geschreven werkt de functie, maar niet in alle omstandigheden.

13.1.1 De witte ruimte bug in `tel-woorden-voorbeeld`

Het in de vorige sectie beschreven commando `tel-woorden-voorbeeld` heeft twee bugs, of eigenlijk een bug met twee manifestaties. Ten eerste als je een region markeert die alleen witte ruimte heeft in het midden van tekst, zegt `tel-woorden-voorbeeld` dat de region maar een woord heeft! Ten tweede wanneer je een region markeert met alleen witte ruimte aan het einde van het buffer of het bereikbare deel van een versmald buffer, toont het een foutboodschap die er zo uit ziet:

```
Search failed: "\\w+\\W*"
```

Wanneer je dit leest in Info in GNU Emacs, kan je deze bugs zelf testen.

Evalueer eerst de functie op de gebruikelijke manier om die te installeren.

Wanneer je dat wilt, kan je ook deze key binding installeren door het te evalueren:

```
(keymap-global-set "C-c =" 'tel-woorden-voorbeeld)
```

Om de eerste test uit te voeren zet mark en point aan begin en eind van de volgende regel en typ dan *C-c =* (of *M-x tel-woorden-voorbeeld* wanneer je *C-c =* niet gebonden hebt.)

```
een   twee  drie
```

Emacs toont je inderdaad dat de region drie woorden heeft.

Herhaal de test, maar plaats de mark aan het begin van de regel en plaats point net *voor* het woord '`een`'. Typ opnieuw het commando *C-c =* (of *M-x tel-woorden-voorbeeld*). Emacs zou je moeten vertellen dat de region geen woorden heeft, omdat het alleen uit witte ruimte aan het begin van de regel bestaat. Maar in plaats daarvan vertelt Emacs je dat de region één woord bevat!

Voor de derde test, kopieer de voorbeeldregel aan het eind van het `*scratch*` buffer en typ dan meerder spaties aan het einde van de regel. Plaats mar direct achter het woord '`drie`' en point aan het einde van de regel. (Het einde van de regel is het einde van het buffer.) Typ *C-c =* (of *M-x tel-woorden-voorbeeld*) net als je eerder deed. Opnieuw zou Emacs je moeten zeggen dat de region geen woorden heeft, omdat het bestaat uit witte ruimte aan het einde van de regel. In plaats daarvan toont Emacs een foutboodschap die '`Search failed`' zegt.

De twee bugs stammen van hetzelfde probleem.

Bekijk de eerste manifestatie van de bug, waar het commando je zegt dat de witte ruimte aan het begin van de regel één woord bevat. Dit is wat er gebeurt: Het commando `M-x tel-woorden-voorbeeld` verplaatst point naar het begin van de region. De `while` test of point kleiner is dan de waarde van `end`, wat zo is. Vervolgens zoekt en vindt de reguliere expressie zoekopdracht het eerste woord. Het zet point achter het woord. `count` wordt op één gezet. De `while` loop herhaalt, maar deze keer is de waarde van point groter dan de waarde van `end`, de loop wordt verlaten en de functie toont een boodschap die zegt dat het aantal woorden in de region één is. Kortom, de reguliere expressie zoekopdracht zoekt en vindt het woord ondanks dat het buiten de gemarkeerde region is.

In de tweede manifestatie van de bug is de region witte ruimte aan het einde van het buffer. Emacs zegt '`Search failed`'. Wat gebeurt is dat de waar-of-onwaar-test in de `while` loop op waar test, dus de zoekopdracht wordt uitgevoerd. Maar omdat er geen woorden meer zijn in het buffer, faalt de zoekopdracht.

In beide manifestaties van de bug zoekt de zoekopdracht buiten de region of probeert dat te doen.

De oplossing is om de zoekopdracht te limiteren tot de region—dat is een vrij makkelijke actie, maar zoals je inmiddels gewend bent, is het niet zo heel simpel als je zou denken.

Zoals we hebben gezien neemt de functie `re-search-forward` een zoekpatroon als eerste argument. Maar naast die eerste, vereiste, argument, accepteert die drie optionele argumenten. Het optionele tweede argument begrenst de zoekopdracht. Het optionele argument, wanneer `t`, zorgt dat de functie `nil` teruggeeft in plaats van een fout wanneer de zoekopdracht faalt. Het optionele vierde argument is het

aantal herhalingen. (In Emacs kan je de documentatie van een functie bekijken door het typen van *C-h f*, de naam van de functie en dan RET.)

In de tel-woorden-voorbeeld definitie wordt de waarde van het einde van de region bewaard in de variabele end, die doorgegeven wordt als een argument aan de functie. We kunnen dus end als argument aan de reguliere expressie zoekopdracht toevoegen.

```
(re-search-forward "\\w+\\W*" end)
```

Als je echter alleen deze wijziging in de tel-woorden-voorbeeld definitie maakt en daarna de nieuwe versie van de definitie test op een stuk witte ruimte, krijg je de foutboodschap die zegt 'Search failed'.

Wat gebeurt is het dit: de zoekopdracht is begrensd tot de region, en faalt zoals je verwacht omdat er geen woord-vormende karakters in de region zijn. Omdat die faalt, krijgen we een foutboodschap. Maar we willen in dit geval geen foutboodschap krijgen, we willen de boodschap "De region heeft GEEN woorden" krijgen.

De oplossing voor dit probleem is om re-search-forward een derde argument t te geven, waardoor de functie nil teruggeeft in plaats van een fout signaleren wanneer de zoekopdracht faalt.

Als je echter deze wijziging aanbrengt en probleert, dan zie je de boodschap "Tel woorden in de region ..." en ... en je blijft die boodschap ... zien, totdat je *C-g* (keyboard-quit) typt.

Dit is wat er gebeurt: de zoekopdracht is begrensd tot de region, zoals eerder, en faalt omdat er geen woord-vormende karakters in de region zijn, zoals verwacht. Vervolgens geeft de re-search-forward expressie nil terug. Het doet niets anders. In het bijzonder verplaatst het point niet, wat het doet als zij-effect wanneer het een zoekdoel vindt. Nadat de re-search-forward expressie nil teruggeeft, wordt de volgende expressie in de while loop geëvalueerd. Deze expressie verhoogt de teller. Daarna herhaalt de loop. De waar-of-onwaar-test test op waar, omdat de waarde van point nog steeds kleiner is dan end, doordat de re-search-forward point niet verplaatst heeft. ... en de cyclus herhaalt ...

De tel-woorden-voorbeeld definitie vereist nog een andere modificatie, om de waar-of-onwaar-test van de while loop naar onwaar te laten testen wanneer de zoekopdracht faalt. Anders gezegd, er zijn twee condities waaraan voldoen moet worden in de waar-of-onwaar-test voordat de woordtelling variabele wordt verhoogd: point moet nog steeds in de region zijn en de zoek expressie moet een woord gevonden hebben om te tellen.

Omdat zowel de eerste als de tweede conditie samen waar moeten zijn, kunnen de twee expressies, de region test en de zoekexpressie worden samengevoegd met een speciale vorm and en ingebed in de while loop, zoals dit:

```
(and (< (point) end) (re-search-forward "\\w+\\W*" end t))
```

Voor informatie over and, zie "De kill-new functie", pagina 98.)

De re-search-forward expressie geeft t terug wanneer de zoekopdracht slaagt en als zij-effect verplaatst die point. Vervolgens, wanneer woorden worden gevonden, wordt point door de region verplaatst. Wanneer de zoekexpressie faalt in het vinden van nog een woord, of wanneer point het einde van de region bereikt, test de waar-

of-onwaar-test op onwaar, de `while` loop eindigt en de `tel-woorden-voorbeeld` functie toont een van zijn boodschappen.

Na het incorporeren van deze laatste wijzigingen, werkt `tel-woorden-voorbeeld` zonder bugs (tenminste, zonder bugs die ik heb gevonden!). Hier is hoe die er uitziet:

```
;;; Finale versie: while
(defun tel-woorden-voorbeeld (beginning end)
  "Toon aantal woorden in de region."
  (interactive "r")
  (message "Tel woorden in region ... ")

;;; 1. Opzetten geschikte condities.
  (save-excursion
    (let ((count 0))
      (goto-char beginning)

;;; 2. Run de while loop.
      (while (and (< (point) end)
                  (re-search-forward "\\w+\\W*" end t))
        (setq count (1+ count)))

;;; 3. Toon een boodschap aan de gebruiker.
      (cond ((zerop count)
             (message
              "De region heeft GEEN woorden."))
            ((= 1 count)
             (message
              "De region heeft 1 woord."))
            (t
             (message
              "The region heeft %d woorden." count))))))
```

13.2 Recursief woorden tellen

Je kunt de functie om woorden te tellen zowel recursief als met een `while` loop schrijven. Laten we zien hoe je dat doet.

Eerst moeten we herkennen dat de functie `tel-woorden-voorbeeld` drie taken heeft: het zet de geschikte condities op om de woorden die voorkomen te tellen, het telt de woorden in de region, and het geeft de gebruiker een boodschap die zegt hoeveel woorden er zijn.

Wanneer we een enkele recursieve functie schrijven die alles doet, krijgen we een boodschap bij elke recursieve aanroep. Wanneer de region 13 woorden bevat, krijgen we dertien boodschappen, de een na de ander. Dat willen we niet! In plaats daarvan moeten we twee functies schrijven om het werk te doen, waarvan eentje (de recursieve functie) wordt gebruikt binnen de ander. Een functie zet de condities op en toont de boodschap, de ander geeft de woordtelling terug.

Laten we starten met de functie die de boodschap laat zien. We blijven deze functie `tel-woorden-voorbeeld` noemen.

Dit is de functie die de gebruiker aanroept. Die is interactief. Die is vergelijkbaar met de vorige versie van deze functie, behalve dat die `recursive-count-words` aanroept om vast te stellen hoeveel woorden er in de region zijn.

We kunnen gemakkelijk voor deze functie een sjabloon samenstellen, gebaseerd op onze voorgaande versies:

```
;; Recursieve versie; met reguliere expressie zoekopdracht
(defun tel-woorden-voorbeeld (beginning end)
  "documentatie..."
  (interactieve-expressie...)

;;; 1. Opzetten geschikte condities.
  (verklarende boodschap)
  (set-up functie...

;;; 2. Tel de woorden.
     recursieve aanroep

;;; 3. Toon een boodschap aan de gebruiker.
     boodschap met de woord telling))
```

De definitie ziet er eenvoudig uit, behalve dat we op enige manier de telling teruggegeven door een recursieve aanroep moet worden doorgegeven aan de boodschap die de woord telling toont. Een beetje nadenken suggereert dat dit kan met een `let` expressie: we kunnen een variabele in de varlist van een `let` expressie binden aan het aantal woorden in de region, die wordt teruggegeven door de recursieve aanroep. Dan kan de `cond` expressie met een binding de waarde aan de gebruiker tonen.

Vaak denkt met bij de binding binnen een `let` expressie als iets ondergeschikt aan het primaire werk van een functie. Maar in dit geval wordt dat wat je als primaire werk van de functie beschouwt, het tellen van woorden, gedaan binnen de `let` expressie.

Met `let` ziet de functiedefinitie er zo uit:

```
(defun tel-woorden-voorbeeld (beginning end)
  "Print number of words in the region."
  (interactive "r")

  ;;; 1. Opzetten geschikte condities.
  (message "Tel woorden in region ... ")
  (save-excursion
    (goto-char beginning)

  ;;; 2. Tel de woorden.
    (let ((count (recursive-count-words end)))

  ;;; 3. Toon een boodschap aan de gebruiker.
      (cond ((zerop count)
             (message
              "De region heeft GEEN woorden."))
            ((= 1 count)
             (message
              "De region heeft 1 woord."))
            (t
             (message
              "The region heeft %d woorden." count))))))
```

Vervolgens moeten we de recursieve telfunctie schrijven.

Een recursieve functie heeft tenminste drie delen: de doe-opnieuw-test, de volgende-stap-expressie en de recursieve aanroep.

De doe-opnieuw-test stelt vast of de functie wel of niet opnieuw aangeroepen wordt. Omdat we woorden in een region tellen en een functie kunnen gebruiken die voor elk woord point voorwaarts verplaatst, kan de doe-opnieuw-test controleren of point nog steeds binnen de region is. De doe-opnieuw-test moet de waarde van point ontdekken en bepalen of point voor, op, of na de waarde van het einde van de region is. We kunnen de functie `point` gebruiken om point te lokaliseren. Vanzelfsprekend moeten we de waarde van het einde van de region als argument aan de recursieve telfunctie doorgeven.

Daarnaast moet de doe-opnieuw-test testen of de zoekopdracht een woord vindt. Wanneer die dat niet doet, moet de functie niet langer zichzelf aanroepen.

De volgende-stap-expressie verandert een waarde zodat wanneer de recursieve functie geacht wort te stoppen met zich zelf aan te roepen, die stopt. Preciezer gezegd, de volgende-stap-expressie verandert een waarde zodat op het juiste moment de doe-opnieuw-test de recursieve functie stopt zichzelf aan te roepen. In dit geval dan de volgende-stap-expressie de expressie zijn die point woord voor woord voorwaarts verplaatst.

Het derde deel van een resursieve functie is de recursieve aanroep.

Ook moeten we ergens een deel hebben dat het werk van de functie doet, een deel dat de woorden telt. Een vitaal deel!

Maar we hebben nu al een overzicht van de recursieve telfunctie:

```
(defun recursieve-tel-woorden (region-end)
  "documentatie..."
   doe-opnieuw-test
   volgende-stap-expressie
   recursieve aanroep)
```

Nu moeten de slots invullen. Laten we beginnen met de meest eenvoudige gevallen: wanneer point aan het einde of voorbij de region is, dan kunnen er geen woorden in de region zijn, en de functie moet nul teruggeven. Ook als de zoekopdracht faalt zijn er geen woorden om te tellen, en moet de functie nul teruggeven.

Anderzijds, wanneer point binnen de region is en de zoekopdracht is succesvol, moet de functie zichzelf opnieuw aanroepen.

Dus de doe-opnieuw-test zou er zo uit moeten zien:

```
(and (< (point) region-end)
     (re-search-forward "\\w+\\W*" region-end t))
```

Merk op dat de zoek expressie onderdeel is van de doe-opnieuw-test—de functie geeft `t` terug wanneer de zoekopdracht slaagt en `nil` wanneer die faalt. (Zie Sectie 13.1.1 “De witte ruimte bug in `tel-woorden-voorbeeld`”, pagina 161, voor een uitleg over hoe `re-search-forward` werkt.)

De doe-opnieuw-test is een waar-of-onwaar-test van een `if` clausule. Wanneer de doe-opnieuw-test slaagt, moet het dan-deel van de `if` clausule de functie opnieuw aanroepen. Maar wanneer die faalt, moet het anders-deel nul teruggeven, omdat of point buiten de region is, of de zoekopdracht faalde omdat er geen woorden waren om te vinden.

Maar voordat we de recursieve aanroep bekijken, moeten we naar de volgende-stap-expressie kijken. Wat is die? Die is interessant genoeg het zoek-deel van de doe-opnieuw-test.

Naast `t` of `nil` voor de doe-opnieuw-test teruggeven verplaatst `re-search-forward` point voorwaarts als zij-effect van een succesvolle zoekopdracht. Dit is de actie die de waarde van point verandert, zodat de recursieve functie stopt met zichzelf aanroepen wanneer point de beweging door de region voltooit. De `re-search-forward` expressie is dus de volgende-stap-expressie.

De body van `recursieve-tel-woorden` ziet er in overzicht zo uit:

```
(if doe-opnieuw-test-en-volgende-stap-expressie-gecombineerd
    ;; then
    recursieve-aanroep-die-telling-teruggeeft
  ;; else
  geef-nul-terug)
```

Hoe het mechanisme dat telt te incorporeren?

Dit kan lastig zijn wanneer je niet gewend bent recursieve functies te schrijven. Maar het kan en moet systematisch worden benaderd.

We weten dat het tel-mechanisme op de een of andere manier geassocieerd moet zijn met de recursieve aanroep. Omdat de volgende-stap-expressie point een woord voorwaarts verplaatst, en omdat de recursieve aanroep voor elk woord plaatsvindt, moet het tel-mechanisme een expressie zijn die een optelt bij de waarde teruggegeven door een aanroep van `recursive-count-words`.

Beschouw verschillende gevallen:

- Wanneer er twee woorden in de region zijn, moet de functie een waarde teruggeven die resulteert van het optellen van één bij de waarde die wordt teruggegeven wanneer die het eerste woord telt, plus het aantal teruggeven wanneer het de resterende woorden in de region telt, wat in dit geval één is.
- Wanneer er één woord in de region is, moet de functie een waarde teruggeven die resulteert van het optellen van één bij de waarde teruggeven van het tellen van dat woord, plus het aantal teruggeven wanneer het de resterende woorden in de region telt, wat in dit geval nul is.
- Wanneer er geen woorden in de region zijn moet de functie nul teruggeven.

Van de schets kunnen we zien dat het anders-deel van de `if` nul teruggeeft in het geval er geen woorden zijn. Dat betekent dat het dan-deel van de `if` een waarde moet teruggeven die resulteert van het optellen van een bij de waarde teruggegeven door het tellen van de resterende woorden.

De expressie ziet er zo uit, waarbij `1+` een functie is die één optelt bij zijn argument.

```
(1+ (recursieve-tel-woorden region-end))
```

De hele functie `recursieve-tel-woorden` ziet er dan zo uit:

```
(defun recursieve-tel-woorden (region-end))
  "documentatie..."

;;; 1. doe-opnieuw-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))

;;; 2. dan-deel: de recursieve aanroep
      (1+ (recursieve-tel-woorden region-end))

;;; 3. anders-deel
    0))
```

Laten we onderzoeken hoe dit werkt:

Wanneer er geen woorden in de region zijn, wordt het anders-deel van de `if` expressie geëvalueerd en geeft de functie dus nul terug.

Wanneer er één woord in de region is, is de waarde van point kleiner dan de waarde van `region-end` en de zoekopdracht slaagt. In dit geval test de waar-of-onwaar-test van de `if` expressie op waar, en wordt het dan-deel van de `if` expressie geëvalueerd. Deze expressie geeft een waarde terug (die wordt opgeteld bij de waarde teruggegeven door de hele functie) die de som is van één opgeteld bij de waarde teruggeven door een recursieve aanroep.

Ondertussen heeft de volgende-stap-expressie point over het eerste (in in dit geval enige) woord laten springen. Dit betekent dat wanneer `(recursieve-tel-woorden region-end)` een tweede keer wordt geëvalueerd, als gevolg van de recursieve aanroep, de waarde van point gelijk of groter is dan de waarde van het einde van de region. Dus deze keer geeft `recursieve-tel-woorden` nul terug. De nul wordt opgeteld bij één, en de oorspronkelijke evaluatie van `recursieve-tel-woorden` geeft een plus nul terug, wat één is, het correcte aantal.

Wanneer er twee woorden in de region zijn, geeft de eerste aanroep van `recursieve-tel-woorden` één opgeteld bij de waarde teruggeven door het aanroepen van `recursieve-tel-woorden` op een region die de resterende woorden bevat—wat inhoudt dat het één bij één optelt, dat twee geeft, wat het correcte aantal is.

Op dezelfde manier, wanneer er drie woorden in de region zijn, geeft de eerste aanroep van `recursieve-tel-woorden` één opgeteld bij de waarde teruggegeven door het aanroepen van `recursieve-tel-woorden` op een region die de resterende twee woorden bevat—enzovoorts.

Met de volledige documentatie zien de twee functies er zo uit:

De recusieve functie:

```
(defun recursieve-tel-woorden (region-end)
  "Aantal woorden tussen point and REGION-END."

;;; 1. doe-opnieuw-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))

;;; 2. dan-deel: de recursieve aanroep
      (1+ (recursieve-tel-woorden region-end))

;;; 3. anders-deel
    0))
```

De omliggende functie:

```
;;; Recursieve versie
(defun tel-woorden-voorbeeld (beginning end)
  "Toon aantal woorden in de region.

Woorden worden gedefinieerd als minstens een woord-vormend
karakter gevolgd door minstens een karakter dat
niet woord-vormend is. De syntax table van het buffer
bepaald welke karakters dat zijn."
  (interactive "r")
  (message "Tel woorden in region ... ")
  (save-excursion
    (goto-char beginning)
    (let ((count (recursieve-tel-woorden end)))
      (cond ((zerop count)
             (message
              "De region heeft GEEN woorden."))
            ((= 1 count)
             (message "De region heeft 1 woord."))
            (t
             (message
              "De region heeft %d woorden.." count))))))
```

13.3 Oefening: Leestekens tellen

Schrijf met een `while` loop een functie die het aantal leestekens in een region telt—punt, komma, punt-komma, dubbele punt, uitroepteken en vraagteken. Doe het nog een keer met recursie.

14 Woorden tellen in een `defun`

Ons volgende project is het tellen van het aantal woorden in een functiedefinitie. Het is duidelijk dat dit kan met een variant van `tel-woorden-voorbeeld`. Zie Hoofdstuk 13 "Tellen via repetitie en regexps", pagina 158. Wanneer we de woorden gaan tellen in éénn definitie, is het makkelijk genoeg om de definitie met het commando *C-M-h* (`mark-defun`) te markeren en dan `tel-woorden-voorbeeld` aan te roepen.

Ik ben echter meer ambitieus: ik wil de woorden en symbolen tellen in elke definitie in de Emacs broncode en dan een grafiek tonen die laat zien hoeveel functies er zijn van elke lengte: hoeveel bevatten 40 tot 49 woorden of symbolen, hoeveel bevatten 50 tot 59 symbolen, enzovoorts. Ik ben vaak nieuwsgierig hoe lang een typische functie is, en dit gaat het vertellen.

In één zin beschreven lijkt dit histogram project ontmoedigend, maar opgedeeld in talrijke kleine stappen, die we een voor een kunnen zetten wordt dit project minder angstaanjagend. Laten we overwegen wat de stappen moeten zijn:

- Ten eerste schrijf een functie die de woorden in één definitie telt. Dit bevat het probleem om zowel symbolen als woorden te behandelen.
- Ten tweede schrijf een functie die het aantal woorden in elke functie in een bestand opsomt. Deze functie kan de functie `tel-woorden-in-defun` gebruiken.
- Ten derde, schrijf een functie die het aantal woorden in elke functie in elk van een aantal bestanden opsomt. Dit bevat automatisch vinden van diverse bestanden, naar ze schakelen en de woorden in de definities in ze tellen.
- Ten vierde, schrijf een functie die de lijst met getallen die we in stap 3 hebben gemaakt omzet in een vorm die geschikt is om als grafiek te tonen.
- Ten vijfde, schrijf een functie die de resultaten als grafiek toont.

Dit is een behoorlijk project! Maar als we elke stap langzaam doen, is het niet moeilijk.

14.1 Wat te tellen?

Wanneer we beginnen met nadenken over hoe de woorden in een functiedefinitie te tellen, is de eerste vraag (of zou dat moeten zijn) wat gaan we tellen? Wanneer we over "woorden" spreken met betrekking tot een Lisp functiedefinitie, dan spreken we feitelijk grotendeels over symbolen. Bijvoorbeeld de volgende `vermenigvuldig-met-zeven` functie bevat vijf de symbolen `defun`, `vermenigvuldig-met-zeven`, `getal`, `*` en `7`. Daarnaast bevat het in de documentatiestring de vier woorden '`Vermenigvuldig`', '`GETAL`', '`met`' en '`zeven`'. Het woord '`getal`' wordt herhaald, zodat de definitie in totaal tien woorden en symbolen bevat.

```
(defun vermenigvuldig-met-zeven (getal)
  "Vermenigvuldig GETAL met zeven."
  (* 7 getal))
```

Als we echter de `vermenigvuldig-met-zeven` definitie met *C-M-h* (`mark-defun`) markeren en dan `tel-woorden-voorbeeld` er op aanroepen, dan zien we dat `tel-woorden-voorbeeld` zegt dat de definitie elf woorden bevat, niet tien! Er is duidelijk iets fout.

Het probleem is tweevoudig: `tel-woorden-voorbeeld` telt niet de '`*`' als woord en het telt het enkele symbool `vermenigvuldig-met-zeven` als drie woorden. De koppeltekens worden behandeld alsof het spaties tussen de woorden zijn in plaats van een verbinding tussen de woorden: '`vermenigvuldig-met-zeven`' wordt geteld alsof er '`vermenigvuldig met zeven`' staat.

De oorzaak van deze verwarring is de reguliere expressie zoekopdracht in de `tel-woorden-voorbeeld` definitie die point woord voor woord voorwaarts verpaatst. In de oorspronkelijke versie van `tel-woorden-voorbeeld` is de regexp:

```
"\\w+\\W*"
```

Deze reguliere expressie is een patroon dat een of meer woord-vormende letters mogelijk gevolgd door een of meer niet woord-vormende letters definieert. Wat bedoeld wordt met "woord-vormende letters" brengt ons bij het probleem van de syntax, wat een aparte sectie waard is.

14.2 Wat vormt een woord of symbool?

Emacs behandelt verschillende karakters als behorend bij verschillende *syntax categorieën*. De reguliere expressie '`\\w+`' bijvoorbeeld is ene patroon dat een of meer woord-vormende karakters specificeert. Woord-vormende karakters zijn leden van een syntax categorie. Andere syntax categorieën bevatten de klasse van leestekens, zoals de punt en komma, en de klasse witte ruimte karakters zoals de spatie en het tab-karakter. (Voor meer informatie zie Sectie "Syntax Tables" in *The GNU Emacs Lisp Reference Manual*.))

Syntax tabellen specificeren welke karakters tot welke categorie behoren. Meestal is een koppelteken niet gespecificeerd als een woord-vormend karakter. In plaats daarvan is het gespecificeerd als onderdeel van de klasse van karakters die deel zijn van symboolnamen maar niet woorden. Dit betekent dat de functie `tel-woorden-voorbeeld` het op dezelfde manier behandelt als het spaties tussen woorden behandelt en dat is waarom `tel-woorden-voorbeeld` '`vermenigvuldig-met-zeven`' als drie woorden telt.

Er zijn twee manieren die zorgen dat '`vermenigvuldig-met-zeven`' als één symbool telt: de syntax tabel aanpassen of de reguliere expressie aanpassen.

We kunnen een koppelteken als woord-vormend karakter herdefiniëren door de syntax tabel aan te passen die Emacs voor elke mode bijhoudt. Deze actie zou ons doel dienen, behalve dat een koppelteken slechts een veel gebruikt karakter binnen symbolen is dat niet typisch een woord-vormend karakter is, er zijn er ook andere.

We kunnen ook de regexp herdefiniëren die wordt gebruikt in de `tel-woorden-voorbeeld` zodat die ook symbolen bevat. Deze methode heeft de waarde van duidelijkheid maar die taak is een beetje lastig.

Het eerste deel eenvoudig genoeg: het patroon moet minstens één karakter matchen dat een woord of symbool vormt. Dus:

```
"\\(\\w\\|\\s_\\)+"
```

De '`\\(`' is het eerste van de groeperende constructie die de '`\\w`' en de '`\\s_`' als alternatieven bevat, geschieden door de '`\\|`'. De '`\\w`' matcht elk woord-vormend karakter en e '`\\s_`' matcht elke karakter dat onderdeel is van een symboolnaam maar geen woord-vormend karakter is. De '`+`' die op de groep volgt geeft aan dat het woord- of symboolvormend karakter tenminste eenmaal moet worden gematcht.x

Het tweede van de regexp is echter moeilijker te maken. Wat we willen is het eerste deel op te volgen met optioneel een of meer karakters die geen woord of symbool vormen. In eerste instantie dacht ik dat ik dit met het volgende kon doen:

```
"\\(\\W\\|\\S_\\)*"
```

De hoofdletters '`W`' en '`S`' matchen karakters die GEEN woord- of symbool vormen. Helaas matcht deze expressie elke karakter dat of niet woord-vormend of niet symbool-vormend is. Het matcht elk karakter!

Daarna merkte ik op dat elk woord of symbool in mijn test-region wordt gevold door witte ruimte (spatie, tab of nieuwe regel). Daarom probeerde een patroon dat een of meer spaties matcht achter het patroon voor een of meer woord- of symboolvormende karakters te plaatsten. Dit mislukte ook. Woorden en symbolen zijn vaak gescheiden door witte ruimte, maar in de code mogen haakjes op symbolen volgen en leestekens op woorden. Daarom maakte uiteindelijk ene patroon waarin woord- of symboolvormende karakters worden gevolgd door optionele karakters die geen witten ruimte zijn en dan optioneel gevold door witte ruimte.

Hier is de volledige reguliere expressie:

```
"\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*"
```

14.3 De `tel-woorden-in-defun` functie

We hebben gezien dat er verschillende manieren zijn om een functie `tel-woorden-region` te schrijven. Om een `tel-woorden-in-defun` te schrijven, hoeven we slechts een van die versies aan te passen.

De versie die een `while` loop gebruikt is makkelijk te begrijpen, daarom ga ik die aanpassen. Omdat `tel-woorden-in-defun` onderdeel wordt van een meer complex programma, hoeft het niet interactief te zijn en hoeft het geen boodschap te tonen, maar slecht de telling teruggeven. Deze overwegingen vereenvoudigen de definitie een beetje.

Anderzijds, `tel-woorden-in-defun` wordt gebruikt in een buffer dat functiedefinities bevat. Daarom is het redelijk te vragen dat de functie vaststelt of het wordt aangeroepen met point binnen een functiedefinitie, en zo ja, de telling voor die definitie terug te geven. Dit voegt complexiteit toe aan de definitie, maar redt ons van de noodzaak om argumenten aan de functie door te geven.

Deze overwegingen leiden ons tot het voorbereiden van het volgende sjabloon:

```
(defun tel-woorden-in-defun ()
  "documentatie..."
  (opzetten...
     (while loop...)
   geef telling terug)
```

Zoals gebruikelijk is het onze taak de slots in te vullen.

Eerst het opzetten

Wij gaan er van uit dat deze functie wordt aangeroepen binnen een buffer dat functiedefinities bevat. Point is in een functiedefinitie of niet. Om `tel-woorden-in-defun` te laten werken moet point naar het begin van de definitie verplaatsen, een teller moet bij nul beginnen en de tellende loop moet stoppen wanneer point het einde van de definitie bereikt.

De functie `beginning-of-defun` zoekt achterwaarts naar een openingsscheidingsteken zoals een '(' aan het begin van de regel en verplaatst point naar die positie of anders naar de begrenzing van de zoekopdracht. In de praktijk betekent dit dat `beginning-of-defun` pint naar het begin van een buitenliggende of voorgaande functiedefinitie verplaatst, of anders naar het begin van het buffer. De kunnen `beginning-of-defun` gebruiken om de point daar neer te zetten waar wij willen beginnen.

De `while` loop vereist een teller om het aantal getelde woorden of symbolen bij te houden. Een `let` expressie kan worden gebruikt om een lokale variabele voor dit doel te maken en te binnen aan een initiële waarde van nul.

De functie `end-of-defun` werkt net als de `beginning-of-defun` behalve dat het point naar het einde van de definitie verplaatst. `end-of-defun` kan worden gebruikt als onderdeel van een expressie die de positie van het einde van de definitie vaststelt.

Het opzetten van de `tel-woorden-in-defun` begint al snel vorm te krijgen: eerst verplaatsten we point naar het begin van de definitie, vervolgens maken we een lokale variable die telling bewaart en tenslotte registeren we de positie van het einde van de definitie zodat de `while` loop weet wanneer te stoppen met loopen.

De code ziet er zo uit:

```
(beginning-of-defun)
(let ((telling 0)
      (einde (save-excursion (end-of-defun) (point))))
```

De code is eenvoudig. De enige kleine complicatie zal waarschijnlijk betrekking hebben op `einde`: het is gebonden aan de positie van het einde van de definitie door een `save-excursion` expressie die de waarde van point na `end-of-defun` tijdelijk verplaatst naar het einde van de definitie.

Het tweede deel van de `tel-woorden-in-defun`, na het opzetten, is de `while` loop.

De loop moet een expressie bevatten die point woord voor woord en symbool voor symbool voorwaarts laat springen, en een andere expressie die de sprongen telt. De waar-of-onwaar-test voor de `while` loop moet op waar testen zolang als point voorwaarts moet springen, en op onwaar wanneer point aan het einde van de definitie is. We hebben hiervoor al een reguliere expressie aangepast, zodat de loop makkelijk is.

```
(while (and (< (point) einde)
            (re-search-forward
             "\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*" einde t))
  (setq telling (1+ telling)))
```

Het derde deel van de functiedefinitie geeft de telling van de woorden en symbolen terug. Dit deel is de laatste expressie in de body van de `let` en kan eenvoudigweg de lokale variabele `telling` zijn, die geëvalueerd de telling teruggeeft.

Samengevoegd ziet de `tel-woorden-in-defun` er zo uit:

```
(defun tel-woorden-in-defun ()
  "Geef het aantal woorden en symbolen in een defun terug."
  (beginning-of-defun)
  (let ((telling 0)
        (einde (save-excursion (end-of-defun) (point))))
    (while
        (and (< (point) einde)
             (re-search-forward
              "\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*"
              einde t))
      (setq telling (1+ telling)))
    telling))
```

Hoe dit te testen? De functie is niet interactief, maar het is makkelijk om een wrapper om de de functie te zetten die het interactief maakt. Wij kunnen bijna dezelfde code gebruiken als voor de recursieve versie van `tel-woorden-voorbeeld`:

```
;;; Interactieve versie.
(defun tel-woorden-defun ()
  "Aantal woorden en symbolen in een functiedefinitie."
  (interactive)
  (message
   "Tel woorden en symbolen in functiedefinitie ... ")
  (let ((telling (tel-woorden-in-defun)))
    (cond
     ((zerop telling)
      (message
       "The definitie heeft GEEN woorden of symbolen."))
     ((= 1 telling)
      (message
       "De definitie heeft 1 woord of symbool."))
     (t
      (message
       "The definitie heeft %d woorden of symbolen." telling)))))
```

Laten we *C-c =* hergebruiken als een handige keybinding:

```
(keymap-global-set "C-c =" 'tel-woorden-defun)
```

Nu kunnen we `tel-woorden-defun` uitproberen: installeer zowel `tel-woorden-in-defun` als `tel-woorden-defun` en zet de keybinding. Kopieer dat het volgende naar een Emacs Lisp buffer (zoals bijvoorbeeld `*scratch*`), plaats de cursor in de functie en gebruik het *C-c =* commando.

```
(defun vermenigvuldig-met-zeven (getal)
  "Vermenigvuldig GETAL met zeven."
  (* 7 getal))
    ⇒ 10
```

Succes! De definitie heeft 10 woorden en symbolen.

Het volgende probleem is om het aantal woorden en symbolen in meerdere definities in een enkel bestand te tellen.

14.4 Tel verschillende `defuns` in een bestand

Een bestand zoals `simple.el` kan honderd of meer functiedefinities bevatten. Ons langetermijndoel is om statistieken over veel bestanden de verzamelen, maar als eerste stap is ons huidige doel om de statistieken over een bestand te verzamelen.

De informatie bestaat uit een reeks getallen, elk getal is de lengte van een functiedefinitie.

We weten dat we de informatie met betrekking tot een bestand willen incorporeren in de informatie van veel andere bestanden. Dit betekent dat de functie voor het tellen van definitie-lengtes binnen een bestand alleen een lijst met de lengtes hoeft terug te geven. Het hoeft geen boodschappen te tonen en moet dat ook niet doen.

Het woord-tel commando bevat een expressie die point woord voor woord voorwaarts laat springen en een andere die de sprongen telt. De functie die de lengtes van de definities teruggeeft kan op een vergelijkbare manier gemaakt worden, met een expressie die point definitie voor definitie voorwaarts springt en een andere expressie die de lijst met lengtes construeert.

Deze formulering van het probleem maakt het eenvoudig om de functiedefinitie te schrijven. We beginnen de telling aan het begin van het bestand, dus het eerste commando wordt `(goto-char (point-min))`. Vervolgens starten we de `while` loop. De waar-of-onwaar-test van de loop kan een reguliere expressie zijn die zoekt naar de volgende functiedefinitie—zolang de zoekopdracht slaagt, wordt point voorwaarts verplaatst en de body van de loop geëvalueerd. De body heeft een expressie nodig die de lijst met lengtes maakt. `cons`, het lijst-constructie commando kan gebruikt worden om de lijst te maken. Dat is bijna alles wat er te zeggen is.

Hier is hoe dit codefragment er uit ziet:

```
(goto-char (point-min))
(while (re-search-forward "^(defun" nil t)
  (setq lengtes-lijst
        (cons (count-words-in-defun) lengtes-lijst)))
```

Wat we overgeslagen hebben is het mechanisme om het bestanden te vinden dat de functiedefinities bevat.

In de voorgaande voorbeelden gebruikten we of dit, het Info bestand, of we schakelden heen en weer naar een ander buffer, zoals het `*scratch*` buffer.

Een bestand vinden is een nieuw proces dat we nog niet eerder besproken hebben.

14.5 Een bestand vinden

Om in Emacs een bestand te vinden, gebruik je het *`C-x C-f`* (`find-file`) commando. Dit commando is bijna, maar niet helemaal geschikt voor het lengte probleem.

Laten we naar de broncode van `find-file` kijken:

```
(defun find-file (filename)
  "Edit file FILENAME.
Switch to a buffer visiting file FILENAME,
creating one if none already exists."
  (interactive "FFind file: ")
  (switch-to-buffer (find-file-noselect filename)))
```

(De meest recente versie van de `find-file` functiedefinitie staat je toe dat je optionele wildcards specificeert om meerder files te bezoeken. Dat maakt de definitie meer complex en dat bespreken we hier niet, omdat het niet relevant is. Je kunt de broncode zien met hetzij *M-.* (`xref-find-definitions`) or *C-h f* (`describe-function`).)

De definitie die ik laat zien bevat een korte maar complete documentatie en een interactieve specificatie die je naar een bestandsnaam vraagt wanneer je het commando interactief gebruikt. De body van de definitie bevat twee functies, `find-file-noselect` en `switch-to-buffer`.

Volgens de documentatiestring die getoond wordt met *C-h f* (het `describe-function` commando), leest de functie `find-file-noselect` het genoemde bestand in in een buffer en geeft het buffer terug. (De meest recente versie bevat eveneens een optioneel *wildcards* argument, naast een ander om het bestand letterlijk te lezen en een ander om foutmeldingen te onderdrukken. Deze optionele argumenten zijn irrelevant.)

De functie `find-file-noselect` selecteert echter geen buffer om het bestand in te stoppen. Emacs schakelt zijn aandacht (of de jouwe als je met `find-file-noselect`) niet naar het geselecteerde buffer. Dat is wat `switch-to-buffer` doet: het schakelt het buffer naar waar Emacs aandacht is gericht, en het schakelt het getoonde buffer in het window naar het nieuwe buffer. We hebben schakelen van buffers eerder besproken (Zie Sectie 2.3 “Van buffer verwisselen”, pagina 23.)

In het histogram-project hoeven we niet elk bestand op het scherm te tonen wanneer het programma de lengte van elke defintie er in vaststelt. In plaats van `switch-to-buffer` te gebruiken kunnen we werken met `set-buffer`, wat de aandacht van het computerprogramma naar een ander buffer herleidt maar vervangt niet het scherm. Daarom moeten we in plaats van `find-file` aan te roepen om het werk te doen onze expressie zelf schrijven.

De taak is makkelijk: gebruik `find-file-noselect` en `set-buffer`.

14.6 `lengte-lijst-bestand` in detail

De kern van de functie `lengte-lijst-bestand` is een `while` loop dat een functie bevat om point defun voor defun voorwaarts te verplaatsen en een functie om het aantal woorden en symbolen in elke defun te tellen. Deze kern moet worden omgeven door functie die verschillende andere taken doen, waaronder het bestand vinden en zorgen dat point start vanaf het begin van het bestand. De functiedefinitie ziet er zo uit:

```
(defun lengte-lijst-bestand (bestandsnaam)
  "Geef een lijst terug met definitielengtes in het bestand genaamd BESTANDSNAAM.
De teruggeven lijst is een lijst van getallen.
Elk getal is een het aahtal woorden of symbolen
in een functiedefinitie."
  (message "Werkt aan `%s' ... " bestandsnaam)
  (save-excursion
    (let ((buffer (find-file-noselect bestandsnaam))
          (lengte-lijst))
      (set-buffer buffer)
      (setq buffer-read-only t)
      (widen)
      (goto-char (point-min))
      (while (re-search-forward "^(defun" nil t)
        (setq lengte-lijst
              (cons (tel-woorden-in-defun) lengte-lijst)))
      (kill-buffer buffer)
      lengte-lijst)))
```

De functie krijgt één argument *bestandsnaam* doorgegeven, de naam van het bestand waarop het gaat werken. Het heeft vier regels documentatie maar geen interactieve specificatie. Omdat mensen bang zijn dat de computer kapot is wanneer ze niets zien gebeuren, bevat de eerste regel van de body een boodschap.

De volgende regel bevat een `save-excursion` die Emacs aandacht terugbrengt bij het huidige buffer wanneer de functie klaar is. Dit is nuttig wanneer je de functie in een andere functie inbed, die verwacht dat point wordt hersteld naar het oorspronkelijke buffer.

In de varlist van de `let` expressie vindt Emacs het bestand en bindt de lokale variabele `buffer` aan het buffer dat het bestand bevat. Tegelijkertijd maakt Emacs `lengte-lijst` als lokale variabele.

Vervolgens schakelt Emacs de aandacht naar het buffer.

In de volgende regel maakt Emacs het buffer read-only. In het ideale geval is deze regel niet nodig. Geen van de functies om woorden en symbolen te tellen zou het buffer moeten veranderen. En daarnaast wordt het buffer niet gesaved zelf wanneer het is gewijzigd. Deze regel is geheel het gevolg van de een grote, misschien overdreven, voorzichtigheid. De reden voor deze voorzichtigheid is dat de functie en de functies die het aanroept werken op de broncode van Emacs en het is ongemakkelijk wanneer die per ongeluk worden gewijzigd. Het spreekt voor zich dat ik de noodzaak voor deze regel niet realiseerde totdat een experiment misging en mijn Emacs broncode begon te wijzigen ...

Hierna komt de aanroep om het buffer te verbreden wanneer het is versmald. Deze functie is meestal niet nodig—Emacs maakt een vers buffer wanneer er niet al een bestaat, maar wanneer er al een buffer het bestand bezoekt kan het versmald zijn en moet het worden verbreed. Wanneer we volledig gebruikersvriendelijk willen zijn zouden we de restrictie en de lokatie van point opslaan, maar dat doen we niet.

De `(goto-char (point-min))` expressie verplaatst point naar het begin van het buffer.

Dan komt de `while` loop waarin het werk van de functie wordt uitgevoerd. In de loop stelt Emacs de lengte van elke definitie vast en construeert een lengte-lijst die de informatie bevat.

Na er doorheen gewerkt te hebben killt Emacs het buffer. Dit is om binnen Emacs ruimte te sparen. Mijn versie van GNU Emacs 19 bevat over 300 bronbestanden die interessant zijn. GNU Emacs 22 bevat meer dan duizend bronbestanden en Emacs 30.2 meer dan 1600. Een andere functie past `lengte-lijst-bestand` op elk bestand toe.

De laatste expressie in de `let` expressie tenslotte is de lengte-lijst variabele, wiens waarde wordt teruggegeven als de waarde van de gehele functie.

Dit kun je uitproberen door het op de gebruikelijke manier te installeren. Plaats daarna de cursor achter de volgende expressie en typ *C-x C-e* (`eval-last-sexp`).

```
(lengte-lijst-bestand
 "/usr/local/share/emacs/30.2/lisp/emacs-lisp/debug.el")
```

Misschien moet je de naam van het bestand aanpassen, deze hier is voor de standaard installatie tree van GNU Emacs versie 30.2. Om de expressie te wijzigen, kopieer je die naar het `*scratch*` buffer en wijzig het.

En om de volledige lengte van de lijst te zien, in plaats van een ingekorte versie, moet je misschien het volgende evalueren:

```
(custom-set-variables '(eval-expression-print-length nil))
```

(Zie Sectie 16.2 “Variabelen specificeren met `defcustom`”, pagina 201. Evalueer daarna de `lengte-lijst-bestand` expressie.)

Het maken van de lengte-lijst voor `debug.el` duurt minder dan een seconde en ziet er zo uit in GNU Emacs 30.2:

```
(79 26 140 34 17 112 81 24 155 54 43 102 21 36 36 117 28 29 102 49 43
 208 101 28 22 728 15 27)
```

(Op mijn oude machine duurde het maken van de versie 19 lengte-lijst van `debug.el` zeven seconden en ziet er zo uit:

```
(75 41 80 62 20 45 44 68 45 12 34 235)
```

De nieuwe versie van `debug.el` bevat meer defuns dan de eerdere, en mijn nieuwe machine is veel sneller dan de oude.)

Merk op dat de lengte van de laatste definitie in het bestand de eerste in de lijst is.

14.7 Woorden tellen in `defuns` in verschillende bestanden

In de vorige sectie hebben we een functie gemaakt die een lijst teruggeeft met de lengte van elke definitie in een bestand. Nu willen we een functie definieren de een hoofdlijst teruggeeft met de lengte van de definities in een lijst van bestanden.

Werken op elk bestand van een bestandenlijst is een herhalende actie, dus we kunnen een `while` loop of recursie gebruiken.

Het ontwerp met een `while` loop is routine. Het aan de functie doorgegeven argument is een lijst van bestanden. Zoals we eerder zagen (zie Sectie 11.1.1 “Loop voorbeeld”, pagina 117), kan je een `while` loop zo schijven dat de body van de loop wordt geëvalueerd zo’n lijst elementen bevat, maar de loop verlaat wanneer de lijst

leeg is. Om dit te laten werken moet de body van de loop een expressie bevatten die elke keer dat de body wordt geëvalueerd de lijst korter maakt, zodat uiteindelijk de lijst leeg is. De gebruikelijke techniek is om elke keer dat de body wordt geëvalueerd de waarde van de lijst op de waarde van de CDR van de lijst te zetten.

Het sjabloon ziet er zo uit:

```
(while test-of-lijst-leeg-is
  body...
  zet-lijst-op-cdr-van-lijst)
```

We herinneren ons ook dat een `while` loop `nil` teruggeeft (het resultaat van het evalueren van de waar-of-onwaar-test) en niet het resultaat van een evaluatie in de body. (De evaluaties in de body van de loop worden gedaan voor hun zij-effecten.) Echter de expressie die de lengte-lijst zet is onderdeel van de body—en dat is de waarde waarvan we willen dat de gehele functie teruggeeft. Om dit te doen plaatsen we de `while` loop binnen een `let` expressie, en zorgen dat het laatste element van de `let` expressie de waarde van de lengte-lijst bevat. (Zie “Loop voorbeeld met een ophogende teller.”, pagina 120.)

Deze overwegingen leiden on direct naar de functie zelf:

```
;;; Gebruik while loop.
(defun lengte-lijst-veel-bestanden (lijst-met-bestanden)
  "Return list of lengths of defuns in LIJST-MET-BESTANDEN."
  (let (lengte-lijst)

;;; waar-of-onwaar-test
    (while lijst-met-bestanden
      (setq lengtes-lijst
            (append
             lengtes-lijst

;;; Genereer een lengte-ljist.
              (lengtes-lijst-bestand
               (expand-file-name (car lijst-met-bestanden)))))

;;; Maak bestandenlijst korter.
      (setq lijst-met-bestanden (cdr lijst-met-bestanden)))

;;; Geef finale waarde lengtes-lijst terug.
    lengtes-lijst))
```

`expand-file-name` is een ingebouwde functie die een bestandsnaam omzet naar de absolute lange padnaam vorm. De functie gebruikt de naam van de directory waarin de functie is aangeroepen.

Dus wanneer `expand-file-name` wordt aangeroepen op `debug.el` terwijl Emacs de directory `/usr/local/share/emacs/22.1.1/lisp/emacs-lisp/` bezoekt,

```
debug.el
```

wordt

```
/usr/local/share/emacs/22.1.1/lisp/emacs-lisp/debug.el
```

Het enige andere nieuwe element van deze functie is de tot nu toe niet bestudeerde functie `append`, die op zichzelf een korte sectie waard is.

14.7.1 De functie `append`

De functie `append` maakt een lijst aan een andere vast. Dus,

```
(append '(1 2 3 4) '(5 6 7 8))
```

produceert de lijst

```
(1 2 3 4 5 6 7 8)
```

Dit is exact hoe we twee door `lengte-lijst-bestand` gemaakte lengte-lijsten aan elkaar willen vastmaken. Het resultaat contrasteert met `cons`,

```
(cons '(1 2 3 4) '(5 6 7 8))
```

die een nieuwe lijst construeert waarin het eerste argument van `cons` het eerste element van de nieuwe lijst wordt:

```
((1 2 3 4) 5 6 7 8)
```

14.8 Recursief woorden tellen in verschillende bestanden

Behalve met een `while` loop kan je met recursie werken op elk bestand in een lijst van bestanden. Een recursieve versie van (lengte-lijst-veel-bestanden) is kort en simpel.

De recursieve functie heeft de gebruikelijke delen: de doe-opnieuw-test, de volgende-stap-expressie en de recursieve aanroep. De doe-opnieuw-test stelt vat of de functie zichzelf opnieuw moet aanroepen, wat die doet wanneer de `lijst-met-bestanden` resterende elementen bevat. De volgende-stap-expressie zet de `lijst-met-bestanden` op de CDR van zichzelf, zodat uiteindelijk de lijst leeg wordt. en de recursieve aanroep roept zichzelf met de kortere lijst. De complete functie is korter dan deze beschrijving!

```
(defun recursieve-lengte-lijst-veel-bestanden (lijst-met-bestanden)
  "Geeft lijst met lengtes terug van elke defun in LIJST-MET-BESTANDEN."
  (if lijst-met-bestanden                    ; doe-opnieuw-test
      (append
       (lengte-lijst-bestand
        (expand-file-name (car lijst-met-bestanden)))
       (recursieve-lengte-lijst-veel-bestanden
        (cdr lijst-met-bestanden)))))
```

In één zin, de functie geeft de lengte-lijst terug voor de eerste van de `lijst-met-bestanden` vastgemaakt aan het resultaat van zichzelf aanroepen op de rest van de `lijst-met-bestanden`.

Hier is een test van `recursieve-lengte-lijst-veel-bestanden`, samen met de resultaten van het draaien van `lengte-lijst-bestand` op elk individueel bestand.

Installeer `recursieve-lengte-lijst-veel-bestanden` en indien nodig `lengte-lijst-bestand` en evalueer de volgende expressies. Je moet misschien de padnamen van bestanden aanpassen, deze werken hier wanneer dit Infobestand and de Emacs bronbestanden op hun gebruikelijke plaats staan. Om de expressies aan te passen kopieer je ze in het `*scratch*` buffer, pas je ze aan en evalueer je ze.

De resultaten worden getoond achter de '⇒'. (Deze resultaten zijn voor bestanden van Emacs versie 22.1.1. Bestanden van andere versies van Emacs kunnen andere resultaten produceren.)

```
(cd "/usr/local/share/emacs/22.1.1/")

(lengte-lijst-bestand "./lisp/macros.el")
     ⇒ (283 263 480 90)

(lengte-lijst-bestand "./lisp/mail/mailalias.el")
     ⇒ (38 32 29 95 178 180 321 218 324)

(lengte-lijst-bestand "./lisp/hex-util.el")
     ⇒ (82 71)

  (recursieve-lengte-lijst-veel-bestanden
   '("./lisp/macros.el"
     "./lisp/mail/mailalias.el"
     "./lisp/hex-util.el"))
       ⇒ (283 263 480 90 38 32 29 95 178 180 321 218 324 82 71)
```

De functie `recursieve-lengte-lijst-veel-bestanden` produceert de output die we willen.

De volgende stap is het voorbereiden van de data in de lijst om in een grafiek te tonen.

14.9 De data voorbereiden om in een grafiek te tonen

De functie `recursieve-lengte-lijst-veel-bestanden` geeft een lijst met getallen terug. Elk getal staat voor de lengte van een functiedefinitie. Wat we nu moeten doen is deze data omzetten in een lijst van getallen die geschikt is om een grafiek mee te genereren. De nieuwe lijst vertelt hoeveel functiedefinities bevatten minder dan 10 woorden en symbolen, hoeveel bevatten tussen 10 en 19 symbolen. enzovoorts.

In het kort, we moeten door de lengte-lijst heen gaan, die de functie `recursieve-lengte-lijst-veel-bestanden` heeft geproduceerd, en het aantal defuns in elke range van lengtes tellen, en een lijst van deze aantallen produceren.

Gebaseerd op wat we eerder gedaan hebben, kunnen we makkelijk voorzien dat het niet al te moeilijk moet zijn een functie te schrijven die CRD-t de lengte-lijst afloopt, vaststelt in welke lengte-range die in is, en een teller verhoogt voor die range.

Voordat we echter beginnen met het schrijven van zo'n functie, moeten we de voordelen van het sorteren van de lengte-lijst in ogenschouw nemen, zodat de getallen zijn gesorteerd van de kleinste tot de grootste. Ten eerste maakt het sorteren het eenvoudiger de getallen in elke range te tellen, omdat twee opeenvolgende getallen

in dezelfde range of aangrenzende ranges zijn. Ten tweede door de gesorteerde lijst te inspecteren kunnen we ontdekken wat het hoogste en het laagste getal zijn, en daarmee bepalen welke grootste en kleinste lengte-range we nodig hebben.

14.9.1 Lijsten sorteren

Emacs heeft een functie om lijsten te sorteren, met de naam (zoals je kunt raden) `sort`. De functie `sort` vereist twee argumenten, de lijst om te sorteren, en een predicaat dat vaststelt of het eerste van twee lijst-elementen kleiner is dan de tweede.

Zoals we eerder zagen (zie Sectie 1.8.4 "Gebruik van het verkeerde objecttype als argument", pagina 13), is een predicaat een functie die vaststelt of een bepaalde eigenschap waar of onwaar is. De functie `sort` herordent een lijst in overeenstemming met de eigenschap die het predicaat gebruikt. Dit betekent dat `sort` kan worden gebruikt om een niet-numerieke lijst te sorteren op niet-numerieke criteria—het kan bijvoorbeeld een lijst op alfabetische volgorde zetten.

De functie `<` wordt gebruikt om een numerieke lijst te sorteren. Bijvoorbeeld,

```
(sort '(4 8 21 17 33 7 21 7) '<)
```

produceert dit:

```
(4 7 7 8 17 21 21 33)
```

(Merk op dat in dit voorbeeld beide argumenten zijn gequote zodat de symbolen niet worden geëvalueerd voordat ze als argumenten aan de `sort` worden doorgegeven.)

Het sorteren van de lijst die teruggeven is door de functie `recursieve-lengte-lijst-veel-bestanden` is eenvoudig, het gebruikt de functie `<`:

```
(sort
 (recursieve-lengte-lijst-veel-bestanden
  '("./lisp/macros.el"
    "./lisp/mailalias.el"
    "./lisp/hex-util.el"))
 '<)
```

produceert:

```
(29 32 38 71 82 90 95 178 180 218 263 283 321 324 480)
```

(Merk op dat in dit voorbeeld het eerste argument van `sort` is niet gequote omdat de expressie moet worden geëvalueerd om de lijst te produceren die wordt doorgegeven aan `sort`.)

14.9.2 Een lijst van bestanden maken

De functie `recursieve-lengte-lijst-veel-bestanden` vereist een lijst van bestanden als argument. Voor onze testvoorbeelden construeerden we handmatig zo'n lijst, maar de Emacs brondirectory is te groot voor ons om dat te doen. In plaats daarvan schrijven we een functie die dat werk voor ons doet. In deze functie gebruiken we zowel een `while` loop en een recursieve aanroep.

Voor oudere versies van GNU Emacs hoefden we zo'n functie niet te schrijven, omdat zij alle '`.el`' bestanden in één directory plaatsten. In plaats daarvan konden we de functie `directory-files` gebruiken, die de namen van de bestanden in een enkele directory vermeldt die aan een specifiek patroon voldoen.

Recente versies van Emacs echter plaatsen Emacs Lisp bestanden in subdirectories van de top-level `lisp` directory. Deze herschikking maakt de navigatie makkelijker. Alle mail gerelateerde bestanden bijvoorbeeld staan in een `lisp` subdirectory met de naam `mail`. Maar tegelijkertijd forceert deze schikking ons om een bestandslisting functie te maken die afdaalt in de subdirectories.

We kunnen deze functie met de naam `bestanden-in-onderiggende-directory` maken met bekende functies zoals `car`, `nthcdr` en `substring` samen met een bestaande functie met de naam `directory-files-and-atributes`. Deze laatste functie vermeldt niet alleen de bestandsnamen in een directory inclusief de namen van de subdirectories, maar ook hun attributen.

Om ons doel opnieuw te formuleren: maak een functie die ons in staat stelt `recursieve-lengte-lijst-veel-bestanden` te voeden met de bestandsnamen als een lijst die er zo iut ziet (maar met meer elementen):

```
("./lisp/macros.el"
 "./lisp/mail/rmail.el"
 "./lisp/hex-util.el")
```

De functie `directory-files-and-atributes` geeft een lijst van lijsten terug. Elk van deze lijsten in de hoofdlijst bestaat uit 13 elementen. Het eerste element is een string die de bestandsnaam bevat—welke, in GNU/Linux, een *dicectory file* kan zijn, met andere woorden een bestand met de speciale attributen van een directory. Het tweede element van de lijst is `t` voor een directory, een string voor een symbolische link (de string is de naam waar naartoe gelinkt), or `nil`.

Bijvoorbeeld het eerste '`.el`' bestand in de `lisp/` directory is `abbrev.el`. Zijn naam is `/usr/local/share/emacs/22.1.1/lisp/abbrev.el` en het is niet een directory of een symbolische link.

Dit is hoe `directory-files-and-attributes` het bestand en zijn atributen vermeldt:

```
("abbrev.el"
nil
1
1000
100
(20615 27034 579989 697000)
(17905 55681 0 0)
(20615 26327 734791 805000)[1]
13188
"-rw-r--r--"
t
2971624
773)
```

[1] Wanneer `current-time-list` `nil` is, zijn de drie timestamps respectievelijk `(1351051674579989697 . 1000000000)`, `(1173477761000000000 . 1000000000)` en `(1351050967734791805 . 1000000000)`.

Anderzijds, `mail/` is een directory in de `lisp/` directory. Het begin van de listing ziet er zo uit:

```
("mail"
t
...
)
```

(Zie de documentatie van `file-attributes` om meer te leren over de verschillende atributen. Hou er rekening mee dat de functie `file-attributes` de bestandsnaam niet vermeldt, zodat het eerste element is het tweede element in `directory-files-and-atributes`.)

Wij willen dat onze nieuwe functie, `bestanden-in-onderiggende-directory` de '`.el`' bestanden vermeldt in de directory die het verteld is te bekijken, en in elke daar onderliggende directory.

Dit geeft ons een hint hoe de `bestanden-in-onderiggende-directory` binnen een directory te construeren, de functie zou '`.el`' bestandsnamen aan een lijst moeten toevoegen. En wanneer de functie binnen een directory op een subdirectory stuit, moet die de subdirectory ingaan en daar de acties herhalen.

We moeten echter opmerken dat elke directory een naam die naar zichzelf refereert bevat, `.` ("dot") genaamd, en een naam die refereert naar de bovenliggende directory, `..` ("dot dot") genaamd. (In `/`, de root-directory, refereert `..` aan zichzelf, omdat `/` geen bovenliggende directory heeft.) We willen natuurlijk niet dat onze functie `bestanden-in-onderiggende-directory` deze directories ingaat, omdat die ons altijd, direct of indirect, naar de huidige directory leiden.

Dus moet onze functie `bestanden-in-onderiggende-directory` verschillende taken doen:

- Controleer of het naar een bestandsnaam kijkt die eindigt in '`.el`' en zo ja, voeg thet aan de lijst toe.
- Controleer of het naar een bestandsnaam kijkt die de naam van een directory is, en zo ja,
  - Controleer of het kijkt naar `.` of `..`, en zo ja, sla die over.
  - Zo niet, ga dan in de directory en herhaal het proces.

Laten we een functiedefinitie schrijven die deze taken uitvoert. We gebruiken een `while` loop om binnen de directory van de ene bestandsnaam naar de ander te gaan en controleren wat moet worden gedaan. En we gebruiken een recursieve aanroep om de acties in elke subdirectory te herhalen. Het resursieve patroon is Accumuleren (zie "Accumuleren", pagina 137), met `append` als de combinerende functie.

Hier is de functie:

```
(defun bestanden-in-onderiggende-directory (directory)
  "Vermeldt de .el files in DIRECTORY en onderliggende subdirectories."
  ;; Hoewel de functie niet interactief gebruikt wordt,
  ;; is het makkelijker te testen wanneer we het interactief maken.
  ;; De directory zal een naam hebben zoals
  ;;  "/usr/local/share/emacs/22.1.1/lisp/"
  (interactive "DDirectory naam: ")
```

```
(let (el-bestanden-lijst
      (huidige-directory-lijst
       (directory-files-and-attributes directory t)))
  ;; zoals we in de huidige directory zijn
  (while huidige-directory-lijst
    (cond
     ;; check of de bestandsnaam eindigt met '.el'
     ;; en zo ja, voeg die toe aan de lijst.
     ((equal ".el" (substring (car (car huidige-directory-lijst)) -3))
      (setq el-bestanden-lijst
            (cons (car (car huidige-directory-lijst)) el-bestanden-lijst)))
     ;; check of bestandsnaam die van een directory is
     ((eq t (car (cdr (car huidige-directory-lijst))))
      ;; beslis tussen skippen of afdalen
      (if
          (equal "."
                 (substring (car (car huidige-directory-lijst)) -1))
          ;; dan doe niets omdat de bestandnaam die is van
          ;;   huidige directory of parent, "." or ".."
          ()
        ;; anders daal af in de directory en herhaalt het proces
        (setq el-bestanden-lijst
              (append
               (bestanden-in-onderiggende-directory
                (car (car huidige-directory-lijst)))
               el-bestanden-lijst)))))
    ;; ga naar volgende bestandnaam in de lijst; dit verkort ook
    ;; de lijst zodat de while loop uiteindelijk aan zijn eind komt
    (setq huidige-directory-lijst (cdr huidige-directory-lijst)))
  ;; geef de bestandsnamen terug
  el-bestanden-lijst))
```

De functie `bestanden-in-onderiggende-directory` vereist een argument, de naam van een directory.

Dus, op mijn systeem,

```
(length
 (bestanden-in-onderiggende-directory "/usr/local/share/emacs/22.1.1/lisp/"))
```

vertelt me dat er in en onder mijn Lisp brondirectory 1031 '`.el`' bestanden zijn.

`bestanden-in-onderiggende-directory` geeft een lijst terug in omgekeerde alfabetische volgorde. Een expressie om de lijst in alfabetische volgorde te sorteren ziet er zo uit:

```
(sort
 (bestanden-in-onderiggende-directory "/usr/local/share/emacs/22.1.1/lisp/")
 'string-lessp)
```

14.9.3 Functiedefinities tellen

Ons doel is een lijst te genereren die vertelt hoeveel functiedefinities minder dan 10 woorden en symbolen bevatten, hoeveel tussen de 10 en 19 woorden en symbolen bevatten, hoeveel tussen de 20 en 29 woorden en symbolen, enzovoorts.

Met een gesorteerde lijst met getallen is dit makkelijk. Tel hoeveel elementen in de lijst kleiner dan 10 zijn, dan na voorbij de net getelde getallen te gaan, tel hoeveel er kleiner dan 20 zijn, dan na voorbij de net getelde getallen te gaan, hoeveel kleiner zijn dan 30, enzovoorts. Elk van de getallen 10, 20, 30, 40 en dergelijke is een groter dan de hoogste van die range. We kunnen de lijst met zulke getallen de `hoogste-van-ranges` lijst noemen.

Als we dat willen kunnen we zo'n lijst automatisch genereren, maar het is simpeler de lijst handmatig te schrijven. Hier is die:

```
(defvar hoogste-van-ranges
 '(10  20  30  40  50
   60  70  80  90 100
  110 120 130 140 150
  160 170 180 190 200
  210 220 230 240 250
  260 270 280 290 300)
 "Lijst met ranges for `defuns-hoogste-per-range'.")
```

Om de ranges aan te passen, kunnen we deze lijst wijzigen.

Vervolgens moeten we een functie schrijven die een lijst maakt van de getallen van de definities binnen een range. Deze functie moet `gesorteerde-lengtes` en `hoogste-van-ranges` als argument gebruiken.

De functie `defuns-per-range` moet twee dingen steeds opnieuw doen: het moet het nummer van definities binnen een range tellen, gespecificeerd door de huidge `hoogste-van-ranges` waarde en het moet naar de eerstvolgende hogere waarde n de `hoogste-van-ranges` lijst opschuiven nadat het de definities in de huidige range geteld heeft. Omdat deze acties repeterend zijn, kunnen we een `while` loop voor de taak gebruiken. Een loop telt het aantal definities in de range gespecificeerd door de huidge hoogste-van-ranges waarde, en de andere loop selecteert elk van de hoogste-van-ranges op hun beurt.

Verschillende entries van de `gesorteerde-lengtes` lijst worden geteld voor elke range. Dit betekent dat de loop voor de `gesorteerde-lengtes` lijst binnen de loop

voor de `hoogste-van-ranges` lijst staat, zoals een klein tandwiel in een groot tandwiel.

De binnenste loop telt het aantal definities binnen een range. Het is een eenvoudige tellende loop van het type zoals we eerder zagen. (Zie Sectie 11.1.3 "Een loop met een incrementele teller", pagina 119.) De waar-of-onwaar-test van de loop test of de waarde van de `gesorteerde-lengtes` lijst kleiner is dan de huidige waarde van de hoogste-van-ranges. Indien dat zo is, dan verhoogt de functie de teller en test de volgende waarde van de `gesorteerde-lengtes` lijst.

De binnenste loop ziet er zo uit:

```
(while lengte-element-kleiner-dan-hoogste-van-range
  (setq getal-in-range (1+ getal-in-range))
  (setq gesorteerde-lengtes (cdr gesorteerde-lengtes)))
```

De buitenste loop moet starten met de laagste waarde in de `hoogste-van-ranges` lijst en daarna per keer steeds op de opvolgende hogere waarde gezet worden. Dit kan met een loop zoals deze:

```
(while hoogste-van-ranges
  body-van-loop...
  (setq hoogste-van-ranges (cdr hoogste-van-ranges)))
```

Samengevoegd zien de twee loops er zo uit:

```
(while hoogste-van-ranges

  ;; Tel het aantal elementen binnen de huidige range.
  (while lengte-element-kleiner-dan-hoogste-van-range
    (setq getal-in-range (1+ getal-in-range))
    (setq gesorteerde-lengtes (cdr gesorteerde-lengtes)))

  ;; Ga naar volgende range.
  (setq hoogste-van-ranges (cdr hoogste-van-ranges)))
```

Daarnaast moet Emacs in elk circuit van de buitenste loop het aantal definities binnen die range (de waarde van `getal-in-range`) in een lijst vastleggen. We kunnen `cons` voor dit doel gebruiken. (Zie Sectie 7.2 "`cons`", pagina 80.)

De functie `cons` werkt prima, behalve dat de lijst die het maakt het aantal definities voor de hoogste range aan het begin en het aantal definities voor de laagste range aan het einde bevat. Dit komt omdat `cons` nieuwe elementen aan het begin van de lijst vastmaakt en omdat de twee loops hun werk door de lengtes-lijst van laag tot hoog doen, zal `defuns-per-range-list` eindigen met het grootste getal aan het begin. Maar wij willen onze grafiek tonen met eerst de kleinste waarden en de grotere daarna. We kunnen dit doen met de functie `nreverse` die de volgorde van een lijst omdraait.

Bijvoorbeeld,

```
(nreverse '(1 2 3 4))
```

produceert:

```
(4 3 2 1)
```

Merk op dat de functie `nreverse` destructief is—dat wil zeggen, het verandert de lijst waarop het wordt toegepast. Dit contrasteert met de functies `car` en `cdr`, die niet-destructief zijn. In dit geval willen we de originele `defuns-per-range-lijst` niet, dus het maakt niet uit dat de lijst wordt vernietigd. (De functie `reverse` maakt een omgedraaide kopie van een lijst, en laat de originele lijst intact.)

Alles bij elkaar ziet de `defuns-per-range` er zo uit:

```
(defun defuns-per-range (gesorteerde-lengtes hoogste-van-ranges)
  "GESORTEERDE-LENGTES defuns in elke HOOGSTE-VAN-RANGES range."
  (let ((hoogste-van-range (car hoogste-van-ranges))
        (getal-in-range 0)
        defuns-per-range-lijst)

    ;; Buitenste loop.
    (while hoogste-van-ranges

      ;; Binnenste loop.
      (while (and
              ;; Getal nodig voor numerieke test.
              (car gesorteerde-lengtes)
              (< (car gesorteerde-lengtes) hoogste-van-range))


        ;; Tel aantal definities in huidige range.
        (setq getal-in-range (1+ getal-in-range))
        (setq gesorteerde-lengtes (cdr gesorteerde-lengtes)))

      ;; Verlaat binnenste loop maar blijf in buitenste loop.


      (setq defuns-per-range-lijst
            (cons getal-in-range defuns-per-range-lijst))
      (setq getal-in-range 0)       ; Reset telling op nul.

      ;; Ga naar volgende range.
      (setq hoogste-van-ranges (cdr hoogste-van-ranges))
      ;; Specify next top of range value.
      (setq hoogste-van-range (car hoogste-van-ranges)))

    ;; Verlaat buitenste loop en tel het aantal defuns groter dan
    ;;   de grootste top-of-range value.
    (setq defuns-per-range-lijst
          (cons
           (length gesorteerde-lengtes)
           defuns-per-range-lijst))


    ;; Geef een lijst terug met het aantal definities in elke range,
    ;;   van klein naar groot.
    (nreverse defuns-per-range-lijst)))
```

De functie is eenvoudig behalve een subtiele eigenschap. De waar-of-onwaar-test van de binnenste loop ziet er zo uit:

```
(and (car gesorteerde-lengtes)
     (< (car gesorteerde-lengtes) hogste-van-range))
```

in plaats van dit:

```
(< (car gesorteerde-lengtes) hoogste-van-range)
```

Het doel van deze test is vast te stellen of het eerste item in de `gesorteerde-lengtes` lijst kleiner is dan de waarde van de hoogste van de range.

De simpele versie van de tekst werkt prima tenzij de `gesorteerde-lengtes` lijst een waarde `nil` heeft. In dat geval geeft de `(car gesorteerde-lengtes)` expressie `nil` terug. De functie `<` kan geen getal vergelijken met `nil`, wat een lege lijst is, zodat Emacs een fout signaleert en de verdere uitvoering van de functie stopt.

De `gesorteerde-lengtes` lijst wordt altijd `nil` wanneer de teller het einde van de lijst bereikt. Dit betekent dat elke poging om de functie `defuns-per-range` met de simpele versie te gebruiken zal falen.

Wij lossen het probleem op met de `(car gesorteerde-lengtes)` expressie in combinatie met de `and` expressie. De `(car gesorteerde-lengtes)` expressie geeft een niet-`nil` waarde zolang de lijst tenminste een getal bevat, maar geeft `nil` terug als de lijst leeg is. De `and` expressie evalueert eerst de `(car gesorteerde-lengtes)` expressie, and als die `nil` is, geeft die onwaar terug *zonder* de `<` expressie te evalueren. Maar wanneer de `(car gesorteerde-lengtes)` expressie een non-`nil` waarde teruggeeft, evalueert de `and` expressie de `<` expressie en geeft die waarde terug als de waarde van de `and` expressie.

Op deze manier voorkomen we een fout.

Voor informatie over `and`, zie "De `kill-new` functie", pagina 98.)

Hier is een korte test van de functie `defuns-per-range`. Evalueer de expressie die een (verkorte) `hoogste-van-ranges` lijst bindt aan de lijst van waardes, en evalueer dan de expressie om de `gesorteerde-lengtes` lijst te binden, en evalueer daarna de functie `defuns-per-range`.

```
;; (Kortere lijst dan we later gaan gebruiken.)
(setq hoogste-van-ranges
 '(110 120 130 140 150
   160 170 180 190 200))

(setq gesorteerde-lengtes
      '(85 86 110 116 122 129 154 176 179 200 265 300 300))

(defuns-per-range gesorteerde-lengtes hoogste-van-ranges)
```

De teruggeven lijst ziet er zo uit:

```
(2 2 2 0 0 1 0 2 0 0 4)
```

Er zijn inderdaad twee elementen van de `gesorteerde-lengtes` lijst kleiner dan 110, twee elemten tussen 110 en 119, twee elementen tussen 120 en 129, enzovoorts. Er zijn vier elementen met een waarde van 200 of groter.

15 Een grafiek voorbereiden

Ons doel is een grafiek te construeren die de getallen toont van de functiedefinities van diverse lengtes in de Emacs lisp bronbestanden.

Als je in de praktijk een grafiek zou maken, zou je daarvoor waarschijnlijk een programma zoals `gnuplot` gebruiken. (`gnuplot` is mooi geïntegreerd en GNU Emacs.) In dit geval echter maken we er een helemaal zelf en in het proces maken wij ons weer vertrouwd met sommige dingen die we eerder geleerd hebben en meer leren.

In dit hoofdstuk schrijven we eerste een simpele functie om een grafiek te tonen. Deze eerste definitie wordt een *prototype*, een snel geschreven functie om het onbekende terrein van grafiek maken te verkennen. We ontdekken draken of ontdekken dat ze een mythe zijn. Na het terrein verkend te hebben voelen we ons meer zelfverzekerd en breiden de functie uit met automatische labeling van de assen.

Omdat Emacs gemaakt is om flexibel te zijn en met allerlei soorten terminals te werken, waaronder alleen-tekst terminals, moet de grafiek gemaakt worden met een typemachine-symbool. Een sterretje is goed genoeg. Wanneer we de functie om de grafiek te tonen uitbreiden, kunnen we van de keuze van het symbool een gebruikersoptie maken.

We kunnen deze functie `grafiek-body-tonen` noemen, het neemt de `getallenlijstn` als enige argument. Op dit moment geven we nog geen labels aan de grafiek, maar tonen alleen de body.

De functie `grafiek-body-tonen` voegt een verticale kolom van sterretjes in voor elk element van de `getallen-lijst`. De hoogte van elke lijn wordt bepaald door de waarde van dat element in de `getallen-lijst`.

Kolommen invoegen is een reperterende actie wat betekent dat deze functie met een `while` loop dan wel recursief geschreven kan worden.

Onze eerste uitdaging is om te ontdekken hoe een kolom van sterretjes te tonen. Meestal tonen we in Emacs horizontaal karakters op het scherm, regel voor regel, door te typen. We kunnen twee routes volgen: we schrijven onze eigen kolom-invoegfunctie of ontdekken of er al een in Emacs bestaat.

Om te ontdekken of er al een in Emacs is, gebruiken we het commando *`M-x apropos`*. Dit commando is vergelijkbaar met het commando *`C-h a`* (`command-apropos`), behalve dat de laatste alleen functies vindt dat commando's zijn. Het commando *`M-x apropos`* vermeldt alle symbolen die een reguliere expressie matchen, inclusief functies die niet interactief zijn.

Waar we naar zoeken is een commando dat kolommen toont of invoegt. Zeer waarschijnlijk bevat de naam van de functie het woord "print", het woord "insert" of het woord "column". Daarom kunnen we eenvoudigweg *`M-x apropos`* `RET` *`print\|insert\|column`* `RET` typen en naar het resultaat kijken. Op mijn systeem kostte dit commando ooit behoorlijk wat tijd en produceerde een lijst met 79 functies en variabelen. Nu kost het amper tijd en produceert een lijst met 211 functies en variabelen. Bij het bekijken van de lijst is de enige functie die lijkt alsof die het werk kan doen is `insert-rectangle`.

Inderdaad, dit is de functie die we willen, de documentatie zegt:

```
insert-rectangle:
Insert text of RECTANGLE with upper left corner at point.
RECTANGLE's first line is inserted at point,
its second line is inserted at a point vertically under point, etc.
RECTANGLE should be a list of strings.
After this command, the mark is at the upper left corner
and point is at the lower right corner.
```

We doen een snelle test om zeker te zijn dat het doet wat we verwachten.

Hier is het resultaat van het plaatsen van de cursor achter de `insert-rectangle` expressie en *C-u C-x C-e* (`eval-last-sexp`) te typen. De functie voegt de strings '`"eerste"`', '`"tweede"`', en '`"derde"`' bij en onder point. Ook geeft de functie `nil` terug.

```
(insert-rectangle '("eerste" "tweede" "derde"))eerste
                                               tweede
                                               derdenil
```

We gaan natuurlijk niet zelf de tekst van de `insert-rectangle` expressie zelf in het buffer invoegen wanneer we de grafie maken, maar roepen de functie aan vanuit ons programma. We moeten echter zeker weten dat point in het buffer op de plaats is waar `insert-rectangle` de kolom invoegt.

Wanneer je dit in Info leest kan je zien hoe dit werkt door naar een ander buffer te schakelen, zoals het `*scratch*` buffer, en point ergens in het buffer te plaatsen. Typ *M-:* en typ de `insert-rectangle` expressie achter de prompt in het minibuffer gevolgd door `RET`. Dit laat Emacs de expressie in het minibuffer evalueren, maar als waarde van point de positie van point in het `*scratch*` buffer gebruiken. (*M-:* is de keybinding voor `eval-expression`. `nil` verschijnt niet in het `*scratch*` buffer omdat de expressie wordt geëvalueerd in het minibuffer.

We ontdekken dat als we dit doen, point aan het einde van de laatst ingevoegde regel staat—met andere woorden, deze functie verplaatst point als zij-effect. Wanneer we het commando herhalen, met point op deze plaats, komt de volgende invoeging onder en rechts van de vorige. Dit willen we niet! Wanneer we een staafgrafiek gaan maken moeten de kolommen naast elkaar komen.

We hebben dus ontdekt dat elke cyclus van de kolom-invoegende `while` loop point moet herpositioneren op de plek waar we die willen, en die plek is aan de bovenkant, niet de onderkant, van de kolom. Bovendien bedenken we dat wanneer we een grafiek tonen, we niet verwachten dat alle kolommen dezelfde hoogte hebben. Dit betekent dat de bovenkant van elke kolom op aan andere hoogte kan zijn dan die van de vorige. We kunnen niet simpel point elke keer naar rechts op dezelfde regel herpositioneren—maar misschien kunnen we. . .

Wij zijn van plan om de kolommen van de grafiek van sterretjes te maken. Het aantal sterretjes in de kolom is het getal dat gespecificeerd wordt door het huidige element in de `getallen-lijst`. We moeten een lijst van sterretjes van de juist lengte maken voor elke aanroep van `insert-rectangle`. Wanneer deze lijst uitsluitend uit het aantal vereiste sterretjes bestaat, dan moeten point precies op het juiste aantal regels boven de bases van de grafiek positioneren. Dit kan moeilijk zijn.

Of, als we een manier kunnen vinden om aan `insert-rectangle` steeds een lijst van dezelfde lengte door te geven, dan kunnen we point steeds op dezelfde regel plaatsen, en het voor elke nieuwe kolom een stapje naar rechts zetten. Wanneer we dit doen moeten echter sommige entries in de aan `insert-rectangle` doorgegeven lijst leeg zijn in plaats van sterretjes. Wanneer bijvoorbeeld de maximum hoogte van de grafiek 5 is, maar de hoogte van de kolom 3 is, dan vereist `insert-rectangle` een argument dat er als volgt uitziet:

```
(" " " " "*" "*" "*")
```

Dit laatste voorstel is niet zo moeilijk, zolang we de kolomhoogte kunnen bepalen. Er zijn twee manieren voor ons om de kolomhoogte te bepalen: we kunnen willekeurig zeggen wat die is, wat goed zou werken voor grafieken van die hoogte, of we kunnen door de lijst met getallen zoeken en de maximale hoogte van de lijst als maximum hoogte van de grafiek te gebruiken. Als de laatste operatie moeilijk zou zin, dan is de eerste het makkelijkst, maar er is een functie in Emacs ingebouwd die het maximum van de argumenten vaststelt. We kunnen deze functie gebruiken. De functie heeft `max` en geeft de grootste van alle argumenten terug, welke getallen moeten zijn. Dus, bijvoorbeeld:

```
(max  3 4 6 5 7 3)
```

geeft 7 terug. (Een overeenkomstige functie genaamd `min` geeft de kleinste waarde van alle argumenten terug.)

We kunnen echter niet simpel `max` aanroepen op de `getallen-lijst`. De functie `max` verwacht getallen als argument, niet een lijst met getallen. Dus, de volgende expressie,

```
(max  '(3 4 6 5 7 3))
```

geeft de volgende foutboodschap:

```
Wrong type of argument:  number-or-marker-p, (3 4 6 5 7 3)
```

We hebben een functie nodig die een lijst van argumenten aan een functie doorgeeft. Deze functie is `apply`. Deze functie past het eerste argument (een functie) toe op de resterende argumenten, waarvan de laatste een lijst mag zijn.

Bijvoorbeeld,

```
(apply 'max 3 4 7 3 '(4 8 5))
```

geeft 8 terug.

(Overigens weet ik niet hoe je deze functie ontdekt zonder een boek als dit. Het is mogelijk andere functies te ontdekken, zoals `search-forward` of `insert-rectangle` door een deel van hun naam te raden en dan `apropos` te gebruiken. Alhoewel in basis de metafoor duidelijk is—past het eerste argument toe op de rest—betwijfel ik of een beginner op dit specifieke woord komt bij het gebruik van `apropos` of een ander hulpmiddel. Ik kan natuurlijk fout zitten, tenslotte heeft de persoon die het uitgevonden heeft het zo genoemd.)

Het tweede en volgende argumenten van `apply` zijn optioneel, we kunnen dus `apply` gebruiken om een functie aan te roepen en de elementen van een lijst laten doorgeven, zoals dit, wat ook 8 teruggeeft:

```
(apply 'max '(4 8 5))
```

Dit laatste is hoe wij `apply` gaan gebruiken. De functie `recursieve-lengte-lijst-veel-bestanden` geeft een lijst met getallen terug waarop we `max` kunnen

toepassen (we kunnen ook de `max` op de gesorteerde getallenlijst toepassen, het maakt niet uit of de lijst is gesorteerd of niet).

Dus de operatie voor het vinden van de maximum hoogte van de grafiek is deze:

```
(setq max-grafiek-hoogte (apply 'max getallen-lijst))
```

Nu kunnen we terugkeren naar de vraag hoe een lijst van strings te maken voor een kolom van de grafiek. Wanneer het de maximum hoogte van de grafiek en het aantal sterretjes dat in de kolom moet verschijnen verteld wordt, moet de functie een lijst van strings teruggeven om die door het commando `insert-rectangle` te laten invoegen.

Elke kolom wordt gemaakt van sterretjes of spaties. Omdat de functie de waarde van de hoogte van de kolom en het aantal sterretjes krijgt doorgegeven, kan het aantal spaties berekend worden door het aantal sterretjes af te trekken van de hoogte van de kolom. Met het aantal spaties en het aantal sterretjes kunnen twee `while` loops gebruikt worden om de lijst te maken:

```
;;; Eerste versie.
(defun kolom-van-grafiek (max-grafiek-hoogte actuele-hoogte)
  "Geeft een lijst van strings terug die een kolom in de grafiek is."
  (let ((invoeglijst nil)
        (aantal-bovenste-spaties
         (- max-grafiek-hoogte actuele-hoogte)))

    ;; Sterretjes invullen.
    (while (> actual-height 0)
      (setq invoeglijst (cons "*" invoeglijst))
      (setq actuele-hoogte (1- actuele-hoogte)))

    ;; Fill in blanks.
    (while (> number-of-top-blanks 0)
      (setq insert-list (cons " " insert-list))
      (setq number-of-top-blanks
            (1- number-of-top-blanks)))

    ;; Geef hele lijst terug.
    invoeglijst))
```

Wanneer je deze functie installeert en dan de volgende expressie evalueert zie je dat het een lijst zoals gewenst teruggeeft:

```
(kolom-van-grafiek 5 3)
```

geeft terug

```
(" " " " "*" "*" "*")
```

Zoals `kolom-van-grafiek` geschreven is bevat die een groot gebrek: de symbolen voor de lege ruimte en voor de gemarkeerde entries in de kolom zijn hard gecodeerd als spatie en sterretje. Dit is prima voor een prototype maar jij, of een andere gebruiker, kan andere symbolen symbolen willen. Bijvoorbeeld bij het testen van de grafiek functie zou je een punt in plaats van een spatie willen, om zeker te zijn dat point juist gepositioneerd is elke keer dat de functie `insert-rectangle` wordt aangeroepen, of je zou liever het sterretje door een '`+`' of ander symbool willen vervangen. Je zou zelfs een grafiek-kolom willen maken die breder is dan een karakter. Het programma moet flexibeler zijn. De manier om dat te doen is het

vervangen van de spatie en het sterretje met twee variabelen die we `grafiek-wit` en `grafiek-symbool` kunnen noemen en deze variabelen apart definieren.

Ook is de documentatie niet goed geschreven. Deze overwegingen leiden ons naar de tweede versie van de functie:

```
(defvar grafiek-symbool "*"
  "String gebruikt als symbool in grafiek, meestal een sterretje.")

(defvar grafiek-wit " "
  "String gebruikt als lege ruimte in grafiek, meestal een spatie.
grafiek-wit moet even breed zijn als grafiek-symbool.")
```

(Voor een uitleg van `defvar`, zie Sectie 8.5 “Initialiseer een variabele met `defvar`”, pagina 104.)

```
;;; Tweede versie.
(defun kolom-van-grafiek (max-grafiek-hoogte actuele-hoogte)
  "Geef MAX-GRAFIEK-HOOGTE strings; ACTUELE-HOOGTE zijn grafiek-symbolen.

De grafieksymbolen zijn aaneengesloten entries aan het eind
van de lijst.
De lijst wordt ingevoegd als een kolom van een grafiek.
De strings zijn ofwel grafiek-wit of grafiek-symbool."

  (let ((invoeglijst nil)
        (aantal-bovenste-spaties
         (- max-grafiek-hoogte actuele-hoogte)))

    ;; grafiek-symbolen invullen.
    (while (> actuele-hoogte 0)
      (setq invoeglijst (cons grafiek-symbool invoeglijst))
      (setq actuele-hoogte (1- actuele-hoogte)))

    ;; grafiek-wit invullen.
    (while (> aantal-bovenste-spaties 0)
      (setq invoeglijst (cons grafiek-wit invoeglijst))
      (setq aantal-bovenste-spaties
            (1- aantal-bovenste-spaties)))

    ;; Geef hele lijst terug.
    invoeglijst))
```

Wanneer we dat zouden willen, kunnen de `kolom-van-grafiek` een derde keer herschrijven om optioneel zowel een lijngrafiek als een staafgrafiek te tonen. Dit is niet moeilijk om te doen. Een manier om over een lijngrafiek te denken is dat het niet meer is dan een staafgrafiek waarbij het deel van elke staaf dat onder de bovenkant is, leeg is. Om een kolom voor een lijngrafiek te maken, maken we eerst een lijst van spaties die één korter is dan de waarde, dan gebruikt het een `cons` om een grafiek-symbool aan de lijst te plakken, en dan gebruikt het `cons` nog een keer om de bovenste-spaties aan de lijst te plakken.

Het is makkelijk te zien hoe zo’n functie te schrijven, maar omdat we die niet nodig hebben, doen we dat niet. Maar het is te doen en als het gedaan word, dan met `kolom-van-grafiek`. Nog belangrijker is het waard op te merken dat een paar

andere verandering moeten worden gemaakt. De uitbreiding, als we die ooit willen maken, is eenvoudig.

Nu komen we uiteindelijk bij onze eerste grafiek tonen functie. Deze toont de body van de grafiek en niet de labels voor de verticale en horizontale assen, daarom noemen we deze `grafiek-body-tonen`.

15.1 De functie `grafiek-body-tonen`

Na onze voorbereiding in de voorgaande sectie, is de functie `grafiek-body-tonen` eenvoudig. De functie toont kolom na kolom van sterretjes en spaties, met de elementen van een lijst met nummers om het aantal sterretjes in elke kolom te specificeren.

De functie `grafiek-body-tonen` vereist de hoogte van de grafiek als een argument, dus die moeten we bepalen en registreren als een lokale variabele.

Dit leidt ons tot het volgende sjabloon voor de `while` loop versie van deze functie:

```
(defun grafiek-body-tonen (getallenlijst)
  "documentatie..."
  (let ((hoogte  ...
         ...))

    (while getallenlijst
      voeg-kolom-in-en-herpostioneer-point
      (setq getallenlijst (cdr getallenlijst)))))
```

We moeten de slots in het sjabloon invullen.

We kunnen de `apply 'max getallenlijst` expressie gebruiken om de hoogte van de grafiek te bepalen.

De `while` loop loopt door de `getallenlijst` een element per keer. Wanneer het wordt verkort door de `(setq getallenlijst (cdr getallenlijst))` expressie, is de `car` van elke instantie van de lijst de waarde van het argument voor `kolom-van-grafiek`.

Bij iedere cyclus van de `while` loop voegt de functie `insert-rectangle` de lijst in die teruggeven is door `kolom-van-grafiek`. Omdat de functie `insert-rectangle` point verplaatst naar rechtsonder het ingevoegde vierkant, moeten we de lokatie van point op het moment dat het vierkant wordt ingevoegd bewaren en dan teruggaan naar die positie nadat het vierkant is ingevoegd en dan hozizontaal verplaatsen naar de volgende plek waar `insert-rectangle` wordt aangeroepen.

Wanneer de ingevoegde kolommen één karakter breed zijn, wat zo is wanneer enkele spaties en sterretjes worden gebruikt, dan is het herpositionering commando simpel `(forward-char 1)`. De breedte van een kolom kan echter groter zijn dan één. Dit betekent dat het herpositionering commando `(forward-char symbool-breedte)` moet zijn. De `symbool-breedte` zelf is de lengte van een `grafiek-wit` en kan bepaald worden met de expressie `(length grafiek-wit)`. De beste plek om de `symbool-breedte` variabele aan de waarde van de breedte van grafiek-kolom te binden is in de varlist van de `let` expressie.

Deze overwegingen leiden tot de volgende functiedefinitie:

```
(defun grafiek-body-tonen (getallenlijst)
  "Toon een staafgrafiek van de getallenlijst.
De getallenlijst bestaat uit de Y-as waarden."

  (let ((hoogte (apply 'max getallenlijst))
        (symboolbreedte (length grafiek-wit))
        van-positie)

    (while getallenlijst
      (setq van-positie (point))
      (insert-rectangle
       (kolom-van-grafiek hoogte (car getallenlijst)))
      (goto-char van-positie)
      (forward-char symboolbreedte)
      ;; Teken grafiek kolom voor kolom.
      (sit-for 0)
      (setq getallenlijst (cdr getallenlijst)))
    ;; Plaats point voor X as labels.
    (forward-line hoogte)
    (insert "\n")
))
```

De enige onverwachte expressie in deze functie is de `(sit-for 0)` expressie in de `while` loop. Deze expressie maakt de operatie die de grafiek toont interessanter om te zien dan zonder. Deze expressie zorgt dat Emacs *sit*, of doet niets voor een nul lengte van tijd en herschrijf dan het scherm. Hier geplaatst zorgt het dat Emacs het scherm kolom voor kolom herschrijft. Zonder dit zou Emacs het scherm niet herschrijven tot de functie eindigt.

We kunnen de `grafiek-body-tonen` testen met een korte lijst met getallen.

1. Installeer `grafiek-symbool`, `grafiek-wit`, `kolom-van-grafiek` die allemaal in Hoofdstuk 15 “Een grafiek voorbereiden”, pagina 191, staan en `grafiek-body-tonen`,
2. Kopieer de volgende expressie:

   ```
   (grafiek-body-tonen '(1 2 3 4 6 4 3 5 7 6 5 2 3))
   ```

3. Schakel naar het `*scratch*` buffer en plaats de cursor daar waar je de grafiek wilt laten beginnen.
4. Typ *M-:* (`eval-expression`).
5. Yank de `grafiek-body-tonen` expressie in het minibuffer met *C-y* (`yank`).
6. Druk op `RET` om de `grafiek-body-tonen` expressie te evalueren.

Emacs toont een grafiek zoals deze:

```
                *
              *   **
              *  ****
             *** ****
            ********* *
           ************
          *************
```

15.2 De functie recursieve-grafiek-body-tonen

De functie `grafiek-body-tonen` kan ook recursief geschreven worden. De recursieve oplossing is gesplitst in twee delen: een buitenste wrapper die een `let` expressie gebruikt om de waarde van diverse variabelen te bepalen die maar een keer hoeven worden te gevonden, zoals de maximum hoogte van de grafiek, en een binnenste functie die recursief aangeroepen wordt om de grafiek te tonen.

De wrapper is ongecompliceerd:

```
(defun recursieve-grafiek-body-tonen (getallenlijst)
  "Print a bar graph of the GETALLENLIJST.
The getallenlijst consists of the Y-axis values."
  (let ((hoogte (apply 'max getallenlijst))
        (symboolbreedte (length grafiek-wit))
        from-position)
    (recursieve-grafiek-body-tonen-intern
     getallenlijst
     hoogte
     symboolbreedte)))
```

De recursieve functie is iets moeilijker. Het heeft vier delen: de doe-opnieuw-test, de tonende code, de resursieve aanroep en de volgende-stap-expressie. De doe-opnieuw-test is een `when` expressie die bepaalt of `getallenlijst` resterende elementen bevat. Wanneer dat zo is, dan toont de functie een kolom van de grafiek met de tonende code en roept zichzelf opnieuw aan. De functie roept zichzelf opnieuw aan met de waarde geproduceerd door de volgende-stap-expressie die zorgt dat de aanroep op een kortere versie van de `getallenlijst` acteert.

```
(defun recursieve-grafiek-body-tonen-intern
  (getallenlijst hoogte symboolbreedte)
  "Print a bar graph.
Gebruikt in de functie recursieve-grafiek-body-tonen."

  (when getallenlijst
        (setq van-positie (point))
        (insert-rectangle
         (kolom-van-grafiek hoogte (car getallenlijst)))
        (goto-char van-positie)
        (forward-char symboolbreedte)
        (sit-for 0)     ; Draw graph column by column.
        (recursieve-grafiek-body-tonen-intern
         (cdr getallenlijst) hoogte symboolbreedte)))
```

Na installatie kan deze expressie getest worden, hier is een voobeeld:

```
(recursieve-grafiek-body-tonen '(3 2 5 6 7 5 3 4 6 4 3 2 1))
```

Hier is wat de `recursieve-grafiek-body-tonen` produceert:

```
                *
               **   *
              ****  *
              **** ***
             * *********
             ************
             *************
```

Beide functies, `grafiek-body-tonen` en `recursieve-grafiek-body-tonen` maken de body van de grafiek.

15.3 Noodzaak voor getoonde assen

Een grafiek moet assen hebben zodat je jezelf kunt oriënteren. Voor een eenmalig project kan het redelijk zijn de assen met de hand te tekenen met de Picture mode van Emacs. Maar een grafiek tekenende functie kan meer dan eens gebruikt worden.

Om deze reden heb ik een uitbreiding geschreven op de basis `grafiek-body-tonen` functie die automatisch labels toont voor de horizontale en vertikale assen. Omdat de label-toon functies niet veel nieuw materiaal bevatten, heb ik hun beschrijving in een appendix geplaatst. Zie Appendix C “Een grafiek met labels op de assen”, pagina 239.

15.4 Oefening

Schrijf een lijngrafiek versie van de grafiek tonende functies.

16 Jouw .emacs bestand

"'Je hoeft Emacs niet leuk te vinden om het leuk te vinden"'—deze schijnbaar paradoxale uitspraak is het geheim van GNU Emacs. De gewone, out-of-the-box Emacs is een generieke tool. De meeste mensen die het gebruiken passen het aan naar hun eigen keuze.

GNU Emacs is voornamelijk geschreven in Emacs Lisp. Dit betekent dat je door het schrijven van expressies in Emacs Lisp Emacs kan veranderen en uitbreiden.

Er zijn mensen die de Emacs standaard configuratie waarderen. Emacs start tenslotte in C mode wanneer je een C-bestand edit, start in Fortran mode wanneer je een Fortran-bestand edit en start in Fundamental mode wanneer je een platte tekst bestand edit. Dit is allemaal logisch wanneer je niet weet wie Emacs gaat gebruiken. Wie weet wat iemand hoopt te doen met een platte tekst bestand? De Fundamental mode is de juiste stndaard voor zo'n bestand, net als C mode de juiste standaard is voor het editten van C code. (Genoeg programmeertalen hebben een syntax die het mogelijk maakt eigenschappen te delen of bijna te delen, waardoor C mode nu door CC mode wordt geleverd, de C Collectie.)

Maar wanneer je weet wie Emacs gaat gebruiken—jij, jezelf—dan is het logisch Emacs aan te passen.

Ik gebruik bijvoorbeeld zelden Fundamental mode wanneer ik een platte tekst bestand edit, dan wil ik Text mode. Dat is waarom ik Emacs aanpas: zo past het goed bij me.

Je kunt Emacs aanpassen en uitbreiden door een `~/.emacs` te maken of aan te passen. Dit is je persoonlijke initialisatiebestand. De inhoud, geschreven in Emacs Lisp, vertelt Emacs wat te doen.[1]

Een `~/.emacs` bestand bevat Emacs Lisp code. Je kunt deze code zelf schrijven of je gebruikt de `customize` eigenschap van Emacs en laat de code voor je schrijven. Je kunt je eigen expressies en de automatisch geschreven Customize expressies in je `.emacs` bestand combineren.

(Ik geef er zelf voorkeur aan mijn eigen expressies te schrijven, behalve die makkelijker te manipuleren zijn met het `customize` commando, zoals fonts. Ik combineer de twee methoden.)

Het grootste deel van dit hoofdstuk gaat over het zelf expressies schrijven. Het beschrijft een eenvoudig `.emacs` bestand, voor meer informatie zie Sectie "The Init File" in *The GNU Emacs Manual*, en Sectie "The Init File" in *The GNU Emacs Lisp Reference Manual*.

[1] Je kunt ook `.el` aan `~/.emacs` toevoegen en het `~/.emacs.el` noemen. In het verleden mocht je niet de extra toetsaanslagen typen die de bestandnaam `~/.emacs.el` vereist, maar nu mag dat. Het nieuwe formaat is consistent met de Emacs Lisp naamgeving conventies, het oude formaat spaart typen.

16.1 Systeembrede initialisatiebestanden

Naast je persoonlijke initialisatiebestand laadt Emacs automatisch verschillende systeembrede initialisatiebestanden, als die bestaan. Deze hebben dezelfde vorm als je `.emacs` bestand, maar worden door iedereen geladen.

Twee systeembrede initialisatiebestanden, `site-load.el` and `site-init.el`, worden in Emacs geleden en dan gedumpt wanneer een gedumpte versie van Emacs wordt gemaakt, wat het meest gebruikelijk is. (Gedumpte exemplaren van Emacs laden sneller. Wanneer echter een bestand is geladen en gedumpt, leidt een verandering er van niet tot een verandering in Emacs, tenzij je het zelf laadt en opnieuw Emacs dumpt. Zie Sectie "Building Emacs" in *The GNU Emacs Lisp Reference Manual*, en het `INSTALL` bestand.).

Drie andere systeembrede initialisatiebestanden worden elke keer dat je Emacs start automatisch geladen, wanneer die bestaan. Dit zijn `site-start.el`, die *voorafgaand* aan je `.emacs` bestand geladen wordt, en `default.el` en het terminal type bestand, die beide *na* je `.emacs` bestand geladen worden.

De instellingen en definities in je `.emacs` bestand overschrijven conflicterende instellingen en definities in een `site-start.el` bestand, als die er is, maar de instellingen en definities in een `default.el` en een terminal type bestand overschrijven die in je `.emacs` bestand. (Je kunt interferenties tussen het terminal type bestand voorkomen door het instellen van `term-file-prefix` op `nil`. Zie Sectie 16.11 "Een eennvoudige uitbreiding", pagina 211)

Het `INSTALL` bestand dat bij de distributie meekomt bevat beschrijvingen van de `site-init.el` en `site-load.el` bestanden.

De `loadup.el`, `startup.el` en `loaddefs.el` bestanden beheren het laden. Deze bestanden staan in de `lisp` directories van de Emacs distributie en zijn het waard om te bekijken.

Het `loaddefs.el` bestand bevat veel goede suggesties over wat in je `.emacs` bestand te zetten, of in een systeembreed initialisatiebestand.

16.2 Variabelen specificeren met `defcustom`

Je kunt variabelen specificeren met `defcustom` waardoor jij en anderen de Emacs `customize` eigenschap kunnen gebruiken om hun waarden te zetten. (Je kunt `customize` niet gebruiken om functiedefinities te schrijven. Maar je kunt `defuns` in je `.emacs` bestand schrijven. Je kunt elke Lisp expressie in je `.emacs` bestand zetten.)

De `customize` eigenschap hangt af van de `defcustom` macro. Hoewel je `defvar` en `setq` kunt gebruiken voor variabelen die gebruikers instellen, is de `defcustom` macro voor die taak gemaakt.

Je kunt je kennis van `defvar` gebruiken voor het schrijven van de eerste drie argumenten van `defcustom`. Het eerste argument van `defcustom` is de naam van de variabele. Het tweede argument is de initiële waarde van de variabele, als die er is, en deze waarde wordt alleen ingesteld als de variabele niet al is gezet. Het derde argument is de documentatie.

Het vierde en volgende argument voor `defcustom` specificeren types en opties. Deze zijn niet opgenomen in `defvar`. (Deze argumenten zijn optioneel.)

Elk van deze argumenten bestaan uit een sleutelwoord gevolgd door een waarde. Elk sleutelwoord start met een dubbelepunt karakter ‘`:`’.

De aanpasbare gebruikersoptie variabele `text-mode-hook` ziet er zo uit:

```
(defcustom text-mode-hook nil
  "Normal hook run when entering Text mode and many related modes."
  :type 'hook
  :options '(turn-on-auto-fill flyspell-mode)
  :group 'wp)
```

De naam van de variabele is `text-mode-hook`. Het heeft geen default waarde en de documentatiestring vertelt je wat het doet.

Het sleutelwoord `:type` vertelt Emacs de datasoort waarop de `text-mode-hook` kan worden gezet en hoe de waarde te tonen in het Customization buffer.

Het sleutelwoord `:options` specificeert een gesuggereerde lijst van waarden voor de variabele. Meestal worden `:options` toegepast op een hook. De lijst is alleen een suggestie en is niet exclusief, iemand die de variabele zet kan het op een andere waarde zetten. De lijst die volgend op het sleutelwoord `:options` getoond wordt is bedoeld om een makkelijke keuze voor de gebruiker te bieden.

Het sleutelwoord `:group` tenslotte vertelt het Emacs Customization commando in welke groep de variabele zich bevindt. Het vertelt waar het te vinden is.

De `defcustom` macro herkent meer dan een dozijn sleutelwoorden. Voor meer informatie, zie Sectie “Writing Customization Definitions” in *The GNU Emacs Lisp Reference Manual*.

Beschouw `text-mode-hook` als een voorbeeld.

Er zijn twee manieren om deze variabele aan te passen. Je kunt het customize commando gebruiken of je schrijft de betreffende expressies zelf.

Om het customize commando te gebruiken, typ je:

```
M-x customize
```

en ontdek dat de groep voor het editten van tekstbestanden “Text” heet. Ga die groep in. Text Mode Hook is het eerste onderdeel. Je kunt op de verschillende opties klikken, zoals `turn-on-auto-fill`, om de waarden te zetten. Daarna klik je op de button en kies je

```
Save for Future Sessions
```

Emacs schrijft een expressie in je `.emacs` bestand. Het ziet er zo uit:

```
(custom-set-variables
  ;; custom-set-variables was added by Custom.
  ;; If you edit it by hand, you could mess it up, so be careful.
  ;; Your init file should contain only one such instance.
  ;; If there is more than one, they won't work right.
 '(text-mode-hook '(turn-on-auto-fill text-mode-hook-identify)))
```

(De functie `text-mode-hook-identify` vertelt `toggle-text-mode- auto-fill` welke buffers in Text mode zijn. Het schakelt automatisch aan.)

De functie `custom-set-variables` werkt een beetje anders dan een `setq`. Hoewel ik nooit geleerd heb wat de verschillen zijn, wijzig ik handmatig de `custom-set-variables` expressies: ik maak wijzigingen de mij redelijke lijken en heb nooit

problemen gehad. Anderen geven de voorkeur om het customize commando te gebruiken en laten Emacs het werk voor hen doen.

Een andere `custom-set-...` functie is `custom-set-faces`. Deze functie stelt verschillende lettertypen in. In de loop der tijd heb ik flink wat lettertypen ingesteld. Af en toe zet ik ze opnieuw met `customize` en af en toe wijzig ik eenvoudig zelf de `custom-set-faces` expressie in mijn `.emacs` bestand.

Een tweede manier om de `text-mode-hook` aan te passen is het zelf te doen in je `.emacs` bestand met code die niets van doen heeft met de `custom-set-...` functies.

Wanneer je dit doet en dan later `customize` gebruikt, zie je een boodschap die zegt:

```
CHANGED outside Customize; operating on it here may be unreliable.
```

Deze boodschap is slechts een waarschuwing. Wanneer je op de button klikt om

```
Save for Future Sessions
```

schrijft Emacs een `custom-set-...` expressie aan het einde van je `.emacs` bestand die wordt geëvalueerd na je handschreven expressie. Het zal daardoor jouw handgeschreven expressie overrulen. Dit kan geen kwaad. Onthoudt welke expressie actief is als je dit doet. Wanneer je dat vergeet kan je in de war raken.

Zolang je onthoudt elke waarden gezet zijn, heb je geen problemen. De waarden zijn in elk geval in je initialisatiebestand gezet, dat meestal `.emacs` heet.

Ik zelf gebruik `customize` weinig. Meestal schrijf ik zelf de expressies.

Overigens om meer compleet te zijn inzake de defines: `debsubst` definieert een inline functie. De systax is net als die van `defun`. `defconst` definieert een symbool als een constante. Het doel is dat zowel programma's als gebruikers nooit aan waarde wijzigen die gezet is met `defconst`. (Je kunt het wijzigen, de waarde is gezet als een variabele, maar doe dat liever niet.)

16.3 Starten met een .emacs bestand

Wanneer je Emacs start laadt het je `.emacs` bestand, tenzij je het zegt dat niet te doen door '`-q`' op de commandline te specificeren. (Het commando `emacs -q` geeft je een standaard out-of-the-box Emacs.)

Een `.emacs` bestand bevat Lisp expressies. Vaak zijn dit niet meer dan expressies om waarden te zetten. Soms zijn het functiedefinities.

Zie Sectie "The Init File `~/.emacs`" in *The GNU Emacs Manual*, voor een korte beschrijving van initialisatiebestanden.

Dit hoofdstuk behandelt een gedeelte van dit gebied, het is een wandeling tussen fragmenten van een compleet, lang gebruikt `.emacs` bestand—mijn eigen.

Het eerste deel van het bestand bestaat uit commentaar: reminders voor mijzelf. Nu onthoud ik die dingen wel natuurijk, maar toen ik startte deed ik dat niet.

```
;;;; Bob's .emacs file
; Robert J. Chassell
; 26 September 1985
```

Zie die datum! Ik startte dit bestand een lange tijd geleden. Sindsdien heb ik er steeds aan toegevoegd.

```
; Each section in this file is introduced by a
; line beginning with four semicolons; and each
; entry is introduced by a line beginning with
; three semicolons.
```

Dit beschrijft de gebruikelijke conventie voor commentaar in Emacs Lisp. Alles op een regel dat achter een puntkomma staat is commentaar. Twee, drie en vier puntkomma's worden gebruikt om subsecties en secties te markeren. (Zie Sectie "Comments" in *The GNU Emacs Lisp Reference Manual*, voor meer over commentaar.

```
;;;; The Help Key
; Control-h is the help key;
; after typing control-h, type a letter to
; indicate the subject about which you want help.
; For an explanation of the help facility,
; type control-h two times in a row.
```

Onthoud: typ twee keer *C-h* voor help.

```
; To find out about any mode, type control-h m
; while in that mode.  For example, to find out
; about mail mode, enter mail mode and then type
; control-h m.
```

"Mode help", zoals ik het noem, is erg behulpzaam. Meestal vertelt het alles wat je moet weten.

Je hoeft natuurlijk geen commentaar zoals deze in je .emacs bestand op te nemen. Ik voegde ze in het mijne in omdat ik steeds Mode help vergat, net als de conventies voor commentaar—maar was altijd in staat te onthouden om hier te kijken om mijzelf te herinneren.

16.4 Text and Auto Fill Mode

Nu komen we bij het gedeelte dat Text mode en Auto fill aanzet[2].

```
;;; Text mode and Auto Fill mode
;; The next two lines put Emacs into Text mode
;; and Auto Fill mode, and are for writers who
;; want to start writing prose rather than code.
(setq-default major-mode 'text-mode)
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

Hier is het eerste deel van dit .emacs bestand dat iets doet, anders dan reminders voor een vergeetachtig mens!

[2] Deze sectie stelt instellingen voor die meer geschikt zijn voor schrijvers. Voor programmeurs wordt de default mode automatisch ingesteld op de corresponderende prog-mode gebaseerd op het soort bestand. En is het helemaal goed als je de fundamental mode houdt als de standaard mode.

De eerste twee regels tussen haakjes vertellen Emacs om Text mode aan te zetten wanneer je een bestand vindt, *behalve* wanneer dat bestand in een andere mode zou moeten, zoals C mode.

Wanneer Emacs een bestand leest kijkt het naar de extensie van de bestandsnaam, als die er is. (De extensie is het gedeelte dat komt na een '`.`'.) Wanneer het bestand eindigt met een '`.c`' of '`.h`' extensie dan zet Emacs C mode aan. Ook kijkt Emacs naar de eerste niet-lege regel van het bestand. Als de regel '`-*- C -*-`' zegt, dan zet Emacs C mode aan. Emacs beschikt over een lijst van extensies en specificaties die het automatisch gebruikt. Daarnaast kijkt Emacs in de buurt van de laatste pagina voor een per-buffer lokale variabelen lijst, als die er is.

Zie de secties "How Major Modes are Chosen" en "Local Variables in Files" in *The GNU Emacs Manual*.

Nu terug naar het `.emacs` bestand.

Hier is de regel opnieuw. Hoe werkt die?

```
(setq-default major-mode 'text-mode)
```

Deze regel is een korte, maar complete Emacs Lisp expressie.

Wij zijn al bekend met `setq`. Wij gebruiken een vergelijkbare macro `setq-default` om de volgende variabele, `major-mode`[3], op de daaropvolgende waarde te zetten, die `text-mode` is. Het enkele aanhalingsteken voor `text-mode` vertelt Emacs om direct met het `text-mode` te werken en niet voor wat dan ook het staat. Zie Sectie 1.9 "De waarde van een variabele zetten", pagina 16, voor een herinnering van hoe `setq` werkt. Het belangrijkste punt is dat er geen verschil is tussen een procedure die je gebruikt om een waarde te zetten in je `.emacs` bestand en een procedure die je ergens anders in Emacs gebruikt.

Hier is de volgende regel:

```
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

In deze regel voegt het `add-hook` commando `turn-on-auto-fill` toe aan de variabele.

`turn-on-auto-fill` is de naam van een programma, dat, je raadde het!, Auto Fill mode aanzet.

Elke keer als Emacs Text mode aanzet, dan voert Emacs de commando's uit die aan Text mode gehooked zijn. Dus elk keer dat Emacs Text mode aanzet, zet Emacs ook Auto Fill mode aan.

In het kort, de eerste regel zort dat Emacs in Text mode gaat wanneer je een bestand bewerkt, tenzij de bestandnaam extensie, de eerste niet-lege regel of lokale variabelen Emacs anders zeggen.

Text mode stelt, naast andere acties, de syntax table in om voor schrijvers makkelijk te werken. In Text mode beschouwt Emacs een apostrof als onderdeel van een woord, net als een letter en beschouwt Emacs een punt en een spatie niet

[3] Wij gebruiken hier `setq-default` omdat `text-mode` buffer-local is. Wanneer we `setq` gebruiken dan wordt het alleen op het huidige buffer toegepast, terwijl het gebruik van `setq-default` ook van toepassing is op nieuw te maken buffers. Dit wordt niet voor programmeurs aangeraden.

als onderdeel van een woord. Dus verplaatst *M-f* je over 'it's'. Anderzijds, in C mode stopt *M-f* direct achter de 't' van 'it's'.

De tweede regel zorgt dat Emacs Auto Fill mode aanzet wanneer het Text mode aanzet. In Auto Fill mode breekt Emacs automatisch een regel die te breed is af en brengt het extreem brede deel van de regel terug naar de volgende regel. Emacs breekt regels tussen woorden, niet midden in ze.

Wanneer Auto Fill mode uitstaat, dan lopen regels door naar rechts net zo lang als je ze typt. Afhankelijk van hoe je de waarde van `truncate-lines` instelt verdwijnen de woorden die je typt aan de rechterkant van het scherm of worden in een nogal lelijke en onleesbare manier als een vervolgregel op het scherm getoond.

Daarnaast vertel ik Emacs fill commando in dit deel van het `.emacs` bestand het Emacs fill commando twee spaties in te voegen na een dubbele punt:

```
(setq colon-double-space t)
```

16.5 Mail aliasen

Hier is een `setq` die samen met meer reminders, mail aliases aanzet.

```
;;; Message mode
; To enter message mode, type 'C-x m'
; To enter RMAIL (for reading mail),
; type 'M-x rmail'
(setq mail-aliases t)
```

Deze `setq` zet de waarde van de variabele `mail-aliases` op `t`. Omdat `t` waar betekent, zegt deze regel in feite "Ja, gebruik mail aliases."

Mail aliases zijn makkelijke korte namen voor lange e-mailaddressen of voor een lijst met e-mailadressen. Het bestand waar je de aliases in opslaat is `~/.mailrc`. Je schrijft een alias als volgt:

```
alias geo george@foobar.wiz.edu
```

Wanneer je een bericht aan George schrijft adresseer je het aan 'geo', de mailer breidt 'geo' automatisch uit tot het volledige adres.

16.6 Indent Tabs Mode

Standaard voegt Emacs in plaats van meerdere spaties tabs in wanneer het een region formateert. (Je kunt bijvoorbeeld meerdere regels tekst in een keer laten inspringen met het commando `indent-region`.) Tabs zien er goed uit op een terminal of met gewoon afdrukken, maar zij produceren slecht ingesprongen output wanneer je TEX of Texinfo gebruikt omdat TEX tabs negeert.

Het volgende zet Indent Tabs mode uit:

```
;;; Prevent Extraneous Tabs
(setq-default indent-tabs-mode nil)
```

Merk op dat deze regel `setq-default` gebruikt in plaats van `setq` dat we eerder gezien hebben. Alleen in buffers die geen eigen lokale waarde voor de variabele hebben stelt `setq-default` waarden in.

Zie sectie "Tabs vs. Spaces" en "Local Variables in Files" in *The GNU Emacs Manual*.

16.7 Enkele keybindings

Nu enkele persoonlijke keybindings.

```
;;; Compare windows
(keymap-global-set "C-c w" 'compare-windows)
```

`compare-windows` is een handig commando dat de tekst in je huidige window vergelijkt met de tekst in het volgende window. Het maakt de vergelijking beginnend bij point in elk window, en springt over tekst in elk window voor zover die gelijk is. Ik gebruik dit commando vaak.

Dit toont ook hoe je een key globaal zet, voor alle modes.

Het commando dat toetsen instelt is `keymap-global-set`. Het wordt gevolgd door de keybinding. In een `.emacs` bestand wordt de keybinding geschreven zoals getoond. `C-c` staat voor Control-C, dit betekent de control toets an de `c` tegelijk aanslaan. De `w` betekent de `w` toets aanslaan. De keybinding wordt omgeven door dubbele aanhalingstekens. (Wanneer je een `META` toets in plaats van een `CTRL` zou binden, dan schrijf je `M-c` in je `.emacs` bestand. Zie Sectie “Rebinding Keys in Your Init File” in *The GNU Emacs Manual*, for details.)

Het commando dat door de toetsen wordt aangeroepen is `compare-windows`. Merk op dat `compare-windows` wordt voorafgegaan door een enkele quote. Anders zou Emacs het eerst het symbool proberen te evalueren om zijn waarde te bepalen.

Deze drie dingen, de dubbele aanhalingstekens, de backslash voor de ‘`C`’ en de enkele quote zijn noodzakelijke onderdelen van keybindings die ik vaak vergeet. Gelukkig ben ik me gaan herinneren dat ik moet kijken in mijn bestaande `.emacs` bestand, en gebruiken wat er in staat.

Voor wat betreft de keybinding zelf: `C-c w`. Dit combineert de prefix toets `C-c`, met een enkel karakter, in dit geval `w`. Deze toetsencombinatie, `C-c` gevold door een enkel karakter, is strikt gereserveerd voor je eigen gebruik. (Ik noem deze *eigen* toetsen, omdat deze voor mijn eigen gebruik zijn.) Je moet altijd in staat zijn om zulke keybindings voor je eigen gebruik te maken zonder de keybindings van iemand anders te verprutsen. Wanneer je ooit een uitbreiding van Emacs schrijft, vermijdt dan alsjeblieft deze toetsen voor publiek gebruik te nemen. Maak in plaats daarvan een toetscombinatie zoals `C-c C-w`. Anders zouden we zonder onze eigen toetscombinaties geraken.

Hier is een andere keybinding met commentaar:

```
;;; Key binding for 'occur'
; I use occur a lot, so let's bind it to a key:
(keymap-global-set "C-c o" 'occur)
```

Het commando `occur` toont alle regels in het huidige buffer die een match voor een reguliere expressie bevatten. Wanneer de region actief is, dan beperkt `occur` matches tot die region. Zo niet dan gebruikt die het gehele buffer. De matchende regels worden getoond in een buffer met de naam `*Occur*`. Dat buffer dient als een menu om naar de matches te springen.

Hier volgt hoe je een toets ontkoppelt, zodat die niet werkt:

```
;;; Unbind 'C-x f'
(keymap-global-unset "C-x f")
```

Er is een reden voor deze ontkoppeling: Ik ontdekte dat ik per ongeluk *C-x f* typ terwijl ik *C-x C-f* bedoelde te typen. In plaats van een bestand te vinden, zoals de bedoeling was, zette ik per ongeluk de breedte voor uitgevulde tekst, vrijwel altijd op een breedte die ik niet wil. Omdat ik haast nooit mijn standaard breedte aanpas, heb ik simpel deze toetscombinatie ontkoppeld.

Het volgende koppelt een bestaande toetscombinatie opnieuw:

```
;;; Rebind 'C-x C-b' for 'buffer-menu'
(keymap-global-set "C-x C-b" 'buffer-menu)
```

Standaard roept *C-x C-b* het commando `list-buffers` aan. Dit commando vermeldt al je buffers in *een ander* window. Maar omdat ik altijd iets wil doen in dat window, geef ik de voorkeur aan het commando `buffer-menu`, dat niet alleen de buffers vermeldt, maar ook point naar dat window verplaatst.

16.7.1 Verouderde globale keybinding commando's

Vroeger werden toetscombinaties globaal gebonden met een functie op laag niveau, `global-set-key`, wat nu als achterhaald wordt beschouwd. Hoewel je wordt aangemoedigd om `keymap-global-set` te gebruiken, kom je waarschijnlijk `global-set-key` op verschillende plekken tegen. Het eerste voorbeeld in deze sectie kan met `global-set-key` worden herschreven als:

```
(global-set-key "\C-cw" 'compare-windows)
```

Het lijkt erg op `keymap-global-set`, met de keybinding volgens een wat anders formaat. Control-C wordt getoond als `\C-c` in plaats van `C-c`. Er staat geen spatie tussen de toestaanslagen, zoals `\C-c` en `w` in dit voorbeeld. Ondanks het verschil wordt dit in documentatie in verband met de leesbaarheid nog steeds geschreven als *C-cw*.

Vroeger werden toestcombinaties globaal ontkoppeld met een functie op laag niveau, `global-unset-key`, wat nu als verouderd beschouwd wordt. Het keybinding formaat volgt dat van `global-set-key`. Het toetscombinatie ontkoppelende voorbeeld in deze sectie kan herschreven worden als:

```
;;; Unbind 'C-x f'
(global-unset-key "\C-xf")
```

16.8 Keymaps

Emacs gebruikt *keymaps* om te registreren welke toetscombinatie welk commando aanroept. Wanneer je *keymaps* gebruikt om de keybinding voor een enkel commando in te stellen in alle delen van Emacs, dan specificeer je de keybinding in de `current-global-map`.

Specifieke modes, zoals C mode of Text mode, hebben hun eigen keymaps. De mode-specifieke keymaps overrulen de globale map die met alle buffers gedeeld is.

De functie `keymap-global-set` bindt of bindt opnieuw de globale keymap. Het volgende bijvoorbeeld bindt de toetscombinatie *C-x C-b* aan de functie `buffer-menu`:

```
(keymap-global-set "C-x C-b" 'buffer-menu)
```

Mode-specifieke keymaps worden gebonden met de functie `keymap-set`, die een specifieke keymap als argument heeft, net als de toestcombinatie en het command. De volgende expressie bijvoorbeeld bindt het commando `texinfo-insert-@group` aan *C-c C-c g*:

```
(keymap-set texinfo-mode-map "C-c C-c g" 'texinfo-insert-@group)
```

Vroeger werden keymaps gebonden met een functie op lager niveau, `define-key` dat nu als verouderd wordt beschouwd. Hoewel je wordt aangemoedigd om `keymap-set` te gebruiken, komt je waarschijnlijk `define-key` op verschillende plaatsen tegen. De keybinding hierboven kan herschreven worden met `define-key` als:

```
(define-key texinfo-mode-map "\C-c\C-cg" 'texinfo-insert-@group)
```

De functie `texinfo-insert-@group` is een kleine uitbreiding op Texinfo mode dat '`@group`' in een Texinfo bestand invoegt. Ik gebruik dit commando vaak en geef de voorkeur aan het typen van drie toestaanslagen *C-c C-c g* in plaast van zes aanslagen *@ g r o u p*. ('`@group`' en het bijbehorende '`@end group`' zijn commando's die alle omsloten tekst bij elkaar op een bladzijdeh houdt. Veel meerregelige voorbeelden in dit boek zijn omgeven door '`@group ... @end group`'.)

Hier is de `texinfo-insert-@group` functiedefinitie:

```
(defun texinfo-insert-@group ()
  "Insert the string @group in a Texinfo buffer."
  (interactive)
  (beginning-of-line)
  (insert "@group\n"))
```

(Ik had natuurlijk Abbrev mode kunnen gebruiken om toetsaanslagen te besparen, in plaats van een functie om een woord in te voegen, maar ik geef voorkeur aan toestaanslagen die consistent zijn met andere Texinfo mode keybindings.)

Je komt veel `keymap-set` en `define-key` expressies tegen in `loaddefs.el` en ook in verschillende mode libraries, zoals `cc-mode.el` en `lisp-mode.el`.

Zie Sectie "Customizing Key Bindings" in *The GNU Emacs Manual*, en Sectie "Keymaps" in *The GNU Emacs Lisp Reference Manual*, voor meer informatie over keymaps.

16.9 Bestanden laden

Veel mensen in de GNU Emacs community hebben uitbreidingen op Emacs geschreven. In de loop der tijd worden die exentensies vaak opgenomen in nieuwe releases. De Calendar en Diary packages bijvoorbeeld zijn nu onderdeel van de de standaard GNU Emacs, net als Calc.

Je kunt met het commando `load` een compleet bestand evalueren en daarmee alle functies en variabelen in het bestand in Emacs installeren. Bijvoorbeeld:

```
(load "~/emacs/slowsplit")
```

Dit evalueert, dat wil zeggen laadt het bestand `slowsplit.el` of, als dat bestaat, het byte-compiled bestand `slowsplit.elc` in de `emacs` subdirectory of in je home directory. Het bestand bevat de functie `split-window-quietly` dat John Robinson in 1989 schreef.

De functie split-window-quietly split een window met een minimum aan scherm-aanpassingen. Ik installeerde het in 1989 omdat het goed werkte op de langzame 1200 baud terminals die ik toen gebruikte. Tegenwoordig kom ik af en toe zo'n langzame connectie tegen, maar ik blijf de functie gebruiken omdat ik de manier waarop het de onderste helft van een buffer onder in het nieuwe window houdt en de bovenste helft in het bovenste window.

Om de keybinding voor het standaard split-window-vertically te vervangen, moet je de toetsaanslagen als volgt binden aan split-window-quietly:

```
(keymap-global-set "C-x 2" 'split-window-quietly)
```

Als je veel uitbreidingen laadt, zoals ik doe, dan kan je in plaats van de exacte lokatie van het uitbreidingsbestand, zoals hierboven, de directory specificeren als deel van de het load-path van Emacs. Wanneer Emacs een bestand laadt kijkt het dan zowel in die directory als in de standaard lijst van directories. (De standaard lijst wordt gespecificeerd in epaths.h wanneer Emacs wordt gebouwd.)

Het volgende commando voegt je ~/emacs directory toe aan het bestaande load-path:

```
(setq load-path (cons "~/emacs" load-path))
```

OVerigens, load-library is een interactieve interface voor de load functie. De complete functie ziet er zo uit:

```
(defun load-library (library)
  "Load the Emacs Lisp library named LIBRARY.
This is an interface to the function `load'.  LIBRARY is searched
for in `load-path', both with and without `load-suffixes' (as
well as `load-file-rep-suffixes').

See Info node `(emacs)Lisp Libraries' for more details.
See `load-file' for a different interface to `load'."
  (interactive
   (list (completing-read "Load library: "
                          (apply-partially 'locate-file-completion-table
                                           load-path
                                           (get-load-suffixes)))))
  (load library))
```

De naam van de functie load-library komt van het gebruik van "library" als conventioneel synoniem voor "file". De bron voor het commando load-library staat in de files.el library.

Een ander interactief commando dat een iets andere taak doet is load-file. Zie Sectie "Libraries of Lisp Code for Emacs" in *The GNU Emacs Manual*, voor informatie over het verschil tussen load-library en dit commando.

16.10 Autoloading

In plaats van het installeren van een functie door het bestand dat het bevat te laden, of door het evalueren van de functiedefinitie, kan je een functie beschikbaar maken zonder die te installeren, totdat die voor het eerst wordt aangeroepen. Dit heet *autoloading*.

Wanneer je een autoloaded functie uitvoert, evalueert Emacs automatisch het bestand dat de definitie bevat en roept daarna de functie aan.

Emacs start sneller met autoloaded functies omdat hun libraries niet meteen geladen worden, maar je moet iets langer wachten wanneer je zo'n functie voor het eerst gebruikt, terwijl het bestand waar het in staat is geëvalueerd.

Weinig gebruikte functies worden vaak autoloaded. De library `loaddefs.el` bevat duizenden autoloaded functies, van `5x5` tot `zone`. Natuurlijk kan het zijn dat je vaak een weinig gebruikte functie gebruikt. Wanneer je dat doet, zou je het bestand met die functie moeten laden met een `load` expressie in je `.emacs` bestand.

In mijn `.emacs` bestand laad ik 14 libraries die functies bevatten die anders worden autoloaded. (Eigenlijk zou het beter zijn deze bestanden op te nemen in mijn gedumpte Emacs, maar dat vergat ik. Zie Sectie "Building Emacs" in *The GNU Emacs Lisp Reference Manual*, en het `INSTALL` bestand voor meer over dumping.)

Je kunt ook autoloaded expressies in je `.emacs` bestand opnemen. `autoload` is een ingebouwde functie die tot vijf argumenten neemt, waarvan de laatste drie optioneel zijn. Het eerste argument is de naam van de functie om te autoloaden, de tweede is de naam van het bestand om de laden. Het derde is de documentatie van de functie en de vierde geeft aan of de functie interactief kan worden aangeroepen. Het vijfde argument vertelt welke objecttype—`autoload` kan met een keymap, macro of functie omgaan (de standaard is een functie).

Hier is een typisch voorbeeld

```
(autoload 'html-helper-mode
  "html-helper-mode" "Edit HTML documents" t)
```

(`html-helper-mode` is een ouder alternatief voor `html-mode` die een standaard onderdeel is van de distributie.
Deze expressie autoload de functie `html-helper-mode`. Het haalt die op uit het bestand `html-helper-mode.el` (of uit de byte compiled versie `html-helper-mode.elc` als die bestaat.) Het bestand moet in een directory staan die gespecificeerd is door `load-path`. De documentatie geeft aan dit een mode is die je helpt bij het aanpassen van documenten die geschreven zijn in de HyperText Markup Language. Je roept deze mode interactief aan door *`M-x html-helper-mode`* te typen. (Je moet de gewone documentatie van de functie in de autoload expressie kopiëren omdat de gewone functie nog niet is geladen waardoor de documentatie nog niet beschikbaar is.)

Zie Sectie "Autoload" in *The GNU Emacs Lisp Reference Manual*, voor meer informatie.

16.11 Een eenvoudige uitbreiding: `line-to-top-of-window`

Hier is een eenvoudige uitbreiding op Emacs die de regel waar point staat naar de bovenkant van het window verplaatst. Ik gebruikt dit heel vaak om de tekst makkelijker leesbaar te maken.

Je kunt de volgende code in een apart bestand zetten en dan laden vanuit je `.emacs` bestand, of je kunt het opnemen in je `.emacs` bestand.

Hier is de definitie:

```
;;; Line to top of window;
;;; replace three keystroke sequence  C-u 0 C-l
(defun line-to-top-of-window ()
  "Move the line that point is on to top of window."
  (interactive)
  (recenter 0))
```

Nu de keybinding.

Functietoetsen, muisbutton-events en non-ASCII karakters schrijf je tussen vierkante haakjes, zonder aanhalingstekens.

Zo bind ik `line-to-top-of-window` aan mijn `F6` functietoets:

```
(keymap-global-set "<f6>" 'line-to-top-of-window)
```

Voor meer informatie, zie Sectie “Rebinding Keys in Your Init File” in *The GNU Emacs Manual*.

Wanneer je twee versies van GNU Emacs draait, zoals versie 27 en 28, en één `.emacs` bestand gebruikt, dan selecteer je met de volgende conditional welke code te evalueren:

```
(cond
 ((= 27 emacs-major-version)
  ;; evaluate version 27 code
  ( ... ))
 ((= 28 emacs-major-version)
  ;; evaluate version 28 code
  ( ... )))
```

Recente versies laten bijvoorbeeld de cursor standaard knipperen. Ik haat dat knipperen net zoals sommige andere features, daarom plaats ik het volgende in mijn `.emacs` bestand[4]:

```
(when (>= emacs-major-version 21)
  (blink-cursor-mode 0)
  ;; Insert newline when you press 'C-n' (next-line)
  ;; at the end of the buffer
  (setq next-line-add-newlines t)
  ;; Turn on image viewing
  (auto-image-file-mode t)
  ;; Turn on menu bar (this bar has text)
  ;; (Use numeric argument to turn on)
  (menu-bar-mode 1)
  ;; Turn off tool bar (this bar has icons)
  ;; (Use numeric argument to turn on)
  (tool-bar-mode nil)
```

[4] Wanneer ik instanties van Emacs start die mijn `.emacs` of elk ander site-bestand niet laden zet ik ook het knipperen uit:

```
emacs -q --no-site-file -eval '(blink-cursor-mode nil)'
```

Of tegenwoordig met gebruik van een meer gesofisticeerde verzamelin

```
;; Turn off tooltip mode for tool bar
;; (This mode causes icon explanations to pop up)
;; (Use numeric argument to turn on)
(tooltip-mode nil)
;; If tooltips turned on, make tips appear promptly
(setq tooltip-delay 0.1)  ; default is 0.7 second
 )
```

16.12 X11 kleuren

Je kunt kleuren opgeven wanneer je Emacs met het MIT X Windowing systeem gebruikt.

Ik vind de standaard kleuren niet prettig een geef mijn eigen op.

Hier staan de expressies in mijn .emacs bestand die de waarden zetten:

```
;; Set cursor color
(set-cursor-color "white")

;; Set mouse color
(set-mouse-color "white")

;; Set foreground and background
(set-foreground-color "white")
(set-background-color "darkblue")

;;; Set highlighting colors for isearch and drag
(set-face-foreground 'highlight "white")
(set-face-background 'highlight "blue")

(set-face-foreground 'region "cyan")
(set-face-background 'region "blue")

(set-face-foreground 'secondary-selection "skyblue")
(set-face-background 'secondary-selection "darkblue")

;; Set calendar highlighting colors
(with-eval-after-load 'calendar
  (set-face-foreground 'diary   "skyblue")
  (set-face-background 'holiday "slate blue")
  (set-face-foreground 'holiday "white"))
```

De verschillende blauwtinten kalmeren mijn oog en voorkomen dat ik het scherm zie knipperen.

Anderzijds had ik mijn specificaties in verschillende X initialisatiebestanden kunnen zetten. Ik kon bijvoorbeeld de kleuren voor de voorgrond, achtergrond, cursor en pijltje (oftewel muis) als volgt in mijn ~/.Xresources bestand zetten:

```
Emacs*foreground:   white
Emacs*background:   darkblue
Emacs*cursorColor:  white
Emacs*pointerColor: white
```

In elk geval, omdat dit geen onderdeel van Emacs is, zet de root kleur van mijn X window in mijn ~/.xinitrc bestand zoals dit[5]:

```
xsetroot -solid Navy -fg white &
```

[5] Ik gebruik ook modernere window managers zoals Enlightenment, Gnome, of KDE. In die gevallen specificeer ik vaak een afbeelding in plaats van een enkele kleur.

16.13 Diverse instellingen voor een .emacs bestand

Hier zijn een paar diverse instellingen

– Stel de vorm en kleur van de muiscursor:

```
; Cursor shapes are defined in
; '/usr/include/X11/cursorfont.h';
; for example, the 'target' cursor is number 128;
; the 'top_left_arrow' cursor is number 132.

(let ((mpointer (x-get-resource "*mpointer"
                                "*emacs*mpointer")))
  ;; If you have not set your mouse pointer
  ;;     then set it, otherwise leave as is:
  (if (eq mpointer nil)
      (setq mpointer "132")) ; top_left_arrow
  (setq x-pointer-shape (string-to-number mpointer))
  (set-mouse-color "white"))
```

– Of je zet de waarden voor verschillende eigenschappen in een alist, zoals hier:

```
(setq-default
 default-frame-alist
 '((cursor-color . "white")
   (mouse-color . "white")
   (foreground-color . "white")
   (background-color . "DodgerBlue4")
   ;; (cursor-type . bar)
   (cursor-type . box)
   (tool-bar-lines . 0)
   (menu-bar-lines . 1)
   (width . 80)
   (height . 58)
   (font .
         "-Misc-Fixed-Medium-R-Normal--20-200-75-75-C-100-ISO8859-1")
   ))
```

– Converteer CTRL-*h* naar DEL en DEL naar CTRL- *h*.
(Sommige oude toetsenborden vereisen dit, hoewel ik het probleem de laatste tijd niet gezien heb.)

```
;; Translate 'C-h' to <DEL>.
; (key-translate "C-h" "C-?")

;; Translate <DEL> to 'C-h'.
(key-translate "C-?" "C-h")
```

– Zet knipperende cursor uit!

```
(if (fboundp 'blink-cursor-mode)
    (blink-cursor-mode -1))
```

of start GNU Emacs met het commando `emacs -nbc`.

– Bij het gebruik van grep
  '-i' Negeer verschil hoofd- en kleine letters
  '-n' Vermeldt het regelnummer op elke output-regel
  '-H' Toon de bestandsnaam voor elke match
  '-e' Bescherm patronen die beginnen met een minteken, '-'

```
(setq grep-command "grep -i -nH -e ")
```

– Vind een bestaand buffer, zelfs als het een andere naam heeft
  Dit voorkomt problemen met symbolische links.

```
(setq find-file-existing-other-name t)
```

– Stel je taalomgeving en de standaard input methode in

```
(set-language-environment "latin-1")
;; Remember you can enable or disable multilingual text input
;; with the toggle-input-method' (C-\) command
(setq default-input-method "latin-1-prefix")
```

  Wanneer je in Chinese GB karakters wilt schrijven, gebruik dan in plaats daarvan dit:

```
(set-language-environment "Chinese-GB")
(setq default-input-method "chinese-tonepy")
```

Onplezierige keybindings verbeteren

Sommige systemen binden keys onplezierig, Soms verschijnt bijvoorbeeld de CTRL op een onhandige plek in plaats van uiterst links in de home-row.

Wanneer mensen deze soort keybindings verbeteren veranderen zij meestal niet hun .emacs bestand. In plaats daarvan binnen zij de juiste toetsen op hun consoles met de loadkeys of install-keymap commando's in hun boot-script en nemen daarna xmodmap commando's in hun .xinitrc of .Xsession bestand voor X Windows.

Voor een boot-script:

```
loadkeys /usr/share/keymaps/i386/qwerty/emacs2.kmap.gz
```

or

```
install-keymap emacs2
```

Voor een .xinitrc of .Xsession bestand wanneer de Caps Lock toets is uiterst links in de home-row:

```
# Bind the key labeled 'Caps Lock' to 'Control'
# (Such a broken user interface suggests that keyboard manufacturers
# think that computers are typewriters from 1885.)

xmodmap -e "clear Lock"
xmodmap -e "add Control = Caps_Lock"
```

In een `.xinitrc` of `.Xsession` bestand om de `ALT` toets te converteren naar een `META` toets:

```
# Some ill designed keyboards have a key labeled ALT and no Meta
xmodmap -e "keysym Alt_L = Meta_L Alt_L"
```

16.14 Een aangepaste mode line

Tenslotte een eigenschap die ik echt waardeer: een aangepaste mode line.

Wanneer ik over een netwerk werk dan vergeet ik welke machine ik gebruik. Ook heb ik de neiging te vergeten waar ik ben, en op welke regel point is.

Daarom stel ik mijn mode line in zodat die er zo uitziet:

```
-:-- foo.texi   rattlesnake:/home/bob/  Line 1  (Texinfo Fill) Top
```

Ik bezoek een bestand met de naam `foo.texi`, op mijn machine `rattlesnake` in mijn `/home/bob` buffer. Ik ben op regel 1, in Texinfo mode en ben aan de bovenkant van het buffer.

Mijn `.emacs` bestand heeft een sectie die er zo uitziet:

```
;; Set a Mode Line that tells me which machine, which directory,
;; and which line I am on, plus the other customary information.
(setq-default mode-line-format
 (quote
  (#("-" 0 1
     (help-echo
      "mouse-1: select window, mouse-2: delete others ..."))
   mode-line-mule-info
   mode-line-modified
   mode-line-frame-identification
   "    "
   mode-line-buffer-identification
   "    "
   (:eval (substring
           (system-name) 0 (string-match "\\..+" (system-name))))
   ":"
   default-directory
   #(" " 0 1
     (help-echo
      "mouse-1: select window, mouse-2: delete others ..."))
   (line-number-mode " Line %l ")
   global-mode-string
   #("   %[(" 0 6
     (help-echo
      "mouse-1: select window, mouse-2: delete others ..."))
   (:eval (format-time-string "%F"))
   mode-line-process
   minor-mode-alist
   #("%n" 0 2 (help-echo "mouse-2: widen" local-map (keymap ...)))
   ")%] "
   (-3 . "%P")
   ;;   "-%-"
   )))
```

Hier herdefinieer ik de standaard mode line. Veel delen zijn van het origineel, maar ik heb een paar wijzigingen. Ik zet het *default* mode line formaat zo dat verschillende modes, zoals Info, het mogen overschrijven.

Veel elementen in de lijst spreken voor zich: `mode-line-modified` is een variabele die aangeeft of het buffer is gewijzigd, `mode-name` geeft de naam van de mode aan, enzovoorts. Het format ziet er echter gecompliceerd uit omdat we twee eigenschappen nog niet hebben besproken:

De eerste string in de mode line is een minteken, '`-`'. In vroegere tijden zou het eenvoudig zijn gespecificeerd met `"-"`. Maar tegenwoordig kan Emacs eigenschappen aan een string toevoegen, zoals highlighting of, zoals in dit geval, een helpfunctie. Wanneer je de muiscursor boven het minteken houdt verschijnt helpinformatie. (Standaard moet je een zeven-tiende van een seconde wachten voor de informatie verschijnt. Je kunt de timing daarvan aanpassen door de waarde van `tooltip-delay` te wijzigen.)

Het nieuwe string-formaat heeft een speciale syntax:

```
#("-" 0 1 (help-echo "mouse-1: select window, ..."))
```

De `#(` begint een lijst. Het eerste element in de lijst is de string zelf, slechts één '`-`'. Het tweede en derde element specificeren de range waarop het vierde element van toepassing is. Een range start *na* een karakter, dus een nul betekent dat de range direct voor het eerste karakter start, een 1 betekent dat de range direct achter het eerste karakter eindigt. Het derde element is een eigenschap voor de range. Het bestaat uit een eigenschappen-lijst, een eigenschap-naam, in dit geval '`help-echo`', gevolgd door een waarde, in dit geval een string. Het tweede, derde en vierde element van dit nieuwe string-formaat kan worden herhaald.

Zie Sectie "Text Properties" in *The GNU Emacs Lisp Reference Manual*, en zi Sectie "Mode Line Format" in *The GNU Emacs Lisp Reference Manual*, voor meer informatie.

`mode-line-buffer-identification` toont het naam van het huidige buffer. Het is een lijst die begint met `(#("%12b" 0 4 ....` De `#(` begint de lijst.

De '`"%12b"`' toont de naam van het huidige buffer, met de functie `buffer-name` waarmee we vertrouwd zijn. De '`12`' specificeert het maximum aantal karakters dat wordt getoond. Wanneer een naam minder karakters heeft, dan wordt witte ruimte toegevoegd om tot dit aantal uit te vullen. (Buffernamen kunnen en moeten langer zijn dan 12 karakters. Deze lengte werkt goed in een typisch 80 karakters breed window.)

`:eval` geeft aan om de volgende vorm te evalueren en het resultaat te gebruiken als een te tonen string. In dit geval toont de expressie het eerste element van de volledige systeemnaam. Het einde van het eerste component is een '`.`' (punt) daarom gebruik ik de `string-match` functie om mij de lengte van het eerste component te vertellen. De substring van het nul-de karakter tot die lengte is de naam van de machine.

Dit is de expressie:

```
(:eval (substring
        (system-name) 0 (string-match "\\..+" (system-name))))
```

‘`%[`’ en ‘`%]`’ laten een paar vierkante haken voor elk recursief editing level verschijnen. ‘`%n`’ vermeldt “Narrow” wanneer versmalling van toepassing is. ‘`%P`’ vertelt je het percentage van het buffer dat boven de onderkant van het window is, of “Top”, “Bottom”, of “All”. (Een kleine letter ‘`p`’ vertelt je het percentage boven de *bovenkant* van het window.) ‘`%-`’ voegt voldoende mintekens in om de regel uit te vullen.

Onthoud, je hoeft niet van Emacs te houden om er van te houden—je eigen Emacs kan andere kleuren, andere commando’s en andere toetscombinaties hebben dan de standaard Emacs.

Anderzijds, als je een standaard out-of-the-box Emacs zonder enige aanpassing wilt starten, typ:

```
emacs -q
```

Dit start een Emacs die *niet* je `.emacs` initialisatiebestand laadt. Een gewone, standaard Emacs. Niets meer.

17 Debuggen.

GNU Emacs heeft twee debuggers, `debug` en `edebug`. De eerste is ingebouwd in Emacs en is altijd bij je in de buurt. De tweede vereist dat je een functie instrumenteert voordat je het kunt gebruiken.

Beide debuggers zijn uitgebreid beschreven in Sectie "Debugging Lisp Programs" in *The GNU Emacs Lisp Reference Manual*. In dit hoofdstuk loop ik een kort voorbeeld van beide door.

17.1 `debug`

Stel dat je een functiedefinitie hebt geschreven die bedoeld is om de som van de getallen 1 tot een opgegeven getal terug te geven. (Dit is de eerder besproken `driehoek` functie. Zie "Voorbeeld met een aflopende teller", pagina 124, voor een bespreking.)

Je functiedefinitie heeft echter een bug. Je hebt '`1-`' verkeerd getypt als '`1=`'. Hier is de kapotte definitie:

```
(defun driehoek-met-bug (getal)
  "Geef som van getallen 1 t/m GETAL terug."
  (let ((totaal 0))
    (while (> getal 0)
      (setq totaal (+ totaal getal))
      (setq getal (1= getal)))      ; Fout hier.
    totaal))
```

Wanneer je dit leest in Info kan je deze definitie op de gebruikelijke manier evalueren. Je ziet `driehoek-met-bug` in het echogebied verschijnen.

Evalueer nu de functie `driehoek-met-bug` met een argument van 4:

```
(driehoek-met-bug 4)
```

Dit creëert en zet je in een `*Backtrace*` buffer met de tekst:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-function 1=)
  (1= getal)
  (setq getal (1= getal))
  (while (> getal 0) (setq totaal (+ totaal getal))
        (setq getal (1= getal)))
  (let ((totaal 0)) (while (> getal 0) (setq totaal ...)
     (setq getal ...)) totaal)
  driehoek-met-bug(4)
  eval((driehoek-met-bug 4) nil)
  elisp--eval-last-sexp(nil)
  #f(compiled-function () #<bytecode ...>()
  handler-bind-1(#f(compiled-function () #<bytecode ...>)
    (error) eval-expression--debug)
  eval-last-sexp(nil)
  funcall-interactively(eval-last-sexp nil)
  call-interactively(eval-last-sexp nil nil)
  command-execute(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(Ik heb dit voorbeeld iets anders geformatteerd. De debugger breekt de lange regels niet af. Zoals gebruikelijk kun je de debugger sluiten door `q` in het `*Backtrace*` buffer te typen.)

Voor een bug zo simpel als deze vertelt in de praktijk de Lisp error regel je wat je moet weten om de definitie te verbeteren. De functie `1=` is void.

Stel echter dat je niet helemaal zeker bent wat er aan de hand is. Dan kan je de complete backtrace lezen.

Emacs start automatisch de debugger die je in het `*Backtrace*` buffer zet. Je kunt de debugger ook handmatig starten zoals hieronder beschreven.

Lees het `*Backtrace*` buffer van onder naar boven. Het vertelt je wat Emacs deed wat tot de fout leidde. Emacs riep interactief `C-x C-e` (`eval-last-sexp`), wat leidde tot de evaluatie van de `driehoek-met-bug`. Elke regel erboven zegt wat de Lisp interpreter daarna evalueerde.

De derde regel van de bovenkant van het buffer is

```
(setq getal (1= getal))
```

Emacs probeerde deze expressie te evalueren. Om dat te doen, probeerde het de binnenste expressie te evalueren, die op de tweede regel van boven staat:

```
(1= getal)
```

Dit is waar de fout optrad. Zoals de bovenste regel zegt:

```
Debugger entered--Lisp error: (void-function 1=)
```

Je kunt de vergissing verbeteren, de functiedefinitie opnieuw evalueren en je test opnieuw uitvoeren.

17.2 debug-on-entry

Emacs start de debugger automatisch wanneer je functie een fout bevat.

Overigens kan je de debugger handmatig starten in alle versies van Emacs. Het voordeel is dat de debugger zelfs draait wanneer je geen bug in je code hebt. Soms is je code zonder bugs!

Je kunt de debugger handmatig starten door de functie aan te roepen met `debug-on-entry`.

Typ:

```
M-x debug-on-entry RET driehoek-met-bug RET
```

Evalueer nu het volgende:

```
(driehoek-met-bug 5)
```

Alle versies van Emacs maken een `*Backtrace*` buffer en vertellen je dat het begint met de evaluatie van de functie `driehoek-met-bug`:

```
Debugger entered--entering a function:
* driehoek-met-bug(5)
  eval((driehoek-met-bug 5) nil)
  #f(compiled-function () #<bytecode ...>)()
  handler-bind-1(#f(compiled-function () #<bytecode ...>) (error) eval-expression--debu
  eval-last-sexp(nil)
  funcall-interactively(eval-last-sexp nil)
  call-interactively(eval-last-sexp nil nil)
  command-execute(eval-last-sexp)
```

Typ `d` in het `*Backtrace*` buffer. Emacs evalueert de eerste expressie in de `driehoek-met-bug`. Het buffer ziet er zo uit:

```
Debugger entered--entering a function:
* apply(#f(lambda (getal) :dynbind "Geef som van getallen 1 t/m GETAL terug."
  (let ((totaal 0)) (while (> getal 0) (setq totaal ...)
        (setq getal ...)) totaal)) (5))
* driehoek-met-bug(5)
  eval((driehoek-met-bug 5) nil)
elisp--eval-last-sexp(nil)
  #f(compiled-function () #<bytecode ...>)()
  handler-bind-1(#f(compiled-function () #<bytecode ...>) (error) eval-expression--debu
  eval-last-sexp(nil)
  funcall-interactively(eval-last-sexp nil)
  call-interactively(eval-last-sexp nil nil)
  command-execute(eval-last-sexp)
```

Typ nu opnieuw, langzaam acht keer `d`. Elke keer dat je `d` typt, evalueert Emacs een volgende expressie in de functiedefinitie.

Uiteindelijk ziet het buffer zer zo uit:

```
Debugger entered--beginning evaluation of function call form:
* (setq getal (1= getal))
* (while (> getal 0) (setq totaal (+ totaal getal))
        (setq getal (1= getal)))

* (let ((totaal 0)) (while (> getal 0) (setq totaal ...)
         (setq getal ...)) totaal)
* #f(lambda (getal) :dynbind "Geef som van getallen 1 t/m GETAL terug."
  (let ((totaal 0)) (while (> getal 0)
     (setq totaal (+ totaal getal)) (setq getal ...)) totaal))(5)
* apply(#f(lambda (getal) :dynbind "Geef som van getallen 1 t/m GETAL terug."
  (let ((totaal 0)) (while (> getal 0) (setq totaal ...)
     (setq getal ...)) totaal)) 5)
* driehoek-met-bug(5)
  eval((driehoek-met-bug 5) nil)
  elisp--eval-last-sexp(nil)
  #f(compiled-function () #<bytecode ...>)()
  handler-bind-1(#f(compiled-function () #<bytecode ...>) (error) eval-expression--debu
  eval-last-sexp(nil)
  funcall-interactively(eval-last-sexp nil)
  call-interactively(eval-last-sexp nil nil)
  command-execute(eval-last-sexp)

  #f(compiled-function () #<bytecode ...>)()
  handler-bind-1(#f(compiled-function () #<bytecode ...>) (error) eval-expression--debu
  eval-last-sexp(nil)
  funcall-interactively(eval-last-sexp nil)
  call-interactively(eval-last-sexp nil nil)
  command-execute(eval-last-sexp)
```

Nadat je *d* nog twee keer getypt hebt, bereikt Emacs tenslotte de fout, en zien de bovenste twee regels van het *Backtrace* buffer er zo uit:

```
Debugger entered--entering a function:
* eval-expression--debug((void-function 1=))
* (1= getal)
...
```

Je kon met het typen van *d* door de functie stappen.

Je kunt *Backtrace* buffer afsluiten door *q* er in te typen. Dit stopt de trace, maar zet debug-on-entry niet uit.

Om het effect van debug-on-entry te stoppen, roep je cancel-debug-on-entry aan en de naam van de functie, zoals dit:

```
M-x cancel-debug-on-entry RET driehoek-met-bug RET
```

(Wanneer je dit in Info leest, stop je debug-on-entry nu.)

17.3 debug-on-quit en (debug)

Naast het zetten van debug-on-error of het aanroepen van debug-on-entry, zijn er twee andere manieren om debug te starten.

Je kunt steeds wanneer *C-g* (keyboard-quit) typt debug starten door de variabele debug-on-quit op t te zetten. Dit is zinvol om oneindige loops te debuggen.

Of je neemt een regel met de tekst (debug) op in je code waar je de debugger wilt starten, zoals dit:

```
(defun driehoek-met-bug (getal)
  "Geef som van getallen 1 t/m GETAL terug."
  (let ((totaal 0))
    (while (> getal 0)
      (setq totaal (+ totaal getal))
      (debug)                          ; Start debugger.
      (setq getal (1= getal)))         ; Fout hier.
    totaal))
```

De functie debug is gedetailleerd beschreven in Sectie “The Lisp Debugger” in *The GNU Emacs Lisp Reference Manual*.

17.4 De edebug Source Level Debugger

Edebug is een source level debugger. Edebug toont gewoonlijk de bron van de code die je aan het debuggen bent met links een pijltje dat de regel aanwijst die je op dit moment uitvoert.

Je kunt regel voor regel door de uitvoering van de functie wandelen of snel er doorheen gaan tot het bereiken van een *breakpoint*, waar de uitvoering stopt.

Edebug is beschreven in Sectie “Edebug” in *The GNU Emacs Lisp Reference Manual*.

Hier is een functiedefinitie voor `driehoek-recursieve` met een bug. Zie Sectie 11.3.4 “Recursie in plaats van een teller”, pagina 132, voor een review er van.

```
(defun driehoek-recursief-bugged (aantal)
  "Geef de som van de getallen 1 tot en met AANTAL terug.
Met recursie."
  (if (= aantal 1)
      1
    (+ aantal
       (driehoek-recursief-bugged
        (1= aantal)))))                ; Fout hier
```

Normaliter installeer je deze definitie door de cursor achter het haakje dat de functie afsluit te plaatsen en *C-x C-e* (`eval-last-sexp`) te typen, of anders de cursor in de definitie te plaatsten en *C-M-x* (`eval-defun`) te typen. (Standaard werkt het `eval-defun` commando alleen in Emacs Lisp mode of in Lisp Interaction mode.)

Echter, om deze functiedefinitie voor Edebug voor te bereiden, moet je eerst de code *instrumenteren* met een ander commando. Je doet dit door de cursors in of direct achter de definitie te plaatsen en dan het volgende te typen:

```
M-x edebug-defun RET
```

Dit zorgt dat Emacs automatisch Edebug laadt, wanneer het nog niet is geladen en de functie correct instrumenteert.

Nadat het instrumenteren van de functie plaats je de cursor achter de volgende expressie en typ je *C-x C-e* (`eval-last-sexp`):

```
(driehoek-recursief-bugged 3)
```

Je springt terug naar de bron van `driehoek-recursief-bugged` met de cursor gepositioneerd aan begin van de `if` regel van de functie. Ook zie je een pijlpunt aan de linkerkant van de regel. De pijlpunt markeert de regel die de functie aan het uitvoeren is. (In de volgende voorbeelden tonen we de pijlpunt met ‘`=>`’. In een windowing systeem kan de pijlpunt er als een gevulde driehoek in de window-kantlijn uitzien.)

```
=>*(if (= number 1)
```

In het voorbeeld is de lokatie van point getoond met een sterretje, ‘*’ (in Info wordt het getoond als ‘`-!-`’.)

Wanneer je nu `SPC` aanslaat, verplaatst point naar de volgende expressie om uit te voeren, de regel ziet er zo uit:

```
=>(if *(= number 1)
```

Terwijl je doorgaat met `SPC` aan te slaan verplaatst point van expressie naar expressie. Steeds als een functie een waarde teruggeeft, wordt die tegelijkertijd in het echogebied getoond. Nadat je point bijvoorbeeld voorbij `number` verplaatst hebt, zie je het volgende:

```
Result: 3 (#o3, #x3, ?\C-c)
```

Dit betekent dat de waarde van `getal` 3 is, wat gelijk is aan octal drie, hexidicimaal drie en ASCII Control-C (de derde letter van het alfabet, voor het geval je deze informatie nodig hebt).

Je kunt door de code blijven gaan totdat je de regel met de fout bereikt. Voor de evaluatie ziet die regel er zo uit:

```
=>       *(1= aantal)))))              ; Fout hier
```

Wanneer je nog een keer `SPC` aanslaat, produceer je een foutmelding met de tekst:

```
Symbol's function definition is void: 1=
```

Dit is de bug

Druk op `q` om Edebug te stoppen.

Om de instrumentalisatie van een functiedefinitie te verwijderen, her-evalueer je die eenvoudig met een commando dat het niet instrumenteert. Bijvoorbeeld kan je de cursor na het haakje dat de functie afsluit plaatsen en `C-x C-e` te typen.

Edebug doet veel meer dan je door een functie te wandelen. Je kunt het zo instellen dat het zelf door een functie rent en alleen stopt bij een fout of bij gespecificeerde stoppunten. Je kunt het de veranderende waardes van verschillende expressies laten tonen, of ontdekken hoe vaak een functie wordt aangeroepen, en meer.

Edebug is beschreven in Sectie “Edebug” in *The GNU Emacs Lisp Reference Manual*.

17.5 Debuggen oefeningen

- Installeer de `tel-woorden-voorbeeld` functie en laat het de ingebouwde debugger ingaaan wanneer je het aanroept. Run het commando op een region die twee woorden bevat. Je moet `d` een opmerkelijk aantal keer aanslaan. Wordt op jouw systeem een hook aangeroepen wanneer het commando eindigt? (Voor informatie over hooks, zie Sectie “Command Loop Overview” in *The GNU Emacs Lisp Reference Manual*.))
- Kopieer `tel-woorden-voorbeeld` in het `*scratch*` buffer, instrumenteer de functie voor Edebug, en wandel door de uitvoering. De functie hoeft geen bug te hebben, maar je kunt er een introduceren als je dat wilt. Wanneer de functie geen bug heeft, eindigt de wandeling zonder problemen.
- Typ `?` terwijl je Edebug draait om een lijst te zien van alle Edebug commando’s. (De `global-edebug-prefix` is meestal `C-x X`, dus `CTRL-x` gevold door een hoofdletter `X`. Gebruik deze prefix voor commando’s buiten het Edebug debugging buffer.)
- Gebruik het `p` (`edebug-bounce-point`) commando in het Edebug debugging buffer om te zien in welke region de `tel-woorden-voorbeeld` aan het werk is.
- Verplaats point naar een plaats verder in de functie en typ dan het `h` (`edebug-goto-here`) commando om naar die lokatie te springen.
- Gebruik het `t` (`edebug-trace-mode`) commando om te zorgen dat Edebug zelf door de functie wandelt. Gebruik een hoofdletter `T` voor `edebug-Trace-fast-mode`.
- Zet een breakpoint en draai Edubug in Trace mode tot het dit stoppunt bereikt.

18 Conclusie

Je hebt nu het einde van deze Introductie bereikt. Je hebt nu genoeg geleerd over het programmeren in Emacs Lisp om waardes te zetten, eenvoudige `.emacs` bestanden voor jezelf en je vrienden te schrijven en om eenvoudige aanpassingen en extensies op Emacs te schrijven.

Dit is een plaats op te stoppen. Of, als je dat wilt, kan je nu verder gaan en je zelf iets leren.

Je hebt wat van de basisfuncties van programmeren geleerd. Maar slechts een paar. Er zijn nog veel meer elementen die makkelijk te gebruiken zijn die we niet geraakt hebben.

Het pad dat je nu kunt volgen ligt ondermeer in de broncode van GNU Emacs en in *The GNU Emacs Lisp Reference Manual*.

De Emacs Lisp broncode is een avontuur. Wanneer je de broncode leest en een onbekende functie of expressie tegenkomt moet je ontdekken wat die doet.

Ga naar de Reference Manual. Het is een grondige, complete en redelijk makkelijk te lezen beschrijving van Emacs Lisp. Het is niet alleen voor experts geschreven, maar voor mensen die weten wat jij weet. (De *Reference Manual* zit bij de standaard GNU Emacs distributie. Net als deze introductie komt het als Texinfo bronbestand, zodat je het op je computer kunt lezen en als een opgemaakt gedrukt boek.)

Ga naar de andere ingebouwde hulp die onderdeel van GNU Emacs is: de ingebouwde documentatie voor alle functies en variabelen, en `xref-find-definitions`, het programma dat je naar de broncode brengt.

Hier is een voorbeeld van hoe ik de broncode verken. Vanwege de naam is `simple.el` het bestand dat ik het eerste bekeek, een lange tijd geleden. Het blijkt dat sommige functies in `simple.el` erg gecompliceerd zijn of op zijn minst op het eerste gezicht er gecompliceerd uitzien. De functie `open-line` bijvoorbeeld ziet er gecompliceerd uit.

Je wilt misschien langzaam door deze functie wandelen, zoals we met de functie `forward-sentence` deden. ((Zie Sectie 12.3 “forward-sentence”, pagina 145.) Of je wilt deze functie overslaan en naar een andere kijken, zoals `split-line`. Je hoeft niet alle functies te lezen. Volgens `tel-woorden-in-defun` bevat de functie `split-line` 102 woorden en symbolen.

Hoewel die kort is, bevat `split-line` expressie die we nog niet bestudeerd hebben: `skip-chars-forward`, `indent-to`, `current-column` en `insert-and-inherit`.

Beschouw de functie `skip-chars-forward`. In GNU Emacs ontdek je meer over `skip-chars-forward` door het typen van *`C-h f`* (`describe-function`) en de naam van de functie. Dit levert je de functiedocumentatie.

Je kunt misschien raden wat gedaan wordt door een functie met een goede naam zoals `indent-to`, of je kunt het opzoeken. Overigens staat de functie `describe-function` zelf in `help.el`. Het is een van die lange, maar ontcijferbare functies. Je kunt de `describe-function` opzoeken met het *`C-h f`* commando!

In dit geval, omdat de code in Lisp is, bevat het `*Help*` buffer de naam van de library met de broncode van de functie. Je kunt point op de naam van de library

zetten en dan op de `RET` toets drukken, die in deze situatie gebonden is aan `help-follow` en je direct naar de broncode brengt, op dezelfde manier als *`M-.`* (`xref-find-definitions`).

De definitie van `describe-function` illustreert hoe de `interactive` aan te passen zonder de standaard karakter-codes te gebruiken en het laat zien hoe je een tijdelijke buffer maakt.

(De functie `indent-to` is in C geschreven in plaats van Emacs Lisp, het is een ingebouwde functie. `help-follow` neemt je naar de broncode net als `xref-find-definitions`, wanneer correct opgezet.)

Je kunt naar een de bron van een functie kijken met `xref-find-definitions` dat aan *`M-.`* gebonden is. Tenslottte kun je ontdekken wat de Reference Manual te zeggen heeft door de manual in Info te bezoeken en *`i`* (`Info-index`) en de naam van de functie te typen, of het op te zoeken in de index van een afgedrukt exemplaar van de manual.

Op dezelfde manier kan je uit vinden vat bedoeld wordt met `insert-and-inherit`.

Andere interessante bronbestanden zijn `paragraphs.el`, `loaddefs.el`, en `loadup.el`. Het bestand `paragraphs.el` bevat zowel korte makkelijk te begrijpen functies als ook langere. Het bestand `loaddefs.el` bevat veel standaard autoloads en veel keymaps. Ik heb het nooit in zijn geheel bekeken, alleen in delen. Het bestand `loadup.el` is het bestand dat veel standaard onderdelen van Emacs laadt, het vertelt je veel over hoe Emacs is gebouwd. (Zie Sectie “Building Emacs” in *The GNU Emacs Lisp Reference Manual*, voor meer over bouwen.)

Zoals ik heb gezegd, je hebt sommmige basisdelen geleerd, echter, en erg belangrijk, hebben we amper belangrijke aspecten van programmeren geraakt. Ik heb niets gezegd over het sorteren van informatie, bevalve het gebruik van de standaard `sort` functie. Ik heb niets gezegd over hoe je programma’s schrijft die programma’s schrijven. Dit zijn onderwerpen voor een ander en ander soort boek, een andere manier van leren.

Je hebt genoeg geleerd om veel practisch werk met GNU Emacs te doen. En wat je gedaan hebt, is beginnen. Dit is het einde van het begin.

Appendix A De de-de functie

Wanneer je je tekst schrijft dupliceer je soms woorden—zoals “je je” aan het begin van deze zin. Ik ontdekte dat ik het meest van “de” dupliceer, daarom noem ik de functie om gedupliceerde woorden te detecteren `de-de`

Als eerste stap zou de volgende reguliere expressie kunnen gebruiken om naar duplicaten te zoeken:

```
\\(\\w+[ \t\n]+\\)\\1
```

Deze regexp matcht een of meer woorden-vormende karakters gevolg door een of meer spaties, tabs of nieuwe-regel-tekens. Het detecteert echter geen op verschillende regels gedupliceerde woorden, omdat het einde van het eerste woord, het einde van de regel, verschilt van het einde van het tweede woord, een spatie. (Voor meer informatie over reguliere expressies zie Hoofdstuk 12 “Reguliere expressie zoekopdrachten”, pagina 143, en ook Sectie “Syntax of Regular Expressions” in *The GNU Emacs Manual*, and Sectie “Regular Expressions” in *The GNU Emacs Lisp Reference Manual*.)

Je zou ook slechts kunnen zoeken naar gedupliceerde woord-vormende karakters maar dat werkt niet omdat het patroon doublures detecteert zoals het twee keer voorkomen van “de” in “belde de”.

Een andere mogelijke regexp zoekt naar woord-vormende karakters gevolgd door niet-woord-vormende karakters, herhaald. Hier matcht ‘`\\w+`’ een of meer woord-vormende karakters en ‘`\\W*`’matcht nul of meer niet-woord-vormende karakters.

```
\\(\\(\\w+\\)\\W*\\)\\1
```

Opnieuw, niet nuttig

Hier is het patroon dat ik gebruik. Het is niet perfect, maar goed genoeg. ‘`\\b`’ matcht de lege string, onder voorwaarde dat het aan het begin of eind van een woord staat. ‘`[^@ \n\t]+`’ matcht een of meer karakters die *niet* een @-teken, spatie, nieuwe-regel-teken of tab zijn.

```
\\b\\([^@ \n\t]+\\)[ \n\t]+\\1\\b
```

Je kunt meer ingewikkelde expressies schrijven, maar ik vind deze expressie goeg genoeg en gebruik die daarom.

Hier is de functie `de-de` die ik in mijn `.emacs` bestand opneem, samen met een handige globale keybinding.

```
(defun de-de ()
  "Zoek voorwaarts naar naar een gedupliceerd woord.."
  (interactive)
  (message "Zoeken naar naar gedupliceerde woorden ...")
  (push-mark)
  ;; Deze regexp is niet perfect
  ;; maar is behoorlijk goed:
  (if (re-search-forward
       "\\b\\([^@ \n\t]+\\)[ \n\t]+\\1\\b" nil 'move)
      (message "Gedupliceerd woord gevonden")
    (message "End of buffer")))

;; Bind 'de-de' to  C-c \
(keymap-global-set "C-c \\" 'de-de)
```

Hier is de test tekst:

```
een twee twee drie vier vijf
vijf zes zeven
```

Je kunt de andere hierboven getoonde reguliere expressies in de functiedefinitie substitueren en elke op deze lijst proberen.

Appendix B De killring hanteren

De killring is een lijst die is getransformeerd in een ring door de functie `current-kill`. De `yank` en `yank-pop` commando's gebruiken de functie `current-kill`.

Deze appendix beschrijft de functie `current-kill` en ook de commando's `yank` en `yank-pop`, maar eerst beschouwen we de werking van de killring.

De killring heeft een standaard maximum lengte van zestig items. Dit getal is te groot voor een uitleg. Zet in het in plaats daarvan op vier. Evalueer het volgende:

```
(setq old-kill-ring-max kill-ring-max)
(setq kill-ring-max 4)
```

Kopieer daarna elke regel van het volgende voorbeeld in de killring. Je kunt elke regel killen met *C-k* of markeren en dan kopiëren met *M-w*.

(In a read-only buffer, such as the `*info*` buffer, the kill command, *C-k* (`kill-line`), will not remove the text, merely copy it to the kill ring. However, your machine may beep at you. Alternatively, for silence, you may copy the region of each line with the *M-w* (`kill-ring-save`) command. You must mark each line for this command to succeed, but it does not matter at which end you put point or mark.)

Roep deze calls op volgorde aan, zodat vijf elementen de killring proberen te vullen.

```
eerst wat tekst
tweede stukje tekst
derde regel
vierde regel tekst
vijfde stukje tekst
```

Vind vervolgens de waarde van `kill-ring` door het evalueren van

```
kill-ring
```

Dit is:

```
("vijfde stukje tekst" "vierde regel tekst"
"derde regel" "tweede stukje tekst")
```

Het eerste element, '`eerst wat tekst`', was verwijderd.

Om de oude waarde van de lengte van de killring te herstellen, evalueer:

```
(setq kill-ring-max old-kill-ring-max)
```

B.1 De `current-kill` functie

De functie `current-kill` wijzigt het element in de killring naar waar `kill-ring-yank-pointer` wijst. (De functie `kill-new` zet ook `kill-ring-yank-pointer` zo dat die wijst naar het laatste element van de killring. De functie `kill-new` wordt direct of indirect gebruikt door `kill-append`, `copy-region-as-kill`, `kill-ring-save`, `kill-line` en `kill-region`.)

De functie current-kill wordt gebruikt door yank en door yank-pop. Hier is de code van current-kill:

```
(defun current-kill (n &optional do-not-move)
  "Rotate the yanking point by N places, and then return that kill.
If N is zero and `interprogram-paste-function' is set to a
function that returns a string or a list of strings, and if that
function doesn't return nil, then that string (or list) is added
to the front of the kill ring and the string (or first string in
the list) is returned as the latest kill.
If N is not zero, and if `yank-pop-change-selection' is
non-nil, use `interprogram-cut-function' to transfer the
kill at the new yank point into the window system selection.
If optional arg DO-NOT-MOVE is non-nil, then don't actually
move the yanking point; just return the Nth kill forward."
  (let ((interprogram-paste (and (= n 0)
                                 interprogram-paste-function
                                 (funcall interprogram-paste-function))))
    (if interprogram-paste
        (progn
          ;; Disable the interprogram cut function when we add the new
          ;; text to the kill ring, so Emacs doesn't try to own the
          ;; selection, with identical text.
          (let ((interprogram-cut-function nil))
            (if (listp interprogram-paste)
              (mapc 'kill-new (nreverse interprogram-paste))
              (kill-new interprogram-paste)))
          (car kill-ring))
      (or kill-ring (error "Kill ring is empty"))
      (let ((ARGth-kill-element
             (nthcdr (mod (- n (length kill-ring-yank-pointer))
                          (length kill-ring))
                     kill-ring)))
        (unless do-not-move
          (setq kill-ring-yank-pointer ARGth-kill-element)
          (when (and yank-pop-change-selection
                     (> n 0)
                     interprogram-cut-function)
            (funcall interprogram-cut-function (car ARGth-kill-element))))
        (car ARGth-kill-element)))))
```

Bedenk dat ook de kill-new functie kill-ring-yank-pointer naar het laatste element van de killring laat wijzen, wat betekent dat al de functies die het aanroepen de waarde indirect zetten: kill-append, copy-region-as-kill, kill-ring-save, kill-line en kill-region.

Hier is de regel in `kill-new`, die uitgelegd is in "The `kill-new` function", pagina 98.

```
(setq kill-ring-yank-pointer kill-ring)
```

De functie `current-kill` ziet er ingewikkeld uit, maar zoals gewoonlijk, wordt die begrijpelijk door het stuk voor stuk uit te rafelen. Kijk eerst naar de vorm van het geraamte:

```
(defun current-kill (n &optional do-not-move)
  "Rotate the yanking point by N places, and then return that kill."
  (let varlist
    body...)
```

De functie heeft twee argumenten waarvan er een optioneel is. Het heeft een documentatiestring. Het is *niet* interactief.

De body van de functiedefinitie is een `let` expressie, die zelf een body en een *varlist* heeft.

De `let` expressie declareert een variabele die alleen bruikbaar is binnen de grenzen van deze functie. Deze variabele heet `interprogram-paste` en is voor het kopiëren naar een ander programma. Het is niet voor kopiëren binnen deze instantie van GNU Emacs. De meeste window systemen verschaffen een faciliteit voor het plakken tussen programma's. Helaas werkt zo'n faciliteit meestal alleen met het laatste element. De meeste window systemen hebben een ring met veel mogelijkheden niet geadopteerd, hoewel Emacs dat al tientallen jaren heeft.

De `if` expressie heeft twee delen, eentje voor als er een `interprogram-paste` bestaat, en eentje voor als dat niet zo is.

Laat ons het anders-gedeelte van de `current-kill` functie beschouwen. (Het dan-gedeelte gebruikt de `kill-new` functie, die we al eerder hebben beschreven. Zie "De `kill-new` functie", pagina 98.)

```
(or kill-ring (error "Kill ring is empty"))
(let ((ARGth-kill-element
       (nthcdr (mod (- n (length kill-ring-yank-pointer))
                    (length kill-ring))
               kill-ring)))
  (or do-not-move
      (setq kill-ring-yank-pointer ARGth-kill-element))
  (car ARGth-kill-element))
```

De code controleert eerst of de killring inhoud heeft, zo niet, dan geeft het een foutmelding.

Merk op dat de `or` expressie erg lijkt op het testen van de lengte met een `if`.

```
(if (zerop (length kill-ring))          ; if-deel
    (error "Kill ring is empty"))       ; dan-deel
  ;; Geen anders-deel
```

Wanneer er niets in de killring is, moet zijn lengte nul zijn en krijgt de gebruiker een foutboodschap: '`Kill ring is empty`'. De functie `current-kill` gebruikt een `or` expressie die eenvoudiger is. Maar een `if` expressie vertelt ons aan wat er gebeurt.

Deze `if` expressie gebruikt de functie `zerop` die waar teruggeeft wanneer de waarde die die test nul is. Wanneer `zerop` waar test, wordt het dan-deel van de `if` geëvalueerd. Het dan-deel is een lijst die begint met de functie `error`, die een functie sis vergelijkbaar met de functie `message` (zie Sectie 1.8.5 "De `message`

functie", pagina 14), het toont een een-regelige boodschap in het echogebied. In aanvulling op het tonen van een boodschap stopt `error` echter ook de evaluatie van de functie waarin die is opgenomen. Dit betekent dat de rest van de functie niet wordt geëvalueerd wanneer de lengte van de killring nul is.

Dan selecteert de functie `current-kill` het element om terug te geven. De selectie hangt af van het aantal plaatsen dat `current-kill` roteert en van waar `kill-ring-yank-pointer` naar wijst.

Vervolgens is hetzij het optionele argument `do-not-move` waar, dan wel de huidige waarde van `kill-ring-yank-pointer` wordt zo gezet dat die naar de lijst wijst. Tenslotte geeft een andere expressie het eerste element van de lijst terug, zelfs als het argument `do-not-move` waar is.

Naar mijn mening is het lichtelijk misleidend, tenminste voor mensen, om de term "error" te gebruiken voor de naam van de functie `error`. Een betere term zou "cancel" zijn. Strikt genomen kan je natuurlijk niet wijzen, of een pointer roteren, naar een lijst die geen lengte heeft, zodat vanuit het perspectief van de computer is het woord "error" correct. Maar een mens verwacht zo iets te proberen, als is het maar om vast te stellen od de killring vol of leeg is. Dit is een daad van verkenning.

Vanuit het menselijk perspectief is de daad van verkenning en ontdekking niet noodzakelijkerwijs een fout, en daarom zou het niet als zodanig moeten worden bestempeld, zelfs in de buik van de computer. Zoals het nu is, impliceert de code in Emacs dat een mens die deugdzaam handelt door zijn of haar omgeving te verkennen een fout maakt. Dat is niet goed. Alhoewel de computer dezelfde stappen zet die het doet wanneer er een fout is, zou een term zoals "cancel" een helderder connotatie hebben.

Naast andere acties zet het anders-deel van de `if` expressie de waarde van `kill-ring-yank-pointer` naar `ARGth-kill-element`, wanneer de killring iets bevat en de waarde van `do-not-move` `nil` is.

De code ziet er zo uit:

```
(nthcdr (mod (- n (length kill-ring-yank-pointer))
             (length kill-ring))
        kill-ring)))
```

Dit vereist wat onderzoek. Tenzij het niet de bedoeling is de pointer te verplaatsen, wijzigt de functie `current-kill` naar waar `kill-ring-yank-pointer` wijst. Dat is wat de expressie `(setq kill-ring-yank-pointer ARGth-kill-element))` doet. Ook wordt `ARGth-kill-element` hetzelfde ingesteld als een CDR in de killring, met de functie `nthcdr` die in een eerdere sectie besproken is. (Zie Sectie 8.3 "copy-region-as-kill", pagina 94.) Hoe doet het dat?

Zoals we eerder zagen (zie Sectie 7.3 "nthcdr", pagina 81), werkt de `nthcdr` functie door herhaaldelijk de CDR van een lijst te nemen—het neemt de CDR van de CDR van de CDR ...

De volgende twee expressies produceren hetzelfde resultaat:

```
(setq kill-ring-yank-pointer (cdr kill-ring))

(setq kill-ring-yank-pointer (nthcdr 1 kill-ring))
```

De `nthcdr` expressie is echter meer gecompliceerd. Het gebruikt de functie `mod` om vast te stellen welke CDR te selecteren.

(Je herinnert je om eerst naar de binnenste functie te kijken, we moeten inderdaad de `mod` in.)

De functie `mod` geeft de waarde van het eerste argument modulo het tweede. Met andere woorden, het geeft de rest na het delen van het eerste argument door het tweede. De teruggegeven waarde heeft hetzelfde teken als het tweede argument.

Dus

```
(mod 12 4)
  ⇒ 0  ;; omdat er geen rest is
(mod 13 4)
  ⇒ 1
```

In dit geval is het eerste argument vaak kleiner dan het tweede. Dat is prima.

```
(mod 0 4)
  ⇒ 0
(mod 1 4)
  ⇒ 1
```

We kunnen raden wat de functie `-` doet. Het is net als `+` maar trekt af in plaats van optellen. De functie `-` trekt het tweede argument af van het eerste. We weten al wat de functie `length` doet (zie Sectie 7.2.1 “length”, pagina 81). Het geeft de lengte van een lijst terug.

En `n` is de naam van het vereiste argument van de functie `current-kill`.

Dus wanneer het eerste argument van `nthcdr` nul is, geeft de `nthcdr` expressie de hele lijst terug, zoals je kunt zien door het evalueren van het volgende:

```
;; kill-ring-yank-pointer en kill-ring hebben een lengte van 4
;; en (mod (- 0 4) 4) ⇒ 0
(nthcdr (mod (- 0 4) 4)
        '("vierde regel tekst"
          "derde regel"
          "tweede stukje tekst"
          "eerst wat tekst"))
```

Wanneer het eerste argument van de functie `current-kill` één is, geeft de `nthcdr` expressie de lijst terug zonder het eerste element.

```
(nthcdr (mod (- 1 4) 4)
        '("vierde regel tekst"
          "derde regel"
          "tweede stukje tekst"
          "eerst wat tekst"))
```

Overigens zijn zowel `kill-ring` als `kill-ring-yank-pointer` *globale variabelen*. Dat betekent dat elke expressie in Emacs Lisp toegang tot ze heeft. Zij zijn niet zoals de lokale variabelen gezet door `let` of zoals de symbolen in een argument list. Lokale variabelen zijn alleen toegankelijk binnen de `let` die ze definieert of de functie die ze specificeert in een argument list (en binnen de door ze aangeroepen expressies).

(Zie Sectie 3.5 “`let` voorkomt verwarring”, pagina 32, en Sectie 3.1 “De `defun` macro”, pagina 26.)

B.2 `yank`

Na het bestuderen van `current-kill` is de code van de functie `yank` bijna makkelijk.

De functie `yank` gebruikt de variabele `kill-ring-yank-pointer` niet direct. Het roept `insert-for-yank` aan, die roept `current-kill` aan welke de variabele `kill-ring-yank-pointer` zet.

De code ziet er zo uit:

```
(defun yank (&optional arg)
  "Reinsert (\"paste\") the last stretch of killed text.
More precisely, reinsert the stretch of killed text most recently
killed OR yanked.  Put point at end, and set mark at beginning.
With just \\[universal-argument] as argument, same but put point at beginning (and mark
With argument N, reinsert the Nth most recently killed stretch of killed
text.

When this command inserts killed text into the buffer, it honors
`yank-excluded-properties' and `yank-handler' as described in the
doc string for `insert-for-yank-1', which see.

See also the command `yank-pop' (\\[yank-pop])."
  (interactive "*P")
  (setq yank-window-start (window-start))
  ;; If we don't get all the way thru, make last-command indicate that
  ;; for the following command.
  (setq this-command t)
  (push-mark (point))
  (insert-for-yank (current-kill (cond
                                  ((listp arg) 0)
                                  ((eq arg '-) -2)
                                  (t (1- arg)))))
  (if (consp arg)
      ;; This is like exchange-point-and-mark, but doesn't activate the mark.
      ;; It is cleaner to avoid activation, even though the command
      ;; loop would deactivate the mark because we inserted text.
      (goto-char (prog1 (mark t)
                   (set-marker (mark-marker) (point) (current-buffer)))))
  ;; If we do get all the way thru, make this-command indicate that.
  (if (eq this-command t)
      (setq this-command 'yank))
  nil)
```

De sleutel-expressie is `insert-for-yank`, die de string teruggeven door `current-kill` invoegt, maar verwijdert enkele teksteigenschappen er van.

Voor dat het bij die expressie komt, zet de functie de waarde van `yank-window-start` op de positie teruggegeven door de `window-start` expressie., de positie waarop het display thans begint. De functie `yank` zet ook `this-command` en pusht de mark.

Nadat het het passende element yankt, wanneer het optionele argument een CONS is in plaats van een getal of niets, dan plaatst het point aan het begin van de geyankte tekst en markeert het einde.

(De functie `prog1` is net als `progn` maar geeft de waarde van het eerste argument terug in plaats van de waarde van het laatste argument. Het eerste argument is geforceerd om de mark van de buffer als integer terug te geven. Je kunt de documentatie van deze functies zien door in dit buffer point op ze te plaatsten en dan *`C-h f`* (`describe-function`) te typen, gevold door een *`RET`*; de default is de functie.)

Het laatste deel van de functie zegt wat te doen wanneer het succesvol is.

B.3 yank-pop

Na het begrijpen van yank en current-kill, weet je hoe de functie yank-pop te benaderen. Met het weglaten van de documentatie om plek te besparen, ziet die er zo uit:

```
(defun yank-pop (&optional arg)
  "..."
  (interactive "*p")
  (if (not (eq last-command 'yank))
      (error "Previous command was not a yank"))
  (setq this-command 'yank)
  (unless arg (setq arg 1))
  (let ((inhibit-read-only t)
        (before (< (point) (mark t))))
    (if before
        (funcall (or yank-undo-function 'delete-region) (point) (mark t))
      (funcall (or yank-undo-function 'delete-region) (mark t) (point)))
    (setq yank-undo-function nil)
    (set-marker (mark-marker) (point) (current-buffer))
    (insert-for-yank (current-kill arg))
    ;; Set the window start back where it was in the yank command,
    ;; if possible.
    (set-window-start (selected-window) yank-window-start t)
    (if before
        ;; This is like exchange-point-and-mark,
        ;;     but doesn't activate the mark.
        ;; It is cleaner to avoid activation, even though the command
        ;; loop would deactivate the mark because we inserted text.
        (goto-char (prog1 (mark t)
                     (set-marker (mark-marker)
                                 (point)
                                 (current-buffer))))))
  nil)
```

De functie is interactief met een kleine letter 'p' zodat het prefix argument wordt verwerkt en doorgegeven aan de functie. Het commando kan alleen worden gebruikt na een voorgaande yank, anders wordt een foutboodschap gegeven. Deze check gebruikt de variabele last-command die gezet is door yank en elders is besproken. (Zie Sectie 8.3 "copy-region-as-kill", pagina 94.)

De let clausule zet de variabele before op waar of onwaar afhankelijk of point voor of na de mark is en delete dan de region tussen point en mark. Dit is de region dat zojuist was ingevoegd door de voorgaande yank en dit is de tekst die wordt vervangen.

funcall roept het eerste argument als functie aan, en geeft de resterende argumenten er aan door. Het eerste argument is wat de or teruggeeft. De twee resterende argumenten zijn de posities van point en mark, gezet door het voorafgaande yank commando.

Er is meer, maar dit is het moeilijkste deel.

B.4 Het bestand `ring.el`

Interessant is dat GNU Emacs een bestand bevat met de naam `ring.el` dat veel van de functies bevat die we zojuist bespaken. Maar functies zoals `kill-ring-yank-pointer` gebruiken deze library niet, mogelijk omdat ze eerder zijn geschreven.

Appendix C Een grafiek met labels op de assen

Printed axes help you understand a graph. They convey scale. In an earlier chapter (zie ⟨undefined⟩ "Readying a Graph", pagina ⟨undefined⟩), we wrote the code to print the body of a graph. Here we write the code for printing and labeling vertical and horizontal axes, along with the body itself.

Since insertions fill a buffer to the right and below point, the new graph printing function should first print the Y or vertical axis, then the body of the graph, and finally the X or horizontal axis. This sequence lays out for us the contents of the function:

1. Set up code.
2. Print Y axis.
3. Print body of graph.
4. Print X axis.

Here is an example of how a finished graph should look:

```
10 -
                *
                *  *
                *  **
                *  ***
 5 -      *   *******
        * *** *******
        ************
      ***************
 1 - ****************
     |   |    |    |
     1   5   10   15
```

In this graph, both the vertical and the horizontal axes are labeled with numbers. However, in some graphs, the horizontal axis is time and would be better labeled with months, like this:

```
5 -      *
        * ** *
        *******
      ********** **
1 - **************
    |    ^      |
    Jan  June   Jan
```

Indeed, with a little thought, we can easily come up with a variety of vertical and horizontal labeling schemes. Our task could become complicated. But complications breed confusion. Rather than permit this, it is better choose a simple labeling scheme for our first effort, and to modify or replace it later.

These considerations suggest the following outline for the `print-graph` function:

```
(defun print-graph (numbers-list)
  "documentation..."
  (let ((height  ...
        ...))
    (print-Y-axis height ... )
    (graph-body-print numbers-list)
    (print-X-axis ... )))
```

We can work on each part of the `print-graph` function definition in turn.

C.1 The `print-graph` Varlist

In writing the `print-graph` function, the first task is to write the varlist in the `let` expression. (We will leave aside for the moment any thoughts about making the function interactive or about the contents of its documentation string.)

The varlist should set several values. Clearly, the top of the label for the vertical axis must be at least the height of the graph, which means that we must obtain this information here. Note that the `print-graph-body` function also requires this information. There is no reason to calculate the height of the graph in two different places, so we should change `print-graph-body` from the way we defined it earlier to take advantage of the calculation.

Similarly, both the function for printing the X axis labels and the `print-graph-body` function need to learn the value of the width of each symbol. We can perform the calculation here and change the definition for `print-graph-body` from the way we defined it in the previous chapter.

The length of the label for the horizontal axis must be at least as long as the graph. However, this information is used only in the function that prints the horizontal axis, so it does not need to be calculated here.

These thoughts lead us directly to the following form for the varlist in the `let` for `print-graph`:

```
(let ((height (apply 'max numbers-list)) ; First version.
      (symbol-width (length graph-blank)))
```

As we shall see, this expression is not quite right.

C.2 De `print-Y-axis` functie

The job of the `print-Y-axis` function is to print a label for the vertical axis that looks like this:

```
10 -


 5 -


 1 -
```

The function should be passed the height of the graph, and then should construct and insert the appropriate numbers and marks.

It is easy enough to see in the figure what the Y axis label should look like; but to say in words, and then to write a function definition to do the job is another matter. It is not quite true to say that we want a number and a tic every five lines: there are only three lines between the '`1`' and the '`5`' (lines 2, 3, and 4), but four lines between the '`5`' and the '`10`' (lines 6, 7, 8, and 9). It is better to say that we want a number and a tic mark on the base line (number 1) and then that we want a number and a tic on the fifth line from the bottom and on every line that is a multiple of five.

The next issue is what height the label should be? Suppose the maximum height of tallest column of the graph is seven. Should the highest label on the Y axis be '`5 -`', and should the graph stick up above the label? Or should the highest label be '`7 -`', and mark the peak of the graph? Or should the highest label be `10 -`, which is a multiple of five, and be higher than the topmost value of the graph?

The latter form is preferred. Most graphs are drawn within rectangles whose sides are an integral number of steps long—5, 10, 15, and so on for a step distance of five. But as soon as we decide to use a step height for the vertical axis, we discover that the simple expression in the varlist for computing the height is wrong. The expression is `(apply 'max numbers-list)`. This returns the precise height, not the maximum height plus whatever is necessary to round up to the nearest multiple of five. A more complex expression is required.

As usual in cases like this, a complex problem becomes simpler if it is divided into several smaller problems.

First, consider the case when the highest value of the graph is an integral multiple of five—when it is 5, 10, 15, or some higher multiple of five. We can use this value as the Y axis height.

A fairly simply way to determine whether a number is a multiple of five is to divide it by five and see if the division results in a remainder. If there is no remainder, the number is a multiple of five. Thus, seven divided by five has a remainder of two, and seven is not an integral multiple of five. Put in slightly different language, more reminiscent of the classroom, five goes into seven once, with a remainder of two. However, five goes into ten twice, with no remainder: ten is an integral multiple of five.

C.2.1 Side Trip: Compute a Remainder

In Lisp, the function for computing a remainder is `%`. The function returns the remainder of its first argument divided by its second argument. As it happens, `%` is a function in Emacs Lisp that you cannot discover using `apropos`: you find nothing if you type *`M-x apropos`* `RET` *`remainder`* `RET`. The only way to learn of the existence of `%` is to read about it in a book such as this or in the Emacs Lisp sources.

You can try the `%` function by evaluating the following two expressions:

```
(% 7 5)

(% 10 5)
```

The first expression returns 2 and the second expression returns 0.

To test whether the returned value is zero or some other number, we can use the `zerop` function. This function returns `t` if its argument, which must be a number, is zero.

```
(zerop (% 7 5))
     ⇒ nil

(zerop (% 10 5))
     ⇒ t
```

Thus, the following expression will return `t` if the height of the graph is evenly divisible by five:

```
(zerop (% height 5))
```

(The value of `height`, of course, can be found from `(apply 'max numbers-list)`.)

On the other hand, if the value of `height` is not a multiple of five, we want to reset the value to the next higher multiple of five. This is straightforward arithmetic using functions with which we are already familiar. First, we divide the value of `height` by five to determine how many times five goes into the number. Thus, five goes into twelve twice. If we add one to this quotient and multiply by five, we will obtain the value of the next multiple of five that is larger than the height. Five goes into twelve twice. Add one to two, and multiply by five; the result is fifteen, which is the next multiple of five that is higher than twelve. The Lisp expression for this is:

```
(* (1+ (/ height 5)) 5)
```

For example, if you evaluate the following, the result is 15:

```
(* (1+ (/ 12 5)) 5)
```

All through this discussion, we have been using 5 as the value for spacing labels on the Y axis; but we may want to use some other value. For generality, we should replace 5 with a variable to which we can assign a value. The best name I can think of for this variable is `Y-axis-label-spacing`.

Using this term, and an `if` expression, we produce the following:

```
(if (zerop (% height Y-axis-label-spacing))
    height
  ;; else
  (* (1+ (/ height Y-axis-label-spacing))
     Y-axis-label-spacing))
```

This expression returns the value of `height` itself if the height is an even multiple of the value of the `Y-axis-label-spacing` or else it computes and returns a value of `height` that is equal to the next higher multiple of the value of the `Y-axis-label-spacing`.

We can now include this expression in the `let` expression of the `print-graph` function (after first setting the value of `Y-axis-label-spacing`):

```
(defvar Y-axis-label-spacing 5
  "Number of lines from one Y axis label to next.")

...
(let* ((height (apply 'max numbers-list))
       (height-of-top-line
        (if (zerop (% height Y-axis-label-spacing))
            height
          ;; else
          (* (1+ (/ height Y-axis-label-spacing))
             Y-axis-label-spacing)))
       (symbol-width (length graph-blank))))
...
```

(Note use of the `let*` function: the initial value of height is computed once by the `(apply 'max numbers-list)` expression and then the resulting value of `height` is used to compute its final value. Zie "The `let*` expression", pagina 150, for more about `let*`.)

C.2.2 Construct a Y Axis Element

When we print the vertical axis, we want to insert strings such as '`5 -`' and '`10 - `' every five lines. Moreover, we want the numbers and dashes to line up, so shorter numbers must be padded with leading spaces. If some of the strings use two digit numbers, the strings with single digit numbers must include a leading blank space before the number.

To figure out the length of the number, the `length` function is used. But the `length` function works only with a string, not with a number. So the number has to be converted from being a number to being a string. This is done with the `number-to-string` function. For example,

```
(length (number-to-string 35))
     ⇒ 2

(length (number-to-string 100))
     ⇒ 3
```

(`number-to-string` is also called `int-to-string`; you will see this alternative name in various sources.)

In addition, in each label, each number is followed by a string such as ‘ - ’, which we will call the `Y-axis-tic` marker. This variable is defined with `defvar`:

```
(defvar Y-axis-tic " - "
   "String that follows number in a Y axis label.")
```

The length of the Y label is the sum of the length of the Y axis tic mark and the length of the number of the top of the graph.

```
(length (concat (number-to-string height) Y-axis-tic)))
```

This value will be calculated by the `print-graph` function in its varlist as `full-Y-label-width` and passed on. (Note that we did not think to include this in the varlist when we first proposed it.)

To make a complete vertical axis label, a tic mark is concatenated with a number; and the two together may be preceded by one or more spaces depending on how long the number is. The label consists of three parts: the (optional) leading spaces, the number, and the tic mark. The function is passed the value of the number for the specific row, and the value of the width of the top line, which is calculated (just once) by `print-graph`.

```
(defun Y-axis-element (number full-Y-label-width)
  "Construct a NUMBERed label element.
A numbered element looks like this `  5 - ',
and is padded as needed so all line up with
the element for the largest number."
  (let* ((leading-spaces
         (- full-Y-label-width
            (length
             (concat (number-to-string number)
                     Y-axis-tic)))))
    (concat
     (make-string leading-spaces ? )
     (number-to-string number)
     Y-axis-tic)))
```

The `Y-axis-element` function concatenates together the leading spaces, if any; the number, as a string; and the tic mark.

To figure out how many leading spaces the label will need, the function subtracts the actual length of the label—the length of the number plus the length of the tic mark—from the desired label width.

Blank spaces are inserted using the `make-string` function. This function takes two arguments: the first tells it how long the string will be and the second is a symbol for the character to insert, in a special format. The format is a question mark followed by a blank space, like this, ‘`? `’. Zie Sectie “Character Type” in *The GNU Emacs Lisp Reference Manual*, for a description of the syntax for characters. (Of course, you might want to replace the blank space by some other character . . . You know what to do.)

The `number-to-string` function is used in the concatenation expression, to convert the number to a string that is concatenated with the leading spaces and the tic mark.

C.2.3 Create a Y Axis Column

The preceding functions provide all the tools needed to construct a function that generates a list of numbered and blank strings to insert as the label for the vertical axis:

```
(defun Y-axis-column (height width-of-label)
  "Construct list of Y axis labels and blank strings.
For HEIGHT of line above base and WIDTH-OF-LABEL."
  (let (Y-axis)

    (while (> height 1)
      (if (zerop (% height Y-axis-label-spacing))
          ;; Insert label.
          (setq Y-axis
                (cons
                 (Y-axis-element height width-of-label)
                 Y-axis))

        ;; Else, insert blanks.
        (setq Y-axis
              (cons
               (make-string width-of-label ? )
               Y-axis)))
      (setq height (1- height)))
    ;; Insert base line.
    (setq Y-axis
          (cons (Y-axis-element 1 width-of-label) Y-axis))
    (nreverse Y-axis)))
```

In this function, we start with the value of `height` and repetitively subtract one from its value. After each subtraction, we test to see whether the value is an integral multiple of the `Y-axis-label-spacing`. If it is, we construct a numbered label using the `Y-axis-element` function; if not, we construct a blank label using the `make-string` function. The base line consists of the number one followed by a tic mark.

C.2.4 The Not Quite Final Version of `print-Y-axis`

The list constructed by the `Y-axis-column` function is passed to the `print-Y-axis` function, which inserts the list as a column.

```
(defun print-Y-axis (height full-Y-label-width)
  "Insert Y axis using HEIGHT and FULL-Y-LABEL-WIDTH.
Height must be the maximum height of the graph.
Full width is the width of the highest label element."
;; Value of height and full-Y-label-width
;; are passed by print-graph.
  (let ((start (point)))
    (insert-rectangle
     (Y-axis-column height full-Y-label-width))
    ;; Place point ready for inserting graph.
    (goto-char start)
    ;; Move point forward by value of full-Y-label-width
    (forward-char full-Y-label-width)))
```

The `print-Y-axis` uses the `insert-rectangle` function to insert the Y axis labels created by the `Y-axis-column` function. In addition, it places point at the correct position for printing the body of the graph.

You can test `print-Y-axis`:

1. Installeer

   ```
   Y-axis-label-spacing
   Y-axis-tic
   Y-axis-element
   Y-axis-column
   print-Y-axis
   ```

2. Kopieer de volgende expressie:

   ```
   (print-Y-axis 12 5)
   ```

3. Switch to the `*scratch*` buffer and place the cursor where you want the axis labels to start.
4. Typ *`M-:`* (`eval-expression`).
5. Yank de `grafiek-body-tonen` expressie in het minibuffer met *`C-y`* (`yank`).
6. Press `RET` to evaluate the expression.

Emacs will print labels vertically, the top one being '`10 - `'. (The `print-graph` function will pass the value of `height-of-top-line`, which in this case will end up as 15, thereby getting rid of what might appear as a bug.)

C.3 De print-X-axis functie

X axis labels are much like Y axis labels, except that the ticks are on a line above the numbers. Labels should look like this:

```
|   |     |     |
1   5    10    15
```

The first tic is under the first column of the graph and is preceded by several blank spaces. These spaces provide room in rows above for the Y axis labels. The second, third, fourth, and subsequent ticks are all spaced equally, according to the value of X-axis-label-spacing.

The second row of the X axis consists of numbers, preceded by several blank spaces and also separated according to the value of the variable X-axis-label-spacing.

The value of the variable X-axis-label-spacing should itself be measured in units of symbol-width, since you may want to change the width of the symbols that you are using to print the body of the graph without changing the ways the graph is labeled.

The print-X-axis function is constructed in more or less the same fashion as the print-Y-axis function except that it has two lines: the line of tic marks and the numbers. We will write a separate function to print each line and then combine them within the print-X-axis function.

This is a three step process:

1. Write a function to print the X axis tic marks, print-X-axis-tic-line.
2. Write a function to print the X numbers, print-X-axis-numbered-line.
3. Write a function to print both lines, the print-X-axis function, using print-X-axis-tic-line and print-X-axis-numbered-line.

C.3.1 X-as tic marks

The first function should print the X axis tic marks. We must specify the tic marks themselves and their spacing:

```
(defvar X-axis-label-spacing
  (if (boundp 'graph-blank)
      (* 5 (length graph-blank)) 5)
  "Number of units from one X axis label to next.")
```

(Note that the value of graph-blank is set by another defvar. The boundp predicate checks whether it has already been set; boundp returns nil if it has not. If graph-blank were unbound and we did not use this conditional construction, we would enter the debugger and see an error message saying 'Debugger entered--Lisp error: (void-variable graph-blank)'.)

Here is the `defvar` for `X-axis-tic-symbol`:

```
(defvar X-axis-tic-symbol "|"
  "String to insert to point to a column in X axis.")
```

The goal is to make a line that looks like this:

```
        |   |    |    |
```

The first tic is indented so that it is under the first column, which is indented to provide space for the Y axis labels.

A tic element consists of the blank spaces that stretch from one tic to the next plus a tic symbol. The number of blanks is determined by the width of the tic symbol and the `X-axis-label-spacing`.

De code ziet er zo uit:

```
;;; X-axis-tic-element
...
(concat
 (make-string
  ;; Make a string of blanks.
  (-  (* symbol-width X-axis-label-spacing)
      (length X-axis-tic-symbol))
  ? )
 ;; Concatenate blanks with tic symbol.
 X-axis-tic-symbol)
...
```

Next, we determine how many blanks are needed to indent the first tic mark to the first column of the graph. This uses the value of `full-Y-label-width` passed it by the `print-graph` function.

The code to make `X-axis-leading-spaces` looks like this:

```
;; X-axis-leading-spaces
...
(make-string full-Y-label-width ? )
...
```

We also need to determine the length of the horizontal axis, which is the length of the numbers list, and the number of ticks in the horizontal axis:

```
;; X-length
...
(length numbers-list)

;; tic-width
...
(* symbol-width X-axis-label-spacing)

;; number-of-X-ticks
(if (zerop (% (X-length tic-width)))
    (/ (X-length tic-width))
  (1+ (/ (X-length tic-width))))
```

All this leads us directly to the function for printing the X axis tic line:

```
(defun print-X-axis-tic-line
  (number-of-X-tics X-axis-leading-spaces X-axis-tic-element)
  "Print ticks for X axis."
    (insert X-axis-leading-spaces)
    (insert X-axis-tic-symbol)  ; Under first column.
    ;; Insert second tic in the right spot.
    (insert (concat
             (make-string
              (-  (* symbol-width X-axis-label-spacing)
                  ;; Insert white space up to second tic symbol.
                  (* 2 (length X-axis-tic-symbol)))
              ? )
             X-axis-tic-symbol))
    ;; Insert remaining ticks.
    (while (> number-of-X-tics 1)
      (insert X-axis-tic-element)
      (setq number-of-X-tics (1- number-of-X-tics))))
```

The line of numbers is equally straightforward:

First, we create a numbered element with blank spaces before each number:

```
(defun X-axis-element (number)
  "Construct a numbered X axis element."
  (let ((leading-spaces
         (-  (* symbol-width X-axis-label-spacing)
             (length (number-to-string number)))))
    (concat (make-string leading-spaces ? )
            (number-to-string number))))
```

Next, we create the function to print the numbered line, starting with the number 1 under the first column:

```
(defun print-X-axis-numbered-line
  (number-of-X-tics X-axis-leading-spaces)
  "Print line of X-axis numbers"
  (let ((number X-axis-label-spacing))
    (insert X-axis-leading-spaces)
    (insert "1")
    (insert (concat
             (make-string
              ;; Insert white space up to next number.
              (-  (* symbol-width X-axis-label-spacing) 2)
              ? )
             (number-to-string number)))
    ;; Insert remaining numbers.
    (setq number (+ number X-axis-label-spacing))
    (while (> number-of-X-tics 1)
      (insert (X-axis-element number))
      (setq number (+ number X-axis-label-spacing))
      (setq number-of-X-tics (1- number-of-X-tics)))))
```

Finally, we need to write the `print-X-axis` that uses `print-X-axis-tic-line` and `print-X-axis-numbered-line`.

The function must determine the local values of the variables used by both `print-X-axis-tic-line` and `print-X-axis-numbered-line`, and then it must call them. Also, it must print the carriage return that separates the two lines.

The function consists of a varlist that specifies five local variables, and calls to each of the two line printing functions:

```
(defun print-X-axis (numbers-list)
  "Print X axis labels to length of NUMBERS-LIST."
  (let* ((leading-spaces
          (make-string full-Y-label-width ? ))
         ;; symbol-width is provided by graph-body-print
         (tic-width (* symbol-width X-axis-label-spacing))
         (X-length (length numbers-list))
         (X-tic
          (concat
           (make-string
            ;; Make a string of blanks.
            (-  (* symbol-width X-axis-label-spacing)
                (length X-axis-tic-symbol))
            ? )
```

```
      ;; Concatenate blanks with tic symbol.
      X-axis-tic-symbol))
    (tic-number
     (if (zerop (% X-length tic-width))
         (/ X-length tic-width)
       (1+ (/ X-length tic-width)))))
  (print-X-axis-tic-line tic-number leading-spaces X-tic)
  (insert "\n")
  (print-X-axis-numbered-line tic-number leading-spaces)))
```

You can test `print-X-axis`:

1. Install `X-axis-tic-symbol`, `X-axis-label-spacing`, `print-X-axis-tic-line`, as well as `X-axis-element`, `print-X-axis-numbered-line`, and `print-X-axis`.
2. Kopieer de volgende expressie:

```
(progn
 (let ((full-Y-label-width 5)
       (symbol-width 1))
   (print-X-axis
    '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16))))
```

3. Switch to the `*scratch*` buffer and place the cursor where you want the axis labels to start.
4. Typ *M-:* (`eval-expression`).
5. Yank the test expression into the minibuffer with *C-y* (`yank`).
6. Press `RET` to evaluate the expression.

Emacs will print the horizontal axis like this:

```
|   |    |    |    |
1   5   10   15   20
```

C.4 De hele grafiek afdrukken

Now we are nearly ready to print the whole graph.

The function to print the graph with the proper labels follows the outline we created earlier (zie ⟨undefined⟩ “A Graph with Labeled Axes”, pagina ⟨undefined⟩), but with additions.

Here is the outline:

```
(defun print-graph (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))
    (print-Y-axis height ... )
    (graph-body-print numbers-list)
    (print-X-axis ... )))
```

The final version is different from what we planned in two ways: first, it contains additional values calculated once in the varlist; second, it carries an option to specify the labels' increment per row. This latter feature turns out to be essential; otherwise, a graph may have more rows than fit on a display or on a sheet of paper.

This new feature requires a change to the `Y-axis-column` function, to add `vertical-step` to it. The function looks like this:

```
;;; Final version.
(defun Y-axis-column
  (height width-of-label &optional vertical-step)
  "Construct list of labels for Y axis.
HEIGHT is maximum height of graph.
WIDTH-OF-LABEL is maximum width of label.
VERTICAL-STEP, an option, is a positive integer
that specifies how much a Y axis label increments
for each line.  For example, a step of 5 means
that each line is five units of the graph."
  (let (Y-axis
        (number-per-line (or vertical-step 1)))
    (while (> height 1)
      (if (zerop (% height Y-axis-label-spacing))
          ;; Insert label.
          (setq Y-axis
                (cons
                 (Y-axis-element
                  (* height number-per-line)
                  width-of-label)
                 Y-axis))
        ;; Else, insert blanks.
        (setq Y-axis
              (cons
               (make-string width-of-label ? )
               Y-axis)))
      (setq height (1- height)))
    ;; Insert base line.
    (setq Y-axis (cons (Y-axis-element
                        (or vertical-step 1)
                        width-of-label)
                       Y-axis))
    (nreverse Y-axis)))
```

The values for the maximum height of graph and the width of a symbol are computed by `print-graph` in its `let` expression; so `graph-body-print` must be changed to accept them.

```
;;; Final version.
(defun graph-body-print (numbers-list height symbol-width)
  "Print a bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values.
HEIGHT is maximum height of graph.
SYMBOL-WIDTH is number of each column."
  (let (from-position)
    (while numbers-list
      (setq from-position (point))
      (insert-rectangle
       (column-of-graph height (car numbers-list)))
      (goto-char from-position)
      (forward-char symbol-width)
```

```
        ;; Draw graph column by column.
        (sit-for 0)
        (setq numbers-list (cdr numbers-list)))
      ;; Place point for X axis labels.
      (forward-line height)
      (insert "\n")))
```

Finally, the code for the `print-graph` function:

```
;;; Final version.
(defun print-graph
  (numbers-list &optional vertical-step)
  "Print labeled bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values.

Optionally, VERTICAL-STEP, a positive integer,
specifies how much a Y axis label increments for
each line.  For example, a step of 5 means that
each row is five units."
  (let* ((symbol-width (length graph-blank))
         ;; height is both the largest number
         ;; and the number with the most digits.
         (height (apply 'max numbers-list))
         (height-of-top-line
          (if (zerop (% height Y-axis-label-spacing))
              height
            ;; else
            (* (1+ (/ height Y-axis-label-spacing))
               Y-axis-label-spacing)))
         (vertical-step (or vertical-step 1))
         (full-Y-label-width
          (length
           (concat
            (number-to-string
             (* height-of-top-line vertical-step))
            Y-axis-tic))))

    (print-Y-axis
     height-of-top-line full-Y-label-width vertical-step)
    (graph-body-print
     numbers-list height-of-top-line symbol-width)
    (print-X-axis numbers-list)))
```

C.4.1 Testing `print-graph`

We can test the `print-graph` function with a short list of numbers:

1. Install the final versions of `Y-axis-column`, `graph-body-print`, and `print-graph` (in addition to the rest of the code.)
2. Kopieer de volgende expressie:

   ```
   (print-graph '(3 2 5 6 7 5 3 4 6 4 3 2 1))
   ```

3. Switch to the `*scratch*` buffer and place the cursor where you want the axis labels to start.

4. Typ *M-:* (`eval-expression`).
5. Yank the test expression into the minibuffer with *C-y* (`yank`).
6. Press `RET` to evaluate the expression.

Emacs will print a graph that looks like this:

```
10 -


          *
         **   *
 5 -     ****  *
         **** ***
       * *********
       ************
 1 - *************

       |   |    |    |
       1   5   10   15
```

On the other hand, if you pass `print-graph` a `vertical-step` value of 2, by evaluating this expression:

```
(print-graph '(3 2 5 6 7 5 3 4 6 4 3 2 1) 2)
```

The graph looks like this:

```
20 -


          *
         **   *
10 -     ****  *
         **** ***
       * *********
       ************
 2 - *************

       |   |    |    |
       1   5   10   15
```

(A question: is the '2' on the bottom of the vertical axis a bug or a feature? If you think it is a bug, and should be a '1' instead, (or even a '0'), you can modify the sources.)

C.4.2 Graphing Numbers of Words and Symbols

Now for the graph for which all this code was written: a graph that shows how many function definitions contain fewer than 10 words and symbols, how many contain between 10 and 19 words and symbols, how many contain between 20 and 29 words and symbols, and so on.

This is a multi-step process. First make sure you have loaded all the requisite code.

It is a good idea to reset the value of `top-of-ranges` in case you have set it to some different value. You can evaluate the following:

```
(setq top-of-ranges
 '(10  20  30  40  50
   60  70  80  90 100
  110 120 130 140 150
  160 170 180 190 200
  210 220 230 240 250
  260 270 280 290 300)
```

Next create a list of the number of words and symbols in each range.
Evaluate the following:

```
(setq list-for-graph
       (defuns-per-range
         (sort
          (recursive-lengths-list-many-files
           (directory-files "/usr/local/emacs/lisp"
                            t ".+el$"))
          '<)
         top-of-ranges))
```

On my old machine, this took about an hour. It looked though 303 Lisp files in my copy of Emacs version 19.23. After all that computing, the `list-for-graph` had this value:

```
(537 1027 955 785 594 483 349 292 224 199 166 120 116 99
90 80 67 48 52 45 41 33 28 26 25 20 12 28 11 13 220)
```

This means that my copy of Emacs had 537 function definitions with fewer than 10 words or symbols in them, 1,027 function definitions with 10 to 19 words or symbols in them, 955 function definitions with 20 to 29 words or symbols in them, and so on.

Clearly, just by looking at this list we can see that most function definitions contain ten to thirty words and symbols.

Now for printing. We do *not* want to print a graph that is 1,030 lines high ... Instead, we should print a graph that is fewer than twenty-five lines high. A graph that height can be displayed on almost any monitor, and easily printed on a sheet of paper.

This means that each value in `list-for-graph` must be reduced to one-fiftieth its present value.

Here is a short function to do just that, using two functions we have not yet seen, `mapcar` and `lambda`.

```
(defun one-fiftieth (full-range)
  "Return list, each number one-fiftieth of previous."
 (mapcar (lambda (arg) (/ arg 50)) full-range))
```

C.4.3 A `lambda` Expression: Useful Anonymity

`lambda` is the symbol for an anonymous function, a function without a name. Every time you use an anonymous function, you need to include its whole body.

Dus

```
(lambda (arg) (/ arg 50))
```

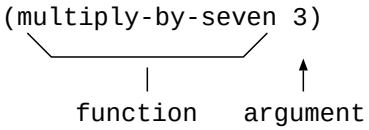
is a function that returns the value resulting from dividing whatever is passed to it as `arg` by 50.

Earlier, for example, we had a function `multiply-by-seven`; it multiplied its argument by 7. This function is similar, except it divides its argument by 50; and, it has no name. The anonymous equivalent of `multiply-by-seven` is:

```
(lambda (number) (* 7 number))
```

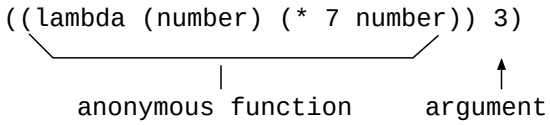
(Zie Sectie 3.1 “The `defun` Macro”, pagina 26.)

If we want to multiply 3 by 7, we can write:

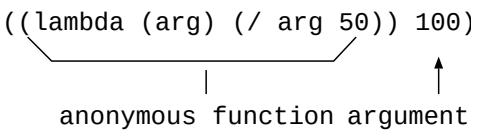


This expression returns 21.

Similarly, we can write:



If we want to divide 100 by 50, we can write:



This expression returns 2. The 100 is passed to the function, which divides that number by 50.

Zie Sectie "Lambda Expressions" in *The GNU Emacs Lisp Reference Manual*, for more about `lambda`. Lisp and lambda expressions derive from the Lambda Calculus.

C.4.4 The `mapcar` Function

`mapcar` is a function that calls its first argument with each element of its second argument, in turn. The second argument must be a sequence.

The '`map`' part of the name comes from the mathematical phrase, "mapping over a domain", meaning to apply a function to each of the elements in a domain. The mathematical phrase is based on the metaphor of a surveyor walking, one step at a time, over an area he is mapping. And '`car`', of course, comes from the Lisp notion of the first of a list.

Bijvoorbeeld,

```
(mapcar '1+ '(2 4 6))
     ⇒ (3 5 7)
```

The function `1+` which adds one to its argument, is executed on *each* element of the list, and a new list is returned.

Contrast this with `apply`, which applies its first argument to all the remaining. (Zie ⟨undefined⟩ "Readying a Graph", pagina ⟨undefined⟩, for an explanation of `apply`.)

In the definition of `one-fiftieth`, the first argument is the anonymous function:

```
(lambda (arg) (/ arg 50))
```

and the second argument is `full-range`, which will be bound to `list-for-graph`.

The whole expression looks like this:

```
(mapcar (lambda (arg) (/ arg 50)) full-range))
```

Zie Sectie "Mapping Functions" in *The GNU Emacs Lisp Reference Manual*, for more about `mapcar`.

Using the `one-fiftieth` function, we can generate a list in which each element is one-fiftieth the size of the corresponding element in `list-for-graph`.

```
(setq fiftieth-list-for-graph
      (one-fiftieth list-for-graph))
```

The resulting list looks like this:

```
(10 20 19 15 11 9 6 5 4 3 3 2 2
1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 4)
```

This, we are almost ready to print! (We also notice the loss of information: many of the higher ranges are 0, meaning that fewer than 50 defuns had that many words or symbols—but not necessarily meaning that none had that many words or symbols.)

C.4.5 Another Bug . . . Most Insidious

I said "almost ready to print"! Of course, there is a bug in the `print-graph` function . . . It has a `vertical-step` option, but not a `horizontal-step` option. The `top-of-range` scale goes from 10 to 300 by tens. But the `print-graph` function will print only by ones.

This is a classic example of what some consider the most insidious type of bug, the bug of omission. This is not the kind of bug you can find by studying the code, for it is not in the code; it is an omitted feature. Your best actions are to try your program early and often; and try to arrange, as much as you can, to write code that is easy to understand and easy to change. Try to be aware, whenever you can, that whatever you have written, *will* be rewritten, if not soon, eventually. A hard maxim to follow.

It is the `print-X-axis-numbered-line` function that needs the work; and then the `print-X-axis` and the `print-graph` functions need to be adapted. Not much needs to be done; there is one nicety: the numbers ought to line up under the tic marks. This takes a little thought.

Here is the corrected `print-X-axis-numbered-line`:

```
(defun print-X-axis-numbered-line
  (number-of-X-tics X-axis-leading-spaces
   &optional horizontal-step)
  "Print line of X-axis numbers"
  (let ((number X-axis-label-spacing)
        (horizontal-step (or horizontal-step 1)))
    (insert X-axis-leading-spaces)
    ;; Delete extra leading spaces.
    (delete-char
     (- (1-
         (length (number-to-string horizontal-step)))))
    (insert (concat
             (make-string
              ;; Insert white space.
              (-  (* symbol-width
                     X-axis-label-spacing)
                  (1-
                   (length
                    (number-to-string horizontal-step)))
                  2)
              ? )
             (number-to-string
              (* number horizontal-step))))
    ;; Insert remaining numbers.
    (setq number (+ number X-axis-label-spacing))
    (while (> number-of-X-tics 1)
      (insert (X-axis-element
               (* number horizontal-step)))
      (setq number (+ number X-axis-label-spacing))
      (setq number-of-X-tics (1- number-of-X-tics)))))
```

If you are reading this in Info, you can see the new versions of `print-X-axis` `print-graph` and evaluate them. If you are reading this in a printed book, you can see the changed lines here (the full text is too much to print).

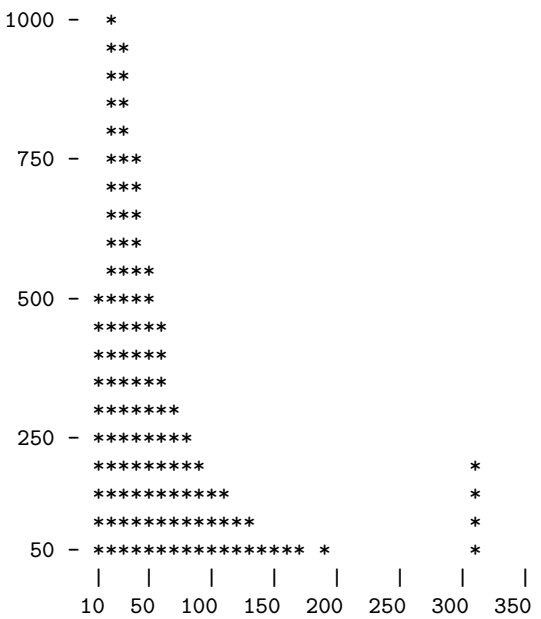
```
(defun print-X-axis (numbers-list horizontal-step)
  ...
    (print-X-axis-numbered-line
     tic-number leading-spaces horizontal-step))
(defun print-graph
  (numbers-list
   &optional vertical-step horizontal-step)
  ...
    (print-X-axis numbers-list horizontal-step))
```

C.4.6 The Printed Graph

When made and installed, you can call the `print-graph` command like this:

```
(print-graph fiftieth-list-for-graph 50 10)
```

Here is the graph:



The largest group of functions contain 10–19 words and symbols each.

Appendix D Vrije software en vrije handleidingen

by Richard M. Stallman

The biggest deficiency in free operating systems is not in the software—it is the lack of good free manuals that we can include in these systems. Many of our most important programs do not come with full manuals. Documentation is an essential part of any software package; when an important free software package does not come with a free manual, that is a major gap. We have many such gaps today.

Once upon a time, many years ago, I thought I would learn Perl. I got a copy of a free manual, but I found it hard to read. When I asked Perl users about alternatives, they told me that there were better introductory manuals—but those were not free.

Why was this? The authors of the good manuals had written them for O'Reilly Associates, which published them with restrictive terms—no copying, no modification, source files not available—which exclude them from the free software community.

That wasn't the first time this sort of thing has happened, and (to our community's great loss) it was far from the last. Proprietary manual publishers have enticed a great many authors to restrict their manuals since then. Many times I have heard a GNU user eagerly tell me about a manual that he is writing, with which he expects to help the GNU project—and then had my hopes dashed, as he proceeded to explain that he had signed a contract with a publisher that would restrict it so that we cannot use it.

Given that writing good English is a rare skill among programmers, we can ill afford to lose manuals this way.

Free documentation, like free software, is a matter of freedom, not price. The problem with these manuals was not that O'Reilly Associates charged a price for printed copies—that in itself is fine. The Free Software Foundation sells printed copies (`https://shop.fsf.org`) of free GNU manuals (`https://www.gnu.org/doc/doc.html`), too. But GNU manuals are available in source code form, while these manuals are available only on paper. GNU manuals come with permission to copy and modify; the Perl manuals do not. These restrictions are the problems.

The criterion for a free manual is pretty much the same as for free software: it is a matter of giving all users certain freedoms. Redistribution (including commercial redistribution) must be permitted, so that the manual can accompany every copy of the program, on-line or on paper. Permission for modification is crucial too.

As a general rule, I don't believe that it is essential for people to have permission to modify all sorts of articles and books. The issues for writings are not necessarily the same as those for software. For example, I don't think you or I are obliged to give permission to modify articles like this one, which describe our actions and our views.

But there is a particular reason why the freedom to modify is crucial for documentation for free software. When people exercise their right to modify the software, and add or change its features, if they are conscientious they will change the manual too—so they can provide accurate and usable documentation with the modified

program. A manual which forbids programmers to be conscientious and finish the job, or more precisely requires them to write a new manual from scratch if they change the program, does not fill our community's needs.

While a blanket prohibition on modification is unacceptable, some kinds of limits on the method of modification pose no problem. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are ok. It is also no problem to require modified versions to include notice that they were modified, even to have entire sections that may not be deleted or changed, as long as these sections deal with nontechnical topics. (Some GNU manuals have them.)

These kinds of restrictions are not a problem because, as a practical matter, they don't stop the conscientious programmer from adapting the manual to fit the modified program. In other words, they don't block the free software community from making full use of the manual.

However, it must be possible to modify all the technical content of the manual, and then distribute the result in all the usual media, through all the usual channels; otherwise, the restrictions do block the community, the manual is not free, and so we need another manual.

Unfortunately, it is often hard to find someone to write another manual when a proprietary manual exists. The obstacle is that many users think that a proprietary manual is good enough—so they don't see the need to write a free manual. They do not see that the free operating system has a gap that needs filling.

Why do users think that proprietary manuals are good enough? Some have not considered the issue. I hope this article will do something to change that.

Other users consider proprietary manuals acceptable for the same reason so many people consider proprietary software acceptable: they judge in purely practical terms, not using freedom as a criterion. These people are entitled to their opinions, but since those opinions spring from values which do not include freedom, they are no guide for those of us who do value freedom.

Please spread the word about this issue. We continue to lose manuals to proprietary publishing. If we spread the word that proprietary manuals are not sufficient, perhaps the next person who wants to help GNU by writing documentation will realize, before it is too late, that he must above all make it free.

We can also encourage commercial publishers to sell free, copylefted manuals instead of proprietary ones. One way you can help this is to check the distribution terms of a manual before you buy it, and prefer copylefted manuals to non-copylefted ones.

Opmerking: De Free Software Foundation onderhoudt een pagina op hun website met een lijst van boeken die vrij beschikbaar zijn bij andere uitgevers:
`https://www.gnu.org/doc/other-free-books.html`

Appendix E GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance

and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a

unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with

a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Over de auteur

Robert J. Chassell (1946–2017) begon te werken met GNU Emacs in 1985. Hij schreef en bewerte tekst, gaf les in Emacs en Emacs Lisp en sprak over de hele wereld over software vrijheid. Chassel was oprichtend directeur en penningmeester van de Free Software Foundation, Inc. He was co-auteur van de *Texinfo* handleiding, en bewerkte meer dan een dozijn andere boeken. Hij studeerde af aan de universiteit van Cambridge in Engeland. Hij had een blijvende interesse in sociale en economische geshiedenis en vloog zijn eigen vliegtuig.
"Goodbye to Bob Chassell" (`https://www.fsf.org/blogs/community/goodbye-to-bob-chassell`)