Programmatuur C1

Inhoudsopgave

1 Inleiding **C1/3**
- 1.1 Apparatuur versus programmatuur C1/4
- 1.2 Generaties van programmeertalen C1/5
- 1.3 Het niveau van programmeertalen C1/6
- 1.4 Families van programmeertalen C1/7
- 1.5 Toepassingsgebieden C1/8
- 1.6 Uitvoeringswijze C1/9

2 Concepten van programmeertalen **C1/10**
- 2.1 Data C1/11
- 2.2 Het imperatieve model C1/15
- 2.3 Belangrijke imperatieve talen C1/22
- 2.4 Object-oriëntatie C1/24
- 2.5 Belangrijke object-georiënteerde talen . . . C1/32
- 2.6 Functioneel programmeren C1/34
- 2.7 Diverse programmeerstijlen C1/38

3 Uitvoeren van programma's **C1/43**
- 3.1 Interpretators C1/46
- 3.2 Compilers C1/47
- 3.3 Hybride systemen C1/59
- 3.4 T-diagrammen C1/61

4 Programmageneratoren **C1/63**
- 4.1 Scannergeneratoren C1/64
- 4.2 Parsergeneratoren C1/66
- 4.3 Codegeneratorgeneratoren C1/67
- 4.4 Applicatiegeneratoren C1/68

5 Programmeeromgevingen **C1/69**
- 5.1 Editors C1/69
- 5.2 Test tools C1/70
- 5.3 Debuggers C1/71
- 5.4 Prestatiemetingen C1/72
- 5.5 Configuratie- en versiebeheer C1/73

6 Literatuur **C1/73**

1 Inleiding

Een programma is een recept dat aangeeft welke elementaire instructies een computer uit moet voeren om een gegeven probleem op te lossen. Van Dale's Groot Woordenboek der Nederlandse Taal zegt het zo: *een programma is een serie van gecodeerde instructies volgens welke een elektronisch apparaat de gegevens in een bepaald probleem moet bewerken*. In het algemeen zal het "elektronische apparaat" dat door het programma geïnstrueerd wordt een mainframe of een personal computer zijn. Het kan echter ook een printer, telefoon, of televisietoestel zijn.

Voor het schrijven van programma's wordt gebruik gemaakt van speciaal ontwikkelde talen. Dergelijke *programmeertalen* moeten aan ten minste twee eisen voldoen. Enerzijds moeten ze efficient door een computer uitgevoerd kunnen worden. Anderzijds moeten ze dusdanige "woorden" en "zinsconstructies" bevatten dat een programmeur het op te lossen probleem op een natuurlijke manier kan beschrijven.

Het spanningsveld dat hierdoor ontstaat — moet een programmeertaal dicht bij de programmeur of juist dicht bij de computer staan — is het onderwerp van studie van het vakgebied *programmatuur*. Vragen op het gebied van de programmatuur zijn onder meer: In welke taal zijn programma's geschreven? Hoe zijn ze tot standgekomen? Wat is hun betekenis? Hoe wordt een programma omgezet in machine-instructies? Hoe kunnen we de correctheid van een programma beoordelen?

In dit hoofdstuk concentreren we ons op vier belangrijke aspekten van programmatuur:

- De *concepten* waarop programmeertalen zijn gebaseerd (Sectie 2). De belangrijkste families van programmeertalen en hun onderliggende ideeën komen aan de orde. Dit leidt tot inzicht in de keuze van de beste programmeertaal voor een gegegeven probleem en helpt bij het beoordelen van nieuwe programmeertalen.

- De wijze waarop programma's omgezet worden in "een rij gecodeerde instructies" die direkt door de computer uitgevoerd kunnen worden, kortom de *uitvoering* van programma's (Sectie 3). We behandelen de interne struktuur van compilers en interpretators en geven hun voor- en nadelen. Inzicht in de gebruikte technieken helpt bij het selecteren van hulpmiddelen (tools) voor nieuwbouw en renovatie van programmatuur en ook bij het zelf bouwen van dergelijke tools.

- *Programmageneratoren* die het mogelijk maken om specifieke programmeerwerkzaamheden te automatiseren (Sectie 4). Aan de orde komen generatoren voor het produceren van onderdelen van compilers en voor het genereren van code voor domeinspecifieke talen.

- De *programmeeromgeving* waarbinnen programma's ontwikkeld worden (Sectie 5). Dit geeft inzicht in de hulpmiddelen die beschikbaar zijn voor het uitvoeren van diverse taken tijdens programmatuurontwikkeling, zoals editen, testen, debuggen, prestatiemeting, en configuratie- en versiebeheer.

Dit hoofdstuk concentreert zich dus op programmeertalen, en de gereedschappen die bij het gebruik van programmeertalen een rol spelen.

Een vraag waar we in dit hoofdstuk niet op ingaan is op welke wijze een correct werkend programam het beste ontwikkeld kan worden. Deze meer methodische vraag komt aan de orde in het hoofdstuk SOFTWARE ENGINEERING.

Voordat we de vier aspekten van programmatuur in meer detail gaan bespreken, geven we eerst nog een aantal invalshoeken van waaruit programmatuur bekeken kan worden.

1.1 Apparatuur versus programmatuur

In eerste benadering lijkt het zinvol om een onderscheid te maken tussen apparatuur (de hardware) en programmatuur (de software). Bij nadere inspektie, blijkt veel apparatuur zelf ook weer programmatuur te bevatten. Denk hierbij aan randapparaten zoals modems, netwerkcontrollers, printers, en disks. Zelfs als we het hart van een computer bekijken, blijkt de CPU zelf ook vaak weer programmeerbaar te zijn met *microcode* die de werking van de machine-instructies vastlegt.

De programmatuur van een totaal computersysteem is opgebouwd uit een aantal lagen: een besturingssysteem (dat het toegangsbeheer en de besturing van de apparatuur regelt); basisfunctionaliteit zoals netwerkbeheer en communicatie, middleware, transactiemonitors, en databases; en tenslotte de applicatieprogramma's die gebruikerstaken verrichten in specifieke toepassingsgebieden.

1.2 Generaties van programmeertalen

Programmeertalen worden meestal in *generaties* ingedeeld.

De eerste-generatie talen (*First Generation Language*: 1GL) wordt gevormd door de machinetalen, d.w.z. de binair gecodeerde rijen machine-instructies die direct door de hardware uitgevoerd kunnen worden.

Doordat het programmeren in machinetalen erg moeizaam was, zijn de tweede-generatie talen (2GL) ontstaan. Dit zijn de assemblertalen, waarin machine-instructies aangegeven kunnen worden met voor de programmeur leesbare namen en waarin ook aan adressen symbolische namen gegegeven kunnen worden.

Programmeertalen uit de derde generatie (3GL) brengen een verdere vereenvoudiging aan in het programmeren: veel voorkomende programmapatronen zoals keuze, herhaling en procedureaanroep zijn hierin al voorgedefinieerd en hoeven dus niet meer door de programmeur steeds opnieuw in assembler uitgeschreven te worden. Programmeertalen als Fortran, C en COBOL horen tot deze generatie.

Verdere automatisering van het programmeren wordt geboden door vierde-generatie talen (4GL), die voor specifieke toepassingen ondersteuning bieden, zoals rapportgeneratie en het gebruik van gegegevensbestanden. De meeste 4GL's zijn uitbreidingen van een 3GL en worden door middel van een voorbereidende fase (*preprocessor*) in de onderliggende 3GL omgezet.

Tenslotte richten de vijfde-generatie talen zich op het schrijven van programma's van nog hoger niveau. In plaats van een pro-

Taal	laag	gemiddeld	hoog
Macro assembler (1950)	220	320	500
C (1978)	60	128	170
BASIC (1964)	70	128	165
FORTRAN (1955)	75	107	160
COBOL (1960)	65	107	150
PASCAL (1971)	50	91	125
C++ (1986)	30	53	125
Visual Basic (1991)	20	32	37

Figuur 1.1: Statements per functiepunt in diverse talen.

bleem*oplossing* worden in deze talen de *randvoorwaarden* beschreven waaraan een oplossing moet voldoen, en genereert het ondersteunende systeem oplossingen die aan de gestelde randvoorwaarden voldoen.

1.3 Het niveau van programmeertalen

Programmeertalen kunnen onderscheiden worden naar hun uitdrukkingskracht. In *laag-niveau* talen is er veel programmatekst nodig om weinig functionaliteit uit te drukken. In *hoog-niveau* talen ligt dit juist omgekeerd: met weinig programmatekst wordt veel functionaliteit uitgedrukt. Welk niveau hebben de diverse programmeertalen?

Bij het beantwoorden van deze vraag stuiten we direct op het probleem hoe we hoeveelheden software eigenlijk moeten uitdrukken. In het aantal tekens waar het programma uit bestaat? In het aantal regels? In het aantal pagina's? Ieder van deze eenheden heeft het bezwaar dat hij erg afhangt van de specifieke programmeertaal *en* van de stijl van de individuele programmeur. Gelukkig bestaat er een internationale eenheid om hoeveelheden software in uit te drukken: het *functiepunt*. Functiepunten houden rekening met inhoudelijke aspecten van een programma zoals het aantal invoer- en uitvoerbestanden, het aantal koppelingen met andere programma's en dergelijke. Een functiepunt komt in verschillende programmeertalen overeen met verschillende aantallen *statements*: de elementaire instrukties waaruit een programma bestaat zoals toekenningen, tests en herhalingen. Let op dat het er niet toe doet over hoeveel regels een statement verspreid staat, het telt slechts voor één statement. Vaak wordt overigens bij conventie elk statement op een aparte regel geplaatst en dan komen aantallen statements en aantallen regels uiteraard weer met elkaar overeen.

Figuur 1.1 geeft gemiddelde waarden (en beneden- en bovengrenzen) voor diverse talen, zoals in 1997 bepaald door Capers Jones. Tussen haakjes staat in deze figuur het jaar van introductie van elke taal. De intuïtie dat een taal van hoger niveau is wanneer *minder* statements nodig zijn om *meer* uit te drukken wordt door deze figuur goed geïllustreerd: in een laag-niveau taal als Macro assembler of C zijn veel meer statements nodig om een functiepunt te implementeren dan in een hoger-niveau taal zoals C++ of Visual Basic. Op grond van deze cijfers zou het het beste zijn om alle programma's in Visual Basic te schrijven.[1]

1.4 Families van programmeertalen

In de loop der tijden zijn verschillende families van programmeertalen ontstaan die op verschillende principes (ook wel *paradigma* genoemd) gebaseerd zijn.

Het bekendste, en oudste, zijn de *imperatieve talen*. Hierin speelt het inspecteren en wijzigen van variabelen (die samen een *programmatoestand* vormen) een sleutelrol. Voorbeelden zijn talen als Fortran, Cobol, Pascal en C. Sterk verwant aan de imperatieve talen zijn de *object-georiënteerde* talen waarin de globale programmatoestand vervangen is door een verzameling objecten die ieder een eigen toestand bezitten. Objecten kunnen eenvoudig dienen voor het modelleren van entiteiten in de fysieke wereld, en leiden ook tot een meer modulaire opbouw van programma's. Voorbeelden zijn Smalltalk, C++ en Java.

Naast de imperatieve en objectgeoriënteerde talen bestaan er andere benaderingen die tot doel hebben om de transparantie van programma's te vergroten en de wiskundige analyse ervan mogelijk te maken. Dergelijke talen hebben meer een *declaratief*

[1] Uiteraard spelen ook andere overwegingen een rol bij de keuze van een programmeertaal zoals robuustheid, leesbaarheid, en onderhoudbaarheid.

karakter: het in detail uitschrijven van de probleemoplossing wordt vervangen door het opschrijven van alleen de voorwaarden waaraan een oplossing moet voldoen.

In *functionele talen* is het begrip programmatoestand geheel uitgebannen doordat alle berekeningen uitgedrukt worden als het toepassen van een functie op een aantal argumenten. Het resultaat van een functie hangt alleen af van de waarden van zijn argumenten. Doordat functies niet van een globale toestand afhangen is hun gedrag geheel voorspelbaar. Sommige berekeningen die essentieel van een globale toestand afhangen, zoals invoer/uitvoer, zijn in functionele talen echter moeilijker uitdrukbaar. Voorbeelden zijn Lisp, Standard ML, en Haskell.

Logische talen zijn gebaseerd op *axioma's* en *inferentieregels*. Uitvoering van een logisch programma komt neer op het zoeken naar een bewijs voor een door de gebruiker geformuleerde stelling. De belangrijkste logische programmeertaal is Prolog.

Scripttalen zijn gericht op het snel en flexibel oplossen van allerlei problemen op het gebied van tekstverwerking, gegevens- en systeembeheer, en gebruikersinterfaces. Deze talen zijn meestal ongetypeerd en programma's worden door een interpretator uitgevoerd; ze bieden uitgebreide mogelijkheden voor tekstbewerking (stringmanipulatie, herkennen van reguliere expressies), associatieve tabellen, dynamische arrays, en geven toegang tot de volledige functionaliteit van het onderliggende besturingssysteem. Voorbeelden zijn Rexx, Perl, Tcl en Python.

1.5 Toepassingsgebieden

Programma's kunnen ook ingedeeld worden naar het toepassingsgebied waarbinnen ze een probleem oplossen. Het toepassingsgebied is in grote mate bepalend voor de eisen die aan een programmeertaal gesteld worden.

In *wetenschappelijke toepassingen* spelen wiskundige concepten zoals reële getallen, vectoren, matrices, numerieke benaderingen, differentiaalvergelijkingen en dergelijke een belangrijke rol. Deze toepassingen zijn vaak zeer rekenintensief, bewerken zeer grote gegegevensbestanden, en lenen zich soms voor uitvoering op een parallelle computer.

Bedrijfstoepassingen worden gekenmerkt door het gebruik van grote databestanden, en het ondersteunen van administratieve processen (*work flow*). Deze toepassingen zijn minder rekenintensief, manipuleren vaak zeer grote databestanden, moeten in een gedistribueerde omgeving functioneren, en stellen hoge eisen t.a.v. betrouwbaarheid.

In *systeemprogrammatuur* spelen laag-niveau besturing en beveiliging van apparatuur een rol. Grote nadruk ligt op efficiëntie, toegang tot de apparatuur op het niveau van individuele bits, beveiliging, en parallelle uitvoering van taken.

In toepassingen voor het *Web/Internet* staan centraal zowel de interactie met de gebruiker (via een Web browser) als het consistente beheer van databases die door gebruikers, tegelijkertijd, geraadpleegd en gewijzigd kunnen worden. Hierbij spelen parallellisme, beveiliging en ondersteuning voor velerlei scripttalen een rol.

In *Mens/machine interfaces* spelen het interactiemodel, de wijze van grafische presentatie van informatie, en ergonomie een grote rol. Andere kenmerken zijn de besturing van niet-standaard apparatuur (grafische kaarten, virtual reality displays), gebruik van speciale bibliotheken, en van speciale scripttalen.

In allerlei *gespecialiseerde domeinen* is men overgegaan tot het ontwikkelen van *domeinspecifieke talen*. Een bekend voorbeeld zijn de vierde-generatie talen (4GL) die voor administratieve toepassingen gebruikt worden. Daarnaast bestaan er domeinspecifieke talen op gebieden als spoorwegbeveiliging, productieplanning, financiële dienstverlening, en voorraadbeheer. Kenmerk van domeinspecifieke talen is dat gespecialiseerde hulpmiddelen (tools) nodig zijn voor het ontwikkelen en uitvoeren van programma's.

1.6 Uitvoeringswijze

Programmeertalen kunnen op verschillende wijze uitgevoerd worden. De meeste traditionele talen worden uitgevoerd door middel van *compilatie*: het oorspronkelijke programma wordt via een aantal tussenstappen omgezet in uitvoerbare machine-instructies. Voordelen zijn het vroegtijdig opsporen van fouten en efficiënte uitvoering.

Een alternatief voor compilatie is *interpretatie*: hierbij wordt het programma direkt uitgevoerd zonder het eerste naar machine-instructies te vertalen. Interpretators kunnen eenvoudiger naar verschillende platforms overgedragen worden en kunnen allerlei faciliteiten ondersteunen die met een compiler moeilijker te realiseren zijn. Dit verklaart dat veel van de *script* talen zoals Awk en Tcl op interpretatie gebaseerd zijn.

2 Concepten van programmeertalen

Wie eenmaal een eerste programmeertaal kent, heeft meestal minder moeite met het leren van een tweede taal. Blijkbaar hebben twee verschillende talen vaak een aantal onderliggende *concepten* gemeen. In deze paragraaf laten we de belangrijkste van dergelijke concepten de revue passeren.

Bij de bespreking van de diverse concepten staat in deze paragraaf de programmeur (of eigenlijk het team van programmeurs) centraal: hoe helpen de diverse taalconcepten de progammeur bij het schrijven van *kwaliteitssoftware*, dat wil zeggen programmatuur die de volgende eigenschappen bezit:

- *Correctheid*: doet de software inderdaad wat er gedaan moet worden, en hoe is dat te controleren?
- *Robuustheid*: hoe gaan de programma's om met uitzonderingsgevallen zoals bijvoorbeeld afwijkende invoerdata?
- *Uitbreidbaarheid*: hoe makkelijk is het om het systeem aan te passen aan nieuwe eisen of wensen?
- *Herbruikbaarheid*: hoe eenvoudig is het om (onderdelen van) de geschreven programmatuur in andere situaties her te gebruiken?

Pas in paragraaf 3 verplaatsen we het perspectief naar de computer: hoe kan een compiler of interpretator de voor de mens zo prettige concepten efficient executeren.

2.1 Data

Alle programma's manipuleren data. In deze paragraaf bekijken we hoe datawaarden weergegeven kunnen worden met behulp van *expressies*. Bovendien bestuderen we hoe verschillende soorten waarden onderverdeeld kunnen worden in *types*. Ten slotte zullen we zien hoe meer gecompliceerde types en waarden samengesteld kunnen worden uit eenvoudigere.

2.1.1 Expressies

Alle programmeertalen bevatten *expressies*, waarmee het mogelijk is berekeningen op te schrijven die een *waarde* als resultaat hebben. De meest eenvoudige expressies zijn *constante* waarden, zoals:

- gehele getallen, zoals $\ldots, -2, -1, 0, 1, 2, \ldots$;
- *floating point* getallen, zoals `3.1415` en `6.626E-34`, notatie voor $6.626 * 10^{-34}$;
- losse karakters, zoals `'a'`, `'b'`, `'!'`;
- strings, d.w.z. rijtjes karakters zoals `"A.P. Beerta"` en `"DC-10"`;
- de Boolean waarden `true` en `false`.

Dergelijke eenvoudige waarden kunnen met behulp van *operatoren* gebruikt worden om berekeningen uit te voeren, zoals

```
3 * 3 + 4 * 4
```

Hierin komen de constanten `3` en `4` voor, en de operatoren `*` en `+`. De waarde van deze expressie kan uitgerekend worden, en bedraagt `25`. Veelvoorkomende operatoren zijn de *aritmetische* (−, +, `*`, `/`, ...), de *logische* (`and`, `or`, `not`, ...), en de *relationele* (>, <, =, ...).

Expressies kunnen ook *variabelen* bevatten, zoals:

```
x * x + y * y
```

In dat geval is de waarde van een expressie pas uit te rekenen als de waarde van de variabelen (in dit geval `x` en `y`) ook bekend is.

2.1.2 Types

De waarden die gemanipuleerd worden in programma's kunnen ingedeeld worden in verschillende *types*. Een type is allereerst een naam voor een *verzameling waarden*. Daarnaast kunnen de diverse waarden uit een type op dezelfde manier intern gerepresenteerd worden.

De meest eenvoudige types zijn we al tegengekomen als constante waarden in expressies (pag. C1/11). De mogelijke constanten konden zijn: gehele getallen (integers); floating point getallen; karakters; strings; en Booleans.

Andere eenvoudige types zijn die waarvan we van te voren alle mogelijke waardes op kunnen sommen, de zogenaamde *enumeratie types*. Voorbeelden zijn de zeven dagen van de week, of de twaalf maanden van het jaar. Het bijbehorende enumeratie type bestaat dan uit de verzameling met symbolen voor de mogelijke waarden, zoals bijv.:

```
type dag = (zo, ma, di, wo, do, vr, za)
```

Voor meer gecompliceerde datastructuren zijn er een aantal manieren om nieuwe types samen te stellen uit andere types. Hiervan bespreken we in eerste instantie arrays (pag. C1/12) en records (pag. C1/13). Later zullen we dan nog referenties (pag. C1/19), object-georiënteerde klassen (pag. C1/30), lijsten (pag. C1/35), en functie-types (pag. C1/37) tegen komen.

2.1.3 Arrays

Een *array* wordt gebruikt om een rij (Engels: array) waarden van hetzelfde type in op te slaan. Een voorbeeld van een array waarde die uit 5 elementen bestaat is

```
{ 400, 420, 600, 620, -50 }
```

Het array type dat hierbij hoort ziet er als volgt uit:

```
array [ 1..5 ] of Integer
```

Hier geeft de `1..5` aan dat het array vijf elementen bevat, en dat bij het nummeren van de elementen met het cijfer `1` begonnen wordt.

Als `R` een variabele van een array type is, kan het *n*-e element verkregen worden via de notatie `R`[*n*]. Als `R` bijvoorbeeld bovenstaande waarde heeft, dan heeft de expressie `R[3]` als waarde `600`, het derde getal uit de rij.

2.1.4 Records

Een *record* is een combinatie (cartesisch product) van types, die de *velden* van het record worden genoemd. Het record bevat een naam voor elk veld. Een voorbeeld van een record waarde (in de notatie van Ada) is:

```
( naam   =>  "B. Droogstoppel",
  straat =>  "Lauriergracht",
  nummer =>  37,
  woonplaats =>  "Amsterdam" )
```

Het bijbehorende *record type* bevat vier velden, en ziet er als volgt uit:

```
type Adres =
  record
    naam: String;
    straat: String;
    nummer: Integer;
    woonplaats: String;
  end;
```

Gegeven een record variabele, kan de waarde van een van de velden via de veldnaam worden opgevraagd. Dus, als `P` een variabele van bovenstaand record type is, dan kan via de expressie

```
P . woonplaats
```

de woonplaats van `P` worden verkregen.

2.1.5 Declaratie en controle van types

Hoe kunnen we weten of een variabele (of procedure) die in een programma voorkomt van een bepaald type is en op de juiste wijze gebruikt wordt?

Bij *expliciete typering* geven we voor elke variabele (of procedure) zelf aan van welk type die moet zijn, via een expliciete *declaratie*. Bij *impliciete typering* wordt het type van een variabele (of procedure) niet expliciet gedeclareerd, maar het kan eventueel wel uit het gebruik afgeleid worden. Uit de expressie `3 + x` kunnen we, bijvoorbeeld, afleiden dat `x` een integer is.

Typecontrole houdt in dat vastgesteld wordt dat constanten, variabelen, functies en procedures in een programma qua typering correct gebruikt worden. Operatoren mogen alleen toegepast worden op operanden van het juiste type en functies en procedures mogen alleen aangeroepen worden met het juiste aantal parameters van het juiste type.

Wanneer wordt de typecontrole uitgevoerd? Bij *statische typecontrole* worden de typecontrole tijdens het compileren van het programma uitgevoerd. In het geval van expliciete typering wordt gecontroleerd of het gedeclareerde type van elke variabele compatibel is met al zijn gebruiken. In het geval van impliciete typering, wordt geprobeerd om het type van elke variabele of procedure af te leiden uit alle plaatsen in het programma waar deze gebruikt wordt. Als dit niet mogelijk blijkt dan is het programma niet type-correct: een variabele of procedure wordt dan op verschillende plaatsen in het programma met verschillende, onderlinge niet-compatibele, types gebruikt.

Bij *dynamisch typecontrole* wordt pas tijdens het uitvoeren van het programma, aan de hand van het gebruik van waarden, vastgesteld of het gebruik typecorrect is.

Expliciete, respectievelijk, impliciete typering is in verschillende talen op verschillende manieren gecombineerd met statische, respectievelijk, dynamische typecontrole:

	Expliciete typering	Impliciete typering
Statische controle	Pascal, C, C++, Java	ML
Dynamische controle		Smalltalk

Impliciete typering levert de programmeur minder schrijfwerk op. Dynamische typecontrole is in het algemeen wat flexibeler.

De combinatie van expliciete typering met statische typecontrole heeft een aantal belangrijke voordelen:

- *Betrouwbaarheid*, doordat de compiler van te voren kan vaststellen welke variabelen niet in overeenstemming met hun declaraties wordt gebruikt;

- *Leesbaarheid*, doordat de variabele declaraties expliciet aangeven met wat voor soort variabele we te maken hebben;

- *Efficiëntie* omdat de variabele declaraties door de compiler gebruikt kunnen worden om betere code te genereren.

Een programmeertaal kan zich in meerdere of mindere mate lenen voor statische typecontrole. Daarom wordt ook wel een onderscheid gemaakt tussen *sterk* getypeerde talen, waarin type informatie wordt gebruikt om *alle* aan typering gerelateerde fouten voorafgaand aan executie te elimineren, en *zwak* getypeerde talen, waarin bepaalde fouten voorkomen kunnen worden, maar niet alle.

2.2 Het imperatieve model

In imperatieve programma's spelen veranderingen van de waardes van variabelen een centrale rol. De combinatie van alle programma-variabelen vormt een *toestand*, die kan veranderen doordat een variabele van waarde verandert. Imperatieve programma's bestaan uit een reeks *opdrachten* die een initiële toestand stap voor stap omzetten in de gewenste eindtoestand. Bekende imperatieve talen zijn COBOL, C, FORTRAN, of Pascal.

Een kenmerkend voorbeeld, geschreven in de taal Pascal, is te vinden in Figuur 2.1. Hier wordt de grootste gemene deler van twee variabelen `x` en `y` berekend. Door herhaald `x` en `y` te verlagen, wordt uiteindelijk een toestand bereikt waarin `x` het gewenste resultaat bevat.

2.2.1 Opdrachten

De belangrijkste *opdrachten* (Engels: *statements*) zijn:

- De *toekenning* (*assignment*), om aan een variabele de waarde van een expressie toe te kennen.

- De conditionele *keuze* (*if-then-else*), waarbij een opdracht wordt uitgevoerd afhankelijk van een conditie.

```
program Euclides(input,output);
var x, y: integer;
begin
  readln(x, y);
  while x ≠ y do
  begin
    if x > y
    then x := x - y
    else y := y - x
  end;
  writeln('ggd is: ', x);
end.
```

Figuur 2.1: Programma in imperatieve stijl voor het berekenen van de grootste gemene deler. Voorbeeld: $\text{ggd}(6,8) = 2$.

- De *gevalsonderscheiding* (*case-statement*) om voor verschillende waarden van een bepaalde variabele verschillende opdrachten uit te voeren.
- De *for-loop*, om een variabele in een lus een serie waarden te laten doorlopen.
- De *while-loop*, om een serie opdrachten te herhalen zolang een conditie waar is.
- Het *compound-statement*, het groeperen van een serie opdrachten, in Pascal via begin-end paren, in C via accolades aangegeven.

In sommige talen is bovendien een *sprongopdracht* (goto) toegestaan, maar het gebruik daarvan is (1) niet nodig; en (2) af te raden omdat het leidt tot moeilijk begrijpbare programma's.

2.2.2 Het imperatieve geheugenmodel

Imperatief programmeren is gebaseerd op het volgende *geheugenmodel*:

- Een *geheugen* bestaat uit een serie *cellen* (ook: *lokaties*).
- Elke cel kan een *waarde* bevatten.

- Elke cel heeft een *adres*.

Bij elke variabele hoort een cel. Een toekenning aan een variabele heeft als effect dat de in de bij de variabele horende cel van waarde verandert.

2.2.3 Procedures

Het belangrijkste structureringsmechanisme in imperatieve talen is de *procedure*. Door middel van een proceduredeclaratie wordt aan een bepaalde serie opdrachten een naam gegeven. Door deze naam *aan te roepen* wordt de serie opdrachten daadwerkelijk uitgevoerd. Is de serie opdrachten uitgevoerd, dan gaat de executie van het programma verder in het statement na de aanroep.

Als voorbeeld is in Figuur 2.2 een procedure `ToCelcius` gedefinieerd die een gegeven temperatuur in graden Fahrenheit omzet in graden Celcius, en vervolgens beide temperaturen afdrukt. Deze procedure wordt vervolgens voor alle waarden tussen 50 en 100 aangeroepen in het *hoofdprogramma*, de serie opdrachten waarmee de executie van het programma begint.

2.2.4 Scope

De *scope* van een variabele bestaat uit dat gedeelte van het programma waar die variabele gebruikt mag worden. Het is verstandig de scope van een variabele zo klein mogelijk te houden, zodat een variabele uitsluitend daar zichtbaar is waar hij nodig is.

Veel talen kennen daarom een *blokstructuur*, waarbij een variabele uitsluitend zichtbaar is in het *blok* waarin deze is gedeclareerd. De blokstructuur kan *plat* zijn, zoals bij COBOL waar elk programma uit precies één blok bestaat, of *genest*, waarbij binnen een blok nieuwe kleinere blokken kunnen voorkomen, bijvoorbeeld voor elke procedure. Dit laatste is het geval voor Pascal: in Figuur 2.2 bestaat het buitenste blok (met de declaratie voor `i`) uit het gehele programma. Daarbinnen zit het blok voor de procedure `ToCelcius`, met de declaratie van `celcius`.

Variabelen gedeclareerd in een binnengelegen blok zijn *lokale* variabelen in dat blok; variabelen in het buitenste blok worden *globaal* genoemd.

```
program FahrenheitCelcius(input,output);
var i : integer;

  procedure ToCelcius(fahr: integer);
  var celcius : integer;
  begin
     celcius := 5 * (fahr - 32) / 9;
     writeln(fahr, ' ', celcius);
  end; { ToCelcius }

begin { hoofdprogramma }
   for i:=50 to 100 do
   begin
      ToCelcius(i);
   end;
end.
```

Figuur 2.2: Pascal programma met procedure om graden Fahrenheit in Celcius om te zetten.

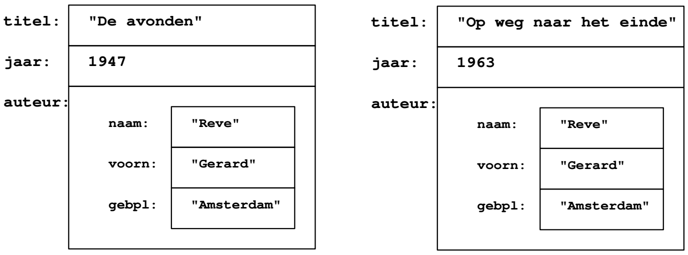
2.2.5 Parameters

Een procedure (zoals bijvoorbeeld die uit Figuur 2.2) kan *geparameteriseerd* zijn. De *formele* parameter van de procedure (in dit geval `fahr`), krijgt pas een waarde bij de *aanroep*. Deze bij de aanroep vermelde waarde wordt de *actuele* parameter genoemd (in dit geval `i`).

De belangrijkste twee mechanismes van parameteroverdracht zijn *call by value* en *call by reference*. In *call by value* (zoals in Figuur 2.2) wordt de *waarde* van de actuele parameter gekopieerd naar de formele parameter, en hebben wijzigingen op de formele paremeter (`fahr`) *geen* effect op de actuele parameter (`i`). Bij call by reference is dit wel het geval: actuele en formele parameter verwijzen dan naar dezelfde geheugenlokatie (pag. C1/16).

2.2.6 De return-opdracht

Een *functie* berekent, gegeven een aantal parameters, de waarde van een expressie (pag. C1/11). In de meeste imperatieve talen



Figuur 2.3: Twee geneste records, elk met een kopie van dezelfde auteursinformatie.

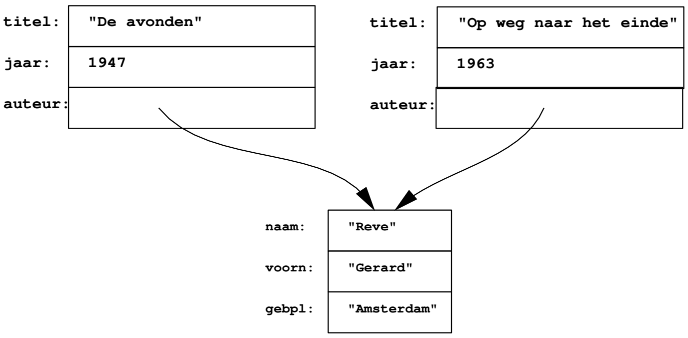
kunnen programmeurs zelf functies definiëren. Dit gebeurt dan via een functie-declaratie, die vaak erg lijkt op een procedure-declaratie. Net als procedures bestaan functies uit een rij opdrachten. Een speciale `return`-opdracht kan dan worden gebruikt om de waarde van de functie te retourneren.

Ten gevolge van de gelijkenis met procedures, is het mogelijk om in de statements van een functie globale variabelen van waarde te veranderen. Deze wijziging is dan niet strikt noodzakelijk om de return-waarde van de functie te berekenen. Om die reden wordt zo'n wijziging een *zij-* of *neven-effect* genoemd. Zij-effecten kunnen zorgen voor onverwachte wijzigingen, en dienen over het algemeen vermeden te worden.

2.2.7 Referentie-types

Een natuurlijke uitbreiding op het imperatieve geheugenmodel (pag. C1/16) is toe te staan dat geheugen-*adressen* ook als *waarde* worden gebruikt. Een dergelijke waarde wordt een *referentie* of *pointer* naar een andere waarde genoemd.

Door referenties te gebruiken is het mogelijk dat verschillende records informatie met elkaar delen. Nemen we als voorbeeld een boek-record, met als velden de titel, het jaar van uitgifte, en de auteur. De auteursinformatie is weer een record met een naam, voornaam en geboorteplaats. Zonder verwijzingen wordt de auteursinformatie voor elk boek gedupliceerd, zoals Figuur 2.3 laat zien. Met verwijzingen is zogenaamde *sharing* wel mogelijk, waarbij beide boek-records een verwijzing bevatten naar het gemeenschappelijke auteurs-record, zoals te zien in



Figuur 2.4: Twee records met een gemeenschappelijk auteurs-record.

```
type  Auteur = record
                 naam:  String;
                 voorn: String;
                 gebpl: String;
               end;
type  AuteurWijzer = ^ Auteur;
type  Boek = record
                 titel:  String;
                 jaar:   Integer;
                 auteur: AuteurWijzer;
             end
```

Figuur 2.5: Type definitie voor de boekinformatie uit Figuur 2.4
.

Figuur 2.4. De bijbehorende record type-definities (in de notatie van Pascal) zijn gegeven in Figuur 2.5. Hierin is ^ de constructor voor referentie-types.

2.2.8 Dynamische geheugenallocatie

Een pointer-variabele die nergens naar wijst heeft de waarde *nil* (ook: *null*). Om een pointer-variabele naar een cel te laten wijzen, moet een nieuwe geheugencel *gealloceerd* (gereserveerd) worden. Dit gaat in Pascal via het `new` commando, in C via de `malloc` operatie.

Is de cel waar een pointer-variabele naar wijst niet meer nodig, dan moet de gealloceerde geheugenruimte weer expliciet vrijgegeven worden (via een *deallocatie* of *free* operatie).

Dit is echter niet in alle talen nodig. Met name object-georiënteerde en functionele talen ondersteunen *garbage collection*: wordt tijdens executie ontdekt dat er geen referenties meer bestaan naar een bepaalde cel, dan wordt deze cel automatisch vrijgegeven. Garbage collection kan nadelige gevolgen op de uitvoeringssnelheid van een programma hebben. Maar een zeer belangrijk voordeel is dat het de programmeur ontslaat van de plicht geheugen vrij te geven. Bovendien maakt het de programmeur *onmogelijk* om geheugen vrij te geven waar nog *wel* naar verwezen wordt.

2.2.9 Excepties

Elke procedure werkt onder bepaalde aannames: het bestand dat gelezen wordt mag niet halverwege het leesproces eindigen, van bepaalde invoerwaardes wordt aangenomen dat ze tussen bepaalde grenzen liggen, enz. (de *precondities* van de procedure). Wanneer tijdens executie wordt ontdekt dat de gemaakte aannames toch niet kloppen, treedt een uitzonderingssituatie of *exceptie* in.

Sommige talen ondersteunen expliciete *exception handling*. De programmeur kan excepties een naam geven, en kan een exceptie genererern (*throw* / *raise* an exception). Bovendien is het mogelijk om te “proberen” een procedure uit te voeren: wordt tijdens de executie daarvan ergens een exceptie gegenereerd, dan kan die expliciet worden afgevangen, waarna een foutmelding kan worden gegeven of een hernieuwde poging kan worden ondernomen (met bijvoorbeeld iets afwijkende parameters).

2.2.10 Modules

Voor het schrijven van grote software-systemen is het van belang de potentieel zeer omvangrijke code op te splitsen in kleinere groepen: *modules*. In imperatieve talen bestaan modules uit een serie declaraties van procedures, variabelen, types, enz.

Een goede module levert een aantal met elkaar samenhangende "diensten" (*services*): deze vormen de *interface* van de module (vaak de API genoemd: *Application Programming Interface*).

Hoe deze diensten worden gerealiseerd is voor het *gebruik* van de module niet van belang: daarom stelt een goed modulariseringsmechanisme de programmeur in staat die procedures, types, variabelen, enzovoorts, die de diensten *implementeren*, te verbergen. Het voordeel van dergelijke *information hiding* is dat slechts de implementerende module op de hoogte is van de details van de implementatie, zoals de gebruikte data-representatie en algorithmes. Is hierin een wijziging nodig waarbij het interface hetzelfde blijft, dan kan deze wijziging probleemloos worden uitgevoerd.

2.3 Belangrijke imperatieve talen

2.3.1 FORTRAN

FORTRAN (uit 1955) was een van de eerste hogere programmeertalen, en zeker de eerste die in brede kring werd gebruikt. FORTRAN, van *The IBM Mathematical FORmula TRANslating System*, wordt nog steeds bijzonder veel gebruikt voor toepassingen op het gebied van numerieke wiskunde, zoals matrixberekeningen en het oplossen van differentiaalvergelijkingen. Deze toepassingen worden niet alleen mogelijk gemaakt door de programmeerconcepten in FORTRAN, maar met name ook door de in de loop van decennia ontwikkelde bibliotheken op het gebied van numerieke wiskunde die in FORTRAN beschikbaar zijn.

FORTRAN is een zwakgetypeerde imperatieve taal, met procedures en diverse soorten statements, en met speciale voorzieningen om het rekenen met arrays te vergemakklijken. FOR-

TRAN 90 kent een aantal constructies die niet in het eerdere FORTRAN 77 zaten: extra operatoren om met matrices te rekenen, records, dynamisch geheugengebruik, en een modulariseringsmechanisme met PRIVATE en PUBLIC procedures.

2.3.2 COBOL

De COmmon Business Oriented Language COBOL stamt uit 1959. Hoewel (of waarschijnlijk juist omdat) COBOL een oude taal is, is COBOL nog steeds met afstand de meest gebruikte programmeertaal ter wereld. De taal wordt vooral gebruikt voor omvangrijke administratieve dataverwerking.

De huidige COBOL standaard is van 1985. COBOL is een tamelijk eenvoudige imperatieve taal, zwak getypeerd, met een platte blokstructuur per programma, en parameterloze procedures (“secties”), en zonder ingebouwde ondersteuning voor functies, referentie-types en excepties. Het modulariseringsmechanisme van COBOL maakt het mogelijke losse programma’s aan te roepen, gebruik makend van parameters. Bij het ontwerp van COBOL is er naar gestreefd de taal zoveel mogelijk op het Engels te laten lijken.

Op dit moment wordt gewerkt aan een nieuwe COBOL standaard. De belangrijkste uitbreiding hierin is de ondersteuning van object-oriëntatie (pag. C1/24).

2.3.3 C

De programmeertaal C behoort tot de familie van talen die zijn voortgekomen uit de *Algorithmic Language* ALGOL-60 (en later ALGOL-68). Deze talen worden gekenmerkt door een geneste blokstructuur, statische typering, en de mogelijkheid van dynamische geheugenallocatie. Naast C behoren Pascal (veel gebruikt in het onderwijs), Modula-2 (met een krachtig modulariseringsmechanisme) en Ada (ontwikkeld door het Amerikaanse Ministerie van Defensie) tot de ALGOL-achtige talen.

De oorsprong van C ligt bij het UNIX besturingssysteem. De eerste versie van UNIX (voor de DEC PDP-11) is geschreven in C, en C was speciaal ontworpen voor de bouw daarvan. Inmiddels wordt C gebruikt voor allerhande toepassingen, van fabrieksbesturing tot adminstratie-afhandeling.

Om het schrijven van systeemsoftware mogelijk te maken, heeft C enkele operatoren die op bit– en byte-niveau werken. Bovendien heeft C een bijzonder liberale vorm van pointers waarop zelfs berekeningen mogelijk zijn (pointers, array-indices, integers en karakters zijn makkelijk in elkaar om te zetten). De flexibiliteit van de pointers in C heeft ook zijn keerzijde: veel fouten in C programma's zijn te herleiden tot pointer-problemen.

Als uitbreiding op C bestaat het object-georiënteerde C++: deze taal wordt besproken in Sectie 2.5.3.

2.4 Object-oriëntatie

Een object-georiënteerd programma bestaat uit een reeks klassedefinities. Van elk van deze klassen kan een willekeurig aantal instanties gemaakt worden. Het programma draait doordat de bijbehorende objecten elkaar "berichten sturen", d.w.z. elkaars methoden aanroepen.

2.4.1 Klasse

Een klasse bestaat uit een combinatie van variabelen en procedures die de variabelen kunnen inspecteren of wijzigen. Procedures van een klasse worden *methoden* genoemd, de variabelen worden *attributen*, *velden* of *instance variables* genoemd. De methoden en instance variabelen bij elkaar worden de *features* (of *members* in C++ terminologie) van een klasse genoemd.

Een voorbeeld van een klassedefinitie in Java is te zien in Figuur 2.6. Hierin wordt een klasse `Cirkel` gedefinieerd, met twee instance variabelen: `PI` (die een constante waarde heeft), en `r`, die de straal van de cirkel bevat, elk van type `double`, d.w.z. een 64-bits floating point getal.

De klasse definieert twee methoden: een om de `omtrek` van een cirkel te berekenen, en een om het `oppervlak` uit te rekenen.

2.4.2 Object

Een *object* is een *instantiatie* van een klasse. Gegeven een klasse-definitie, kunnen diverse objecten in die klasse *gecreeerd* worden. Voor het cirkel voorbeeld (Figuur 2.6) betekent dit dat

```
public class Cirkel {
  public double r;
  public double PI = 3.1415;

  public double omtrek () {
    return 2 * PI * r;
  }

  public double oppervlak () {
    return PI * r * r;
  }
}
```

Figuur 2.6: Een Java klasse voor cirkel-operaties.

er diverse concrete cirkels gemaakt worden, elk met een eigen straal.

In Java worden objecten gecreeerd via een expressie (pag. C1/11), gebruik makend van de `new` operator. Dus, via

```
c = new Cirkel();
```

wordt een nieuwe cirkel gecreeerd en aan de variabele `c` toegekend. `new` wordt gevolgd door een *constructor-functie*: de naam van de klasse met tussen haakjes eventuele argumenten die voor creatie nodig zijn.

2.4.3 Access

Is een object van een bepaalde klasse gecreeerd, dan kan de waarde van de instance variabelen worden opgevraagd en gewijzigd, en kunnen de methoden van dat object worden aangeroepen. In het Cirkel voorbeeld, geeft de expressie

```
c.r
```

de straal van de cirkel `c`. De straal van `c` kan veranderd worden door een toekenning (pag. C1/15):

```
c.r = 3.0;
```

Door middel van de expressie

```
c.omtrek()
```

wordt de methode `omtrek` van de `c` aangeroepen (die als antwoord $2*3.1415*3.0 = 18.8490$ zal retourneren).

2.4.4 Encapsulatie

Een van de belangrijkste object-georiënteerde concepten is *encapsulatie* (ook *data hiding* genoemd): het verbergen van de instance variabelen voor andere klassen, en het opsplitsen van de methoden in *geëxporteerde* en *verborgen* methoden. De verborgen features zijn alleen binnen de klasse te gebruiken.

Een dergelijke opsplitsing heeft drie belangrijke voordelen:

- De klasse krijgt een expliciet *interface*, namelijk de *geëxporteerde* methoden en variabelen.
- De details van de implementatie (wordt een array of een string gebruikt, hoe heet een bepaalde variabele?) zijn verborgen, en kunnen zo nodig eenvoudig worden aangepast.
- De *consistentie* van de interne toestand van een klasse kan beter gegarandeerd worden. Verborgen instance variabelen kunnen niet van buiten gewijzigd worden. Het is daarom voldoende te controleren dat de geëxporteerde features de interne consistentie bewaren.

In Java zijn variabelen en methoden verborgen, tenzij ze voorzien zijn van de *visibility modifier* `public`. Figuur 2.7 toont een versie van de `Cirkel` klasse waarbij de variabelen `r` en `PI` lokaal zijn gemaakt. De klasse heeft een extra constructor (pag. C1/24) functie `Cirkel`, die als argument de gewenste straal heeft (die nu immers niet meer expliciet gewijzigd kan worden).

2.4.5 Overerving

Echte software systemen zullen bestaan uit zeer veel verschillende klassen. Sommige daarvan zullen veel op elkaar lijken, en zijn feitelijk varianten van elkaar. Door middel van *overerving* (Engels: *inheritance*) is het mogelijk een klasse (de *subklasse*)

```
public class Cirkel {
  double r;
  double PI = 3.1415;

  public Cirkel(s) {
    r = s;
  }

  public double omtrek () {
    return 2 * PI * r;
  }

  public double oppervlak () {
    return PI * r * r;
  }
}
```

Figuur 2.7: Cirkel-klasse met verborgen (niet `public`) instance variabelen.

```
public class MaxCirkel extends Cirkel {
  double MAXR = 100.0;

  public MaxCirkel(double s) {
    if (s <= MAXR) {
      r = s;
    } else {
      r = MAXR;
    }
  }
}
```

Figuur 2.8: Klasse `MaxCirkel` (cirkels met beperkte straal) die via overerving de methodes van `Cirkel` hergebruikt.

de features van een andere klasse (de *superklasse*) te laten erven. De subklasse wordt dan gedefinieerd als *uitbreiding* van de superklasse.

Een voorbeeld van overerving is de klasse `MaxCirkel` in Figuur 2.8 van cirkels met een beperkte straal. Deze klasse erft alle features van de klasse `Cirkel`, en heeft een eigen extra instance variabele `MAXR` die de maximum straal bevat. De constructor-functie `MaxCirkel` zorgt ervoor dat de waarde van de instance variabele `r` (geërfd van `Cirkel`) nooit meer wordt dan de waarde van `MAXR`.

2.4.6 Method overriding

Bij overerving is de subklasse een *uitbreiding* van de superklasse. Daarnaast is het ook mogelijk overerving te gebruiken om de subklasse een *verfijning* van de superklasse te laten zijn. Dit is mogelijk doordat de subklasse voor een bepaalde geërfde methode een eigen implementatie kan bevatten. Deze eigen methode implementatie vervangt dan de methode van de superklasse (method *overriding*).

2.4.7 Abstracte klassen

Vaak zal een superklasse meerdere subklasses onder zich hebben hangen, de diverse varianten van de superklasse. In het voorbeeld van de cirkel, is het goed denkbaar dat er ook klassen bestaan voor driehoeken, vierkanten, ellipsen, enz. Al deze klassen hebben twee eigenschappen gemeenschappelijk, namelijk dat van elk van deze figuren de omtrek en oppervlakte uitgerekend kan worden.

In zo'n geval is het mogelijk een superklasse `Figuur` te definiëren, waarin vermeld wordt dat alle subklassen van `Figuur` ten minste de methodes `omtrek` en `oppervlakte` moeten hebben. De klasse `Figuur` kan echter geen implementatie voor deze methoden geven. Om die reden heten deze methoden *abstract* (of *deferred*, uitgesteld). De feitelijke implementatie wordt gegeven door de subklasses, die wel voldoende informatie hebben om de abstracte methodes te implementeren.

De abstracte klasse `Figuur` is te zien in Figuur 2.9. Als uitbreiding daarop toont Figuur 2.10 de klasse van vierkanten met een

```
public abstract class Figuur {
  public abstract double omtrek();
  public abstract double oppervlakte();
}
```

Figuur 2.9: *Abstracte* klasse van figuren in het algemeen, met een omtrek en een opervlakte.

```
public class Vierkant extends Figuur {
  double len;
  public Vierkant(double l) {
    len = l;
  }
  public double omtrek() {
    return 4 * len;
  }
  public double oppervlakte() {
    return len * len;
  }
}
```

Figuur 2.10: Klasse van vierkanten, met oppervlakte en omtrek.

eigen manier om omtrek en oppervlakte uit te rekenen.

2.4.8 Late binding

Door het gebruik van inheritance is het mogelijk dat voor één methode diverse implementaties bestaan. Een voorbeeld is de methode `omtrek` uit de klasse `Figuur` (Fig. 2.9), die verschillende definities heeft voor de klassen `Cirkel` en `Vierkant` (Fig. 2.10).

Stel nu dat we in een for-loop van een array van figuren de gemiddelde omtrek willen berekenen. De Java code hier voor is te zien in Figuur 2.11. Welke `omtrek` moeten we dan gebruiken?

Het antwoord is dat deze keuze niet van te voren (*compile time*) gemaakt kan worden. Daarom voorzien object-georiënteerde talen in *late binding*: pas tijdens executie (run time) wordt de implementatie bij een gegeven methode-naam gezocht. Dus, pas

```
tot = 0;
for (i=0; i < figs.length; i++) {
  tot = tot + figs[i].omtrek();
}
gem = tot / figs.length;
```

Figuur 2.11: Berekening van de gemiddelde omtrek van een array van figuren. Late binding bepaalt welke `omtrek` methode wordt gebruikt.

tijdens executie is duidelijk welk van de figuren feitelijk vierkanten of cirkels zijn, en pas dan wordt beslist op welke wijze de omtrek berekend wordt.

In de meeste object-georiënteerde talen (Java, Eiffel, Smalltalk) is late binding de standaardoptie; in C++ moet dit expliciet aangegeven worden (via het keyword `virtual`).

2.4.9 Meervoudige overerving

De meest gangbare vorm van overerving is *enkelvoudig* (*single inheritance*): een klasse erft van maximaal één andere klasse. In sommige gevallen is het zinvol om van meerdere klassen features te erven (*multiple inheritance*). Een voorbeeld is de klasse "piano", die we kunnen zien als zowel "snaarinstrument" als "toetseninstrument".

Talen als C++ en Eiffel ondersteunen meervoudige overerving. Hiertoe hebben zij mechanismes om problemen met dubbel voorkomende feature-namen of via verschillende routes geërfde features op te lossen. Java heeft een beperkte vorm van meervoudige overerving: het is wel mogelijk meerdere *interfaces* te erven, maar onmogelijk om meerdere *implementaties* te erven.

2.4.10 Klassen als types

In object-georiënteerde talen spelen klassen de rol van types (pag. C1/13). Een object-georiënteerd type is dan niet alleen een karakterisering van een verzameling waarden, maar ook van de toegestane *operaties* (de methoden van de klasse) op die waarden. Door encapsulatie (pag. C1/26) wordt de representatie van de waarden verborgen, en om die reden worden op klassen ge-

```
Auteur a;
Boek b1, b2;

a = new Auteur("Reve",
  "Gerard", "Amsterdam");

b1 = new Boek("De avonden", 1947, a);

b2 = new Boek("Op weg naar het einde",
   1963, a);
```

Figuur 2.12: Twee objecten met een gemeenschappelijk auteurs-object.

baseerde types dan ook *abstracte data types* genoemd.

Sommige object-georiënteerde talen, zoals Smalltalk, zijn dynamisch getypeerd. Pas tijdens executie wordt vastgesteld of een bepaald object een gegeven methode wel ondersteunt. Dit leidt tot een zeer liberale vorm van *polymorfisme*: argumenten van methodes mogen van elke denkbare klasse zijn, zolang ze de methodes die op de argumenten worden losgelaten maar ondersteunen.

Meer recente object-georiënteerde talen, zoals Eiffel en Java, zijn statisch getypeerd. Van elke variabele of parameter wordt van te voren vastgesteld tot welke klasse deze behoort. Dit leidt tot een meer gecontroleerde vorm van polymorfisme, op basis van overerving (pag. C1/26). Op elke plek waar een variabele van een bepaald type (van een bepaalde klasse) wordt verwacht, is ook een variabele van een van zijn subtypes (subklassen) toegestaan.

2.4.11 Objecten als referenties

In de meeste object-georiënteerde talen bevatten variabelen referenties (pag. C1/19) naar objecten. Dit betekent dat het heel eenvoudig is om netwerken van objecten te construeren die informatie met elkaar delen. Als voorbeeld toont Figuur 2.12 hoe de record-structuur met sharing uit Figuur 2.4 met behulp van objecten kan worden opgebouwd: bij het maken het maken van beide boek-objecten wordt hetzelfde argument a meegegeven.

Het gebruik van referenties betekent dat variabelen slechts namen voor objecten zijn, en zelf geen data bevatten. Een toekenning van een object aan een variabele heeft dus slechts het effect dat de variabele naar dat object wijst. Een toekenning kan dus *niet* gebruikt worden om data te *kopiëren* — daarvoor is het uitvoeren van een *clone*-opdracht of –methode vereist.

Verder houdt het gebruik van referenties in dat de door objecten in beslag genomen ruimte vrijgegeven moet worden als de objecten niet langer nodig zijn. In sommige talen (C++) is de programmeur hier zelf voor verantwoordelijk. In de meeste object-georiënteerde talen gaat dit echter automatisch, via garbage collection (pag. C1/21).

2.5 Belangrijke object-georiënteerde talen

De eerste object-georiënteerde taal is Simula-67, een uitbreiding van Algol-60 met klassen, ontwikkeld door de Noren Ole-Johan Dahl en Krysten Nygaard. Van de diverse andere object-georiënteerde talen bespreken we Smalltalk (pag. C1/32), Eiffel (pag. C1/32), C++ (pag. C1/33) en Java (pag. C1/33).

2.5.1 Smalltalk

De ontwikkeling van Smalltalk-80 heeft een belangrijke bijdrage geleverd tot de verdere verspreiding van object-oriëntatie. Smalltalk is een produkt van het Xerox Palo Alto Reserach Center (Xerox PARC), waar ook de eerste experimenten met het gebruik van de muis en grafische user interfaces zijn uitgevoerd. Smalltalk wordt dan ook geleverd met een krachtige programmeeromgeving, met ondermeer een standaardverzameling klassen en *browsers* om daarin interaktief te zoeken naar herbruikbare componenten. Smalltalk is een dynamisch getypeerde taal, waarin late binding een belangrijke rol speelt.

2.5.2 Eiffel

De taal Eiffel, ontwikkeld in de jaren ’80 door Bertrand Meyer uit Frankrijk, legt juist de nadruk op *statische typering*. Hierdoor kunnen aanmerkelijk meer fouten al tijdens compilatie ontdekt worden. Bovendien levert het, in vergelijking met bijvoorbeeld Smalltalk, efficientere code op, omdat de compiler kan profiteren van de extra type-informatie.

2.5.3 C++

Naast de “zuivere” object-georiënteerde talen bestaat er een groot aantal talen waarin het wel mogelijk is om in een object-georiënteerde stijl te programmeren, maar waarin dat niet afgedwongen wordt. Vaak zijn dit uitbreidingen op al bestaande talen. Dergelijke “object-based” talen zijn vooral van belang wanneer een organisatie al aanmerkelijke ervaring met een niet-object-georiënteerde taal heeft, zoals C, en stap voor stap wil overstappen naar de object-georiënteerde versie, C++.

Elk C (pag. C1/23) programma is ook een C++ programma. Het is dus mogelijk om in C++ op de traditionele, imperatieve manier te programmeren. Als uitbreiding op C, ondersteunt C++ klassen, objecten, encapsulatie, excepties, en een expliciet aan te geven vorm van dynamic binding. C++ kent volledige ondersteuning voor meervoudige overerving. Het dynamisch geheugengebruik van C++ sluit aan op dat van C, en C++ kent dus geen garbage collection.

2.5.4 Java

De meest recente telg in de familie van object-georiënteerde talen is Java. Op het moment van schrijven is de belangstelling voor Java overweldigend, wat vooral komt door het gemak waarmee Java gebruikt kan worden voor het bouwen van Internet applicaties.

De voorbeelden uit Sectie 2.4 zijn allemaal in Java geschreven. Een van de ontwerpdoelen van Java was het ontwikkelen van een kleine taal, waarin de belangrijkste concepten van C++ beter tot hun recht kwamen. Op het eerste gezicht lijkt Java dan ook op C++. Er zijn echter een aantal belangrijke verschillen met C++:

- In Java is het niet mogelijk om met pointers te rekenen, waarmee een bron van veel fouten in C en C++ programma’s is geëlimineerd.
- Java is aanmerkelijk sterker getypeerd dan C++.

- Java kent garbage collection, C++ niet.
- Java heeft geen meervoudige overerving, C++ wel.

Java kan direct worden gecompileerd, maar kan ook worden vertaald naar zogenaamde *byte code*, die dan door een *Java Virtual Machine* kan worden uitgevoerd: de principes hierachter worden beschreven in par. 3.3.

Bij de taal Java horen een aantal klassen met een speciale status. De belangrijkste is de *applet*-klasse, waaruit byte code gegeneerd kan worden die in World Wide Web home-pagina's kan worden opgenomen, en die dan door *web-browsers* (zoals Netscape Navigator) kan worden uitgevoerd.

2.6 Functioneel programmeren

De basis van functioneel programmeren is buitengewoon eenvoudig: het gaat om het uitrekenen van expressies (pag. C1/11). Expressies uitrekenen hebben we op de lagere school geleerd ($3+3$ wordt 6, $5*(3+3)=30$, enz.), en kunnen we allemaal. In het functionele paradigma wordt een poging gedaan het programmeren zo veel mogelijk op het uitrekenen van expressies te laten lijken. Een functioneel programma bestaat uit een serie functie-definities, samen met een expressie die, gebruik makend van de gedefinieerde functies, de gewenste eindwaarde beschrijft.

In functionele talen is het imperatieve geheugenmodel geheel verlaten: er zijn geen variabelen die van waarde kunnen veranderen, er zijn geen assignments, geen referenties, geen side effects, geen opdrachten, geen while-loops, en geen procedures. In plaats daarvan is zijn de concepten functie en expressie aanmerkelijk versterkt, in het bijzonder door het gebruik van lijsten (pag. C1/35), polymorfisme (pag. C1/37), en hogere-orde functies (pag. C1/37).

Functionele talen zijn eenvoudiger dan imperatieve en objectgeoriënteerde talen. Om deze redenen worden ze veel gebruikt in het universitaire onderwijs: aan de hand van een functionele taal zijn veel van de programmeerconepten gemakkelijker uit te leggen en te begrijpen.

Hier zullen we de belangrijkste functionele concepten uitleggen aan de hand van een aantal voorbeelden. Onze voorbeelden staan in Figuur 2.13. Ze zijn geschreven in de taal Haskell, een functionele taal die zowel in het onderwijs als voor het schrijven van systeemprogrammatuur wordt gebruikt.

2.6.1 Functies

Wanneer we door de voorbeelden van Figuur 2.13 heen lopen, vallen een aantal zaken op. Allereerst zijn functies *getypeerd*: De functie `square` bijvoorbeeld heeft het type `Int -> Int`, wat wil zeggen dat `square` een integer waarde verwacht, en weer een integer waarde als resultaat oplevert.

Voor de functie `square` is één definiërende regel, die zegt hoe het kwadraat van het argument berekend kan worden. Voor de functie `fac` (faculteit) zijn *twee* definierende regels: een voor het geval dat het integer argument de waarde `0` heeft, en een voor de resterende gevallen. De functie `fac` wordt gedefinieerd via *pattern matching* (patroonherkenning, gevalsonderscheiding) op het argument. De functie `fac` is *recursief*, de tweede definitie roept `fac` zelf weer aan, maar voor een lagere argumentwaarde.

2.6.2 Lijsten

Functionele talen kennen bijzondere ondersteuning voor *lijsten*. Het type "lijst van integers" wordt genoteerd via `[Int]`. De eenvoudigste lijstnotatie is bijvoorbeeld `[1, 2, 5]` voor een lijst met de drie elementen 1, 2, 5, of `[]` voor de lege lijst.

De definitie van `sum` gebruikt patroonherkenning om de som van een aantal elementen uit te rekenen. Hierin verwijst `n:x` naar een lijst die begint met element `n`, gevolgd door de resterende elementen `x`.

De uit de wiskunde bekende notatie

$$\{\, i \mid i \in \{1,..,n-1\},\ n \,\mathbf{mod}\, i = 0 \,\}$$

voor het opbouwen van een verzameling getallen die aan een bepaalde voorwaarde voldoen (in dit geval dat n deelbaar is door dat getal), wordt direct ondersteund voor lijsten, zoals te zien in de definitie van de functie `factors`.

```
square :: Int -> Int
square x = x * x
```
Voorbeeld: square 5 = 25

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac(n-1)
```
Voorbeeld: fac 4 = 24

```
sum :: [Int] -> Int
sum [] = 0
sum (n:x) = x + sum(x)
```
Voorbeeld: sum [1, 2, 3] = 6

```
factors :: Int -> [Int]
factors n = [ i | i<-[1..n-1],
                  n `mod` i == 0 ]
```
Voorbeeld: factors 6 = [1, 2, 3]

```
perfect :: Int -> Bool
perfect n = sum(factors n) == n
```
Voorbeeld: perfect 6 = True; perfect 7 = False

```
ifilter :: (Int -> Bool) -> [Int] -> [Int]
ifilter p l  = [ a | a <- l, p(a) ]
```
Voorbeeld: ifilter perfect [5, 6, 7, 27, 28, 29] = [6, 28]

```
map :: (t -> u) -> [t] -> [u]
map f []    = []
map f (a:x) = f a : map f x
```
Voorbeeld: map square [1,2,3,4] = [1,4,9,16]

Figuur 2.13: Serie kenmerkende functie-definities in de taal Haskell.

2.6.3 Hogere-orde functies

Functies kunnen als argument meegegeven worden aan andere functies. De functie `ifilter` kiest uit een lijst die elementen die voldoen aan bepaald predicaat. Het eerste argument van `ifilter` is dat predicaat, met type `Int -> Bool`. Het tweede argument is, net als het resultaat, de lijst van integers, met type `[Int]`. In de definitie van `ifilter` is `p` het predicaat, dat wordt toegepast op `a` via `p(a)`.

2.6.4 Polymorfisme

Naast `[Int]` kunnen net zo goed lijsten van strings `[String]`, of andere lijsten geconstrueerd worden. De generalisatie naar lijst-types in het algemeen wordt genoteerd via `[t]`, waarbij `t` een willekeurig type kan zijn. Types met een dergelijke type-variabele worden *polymorf* genoemd.

Polymorfe functies kunnen omgaan met argumenten van diverse types. Zo kan `map` gebruikt worden om een bepaalde functie op elk element van een lijst toe te passen. De `map` van `square` op bijvoorbeeld de lijst [1,2,3,4] levert dus een lijst van kwadraten op: [1,4,9,16]. De argumenten van map zijn: (1) een functie die een argument van type `t` omzet in een waarde van type `u`; en (2) een lijst waarin elementen van type `t` zitten. Hieruit wordt een nieuwe lijst van resultaat waardes van type `u` geconstrueerd.

2.6.5 Belangrijke functionele talen

Een van de eerste talen met speciale ondersteuning voor lijsten en hogere-orde functies was LISP, ontwikkeld door de Amerikaan McCarty in de jaren 50. LISP is een ongetypeerde taal, die naast functies en expressies ook imperatieve opdrachten, zoals de assignment, kent. LISP wordt veel gebruikt voor programmatuur op het gebied van kunstmatige intelligentie.

Standard ML is een sterk getypeerde functionele taal, in de jaren 80 ontwikkeld door de groep van Robin Milner uit Edinburgh. Hoewel het hoofddoel functioneel programmeren is, staat Standard ML assignments, referentie-types, en excepties toe. Standard ML kent een bijzonder krachtig modulariseringsmechanisme.

Haskell behoort tot een familie van talen die gebaseerd is op *lazy evaluation*. Dit is een manier van uitrekenen van expressies, waarbij een subexpressie pas dan wordt uitgerekend wanneer dat noodzakelijk is (voor het evalueren van bijv. $0 * x$ hoeft de waarde van x niet bekend te zijn). Dit maakt het mogelijk met oneindige datastructuren te rekenen. Haskell bevat bovendien *type classes* (groepen van types waarop een aantal gemeenschappelijke functies is gedefinieerd), en *monads*, een manier om imperatief programmeren te integreren met het functionele paradigma.

2.7 Diverse programmeerstijlen

2.7.1 Logisch programmeren

In het *logisch programmeren* wordt de *formele logica* gebruikt om antwoorden op vragen te verkrijgen. Logische programmeertalen worden veel gebruikt in kennisbanken. Hierin worden feiten over een bepaald applicatie domein opgeslagen, samen met logische regels om kennis uit deze feiten af te leiden, en vragen over deze feiten te kunnen beantwoorden. Bij het formuleren van deze regels wordt geen gebruik gemaakt van imperatieve concepten, zoals assignments.

De bekendste logische programmeertaal is Prolog. Een voorbeeld van een Prolog-regel is de volgende formule:

```
vader(V,K) :-
  kindvan(K,V), man(V).
```

Hiermee wordt uitgedrukt dat we kunnen afleiden dat `V` de vader van `K` is als we kunnen afleiden dat `K` een kind van `V` is, en als we bovendien weten dat `V` een man is.

Combineren we bovenstaande regels met de volgend *feiten*:

```
kindvan(iwan, karamazow).
man(karamazov).
```

dan levert de *vraag*

```
? vader(iwan, karamazov)
```

het antwoord `yes` op. We kunnen ook een *open* vraag stellen:

```
? vader(iwan, X)
```

en dan is het antwoord

```
X = karamazov
```

Een wat ingewikkeldere regel is de volgende:

```
broers(M1, M2) :-
  kindvan(M1, X), man(M1),
  kindvan(M2, X), man(M2),
  M1 ≠ M2.
```

Combineren we deze regels met de volgende feiten:

```
kindvan(aljosja, karamazow).
kindvan(iwan, karamazow).
kindvan(dmitri, karamazow).
man(aljosja).
man(iwan).
man(dmitri).
```

dan levert de *vraag*

```
? broers(aljosja, X)
```

de volgende twee antwoorden:

```
X = iwan;
X = dmitri;
```

Een logisch programma bestaat uit een serie regels en feiten, die *Horn*-clauses worden genoemd. Een Horn clause heeft de vorm: P :- $Q_1,...,Q_n$. Hierin is P de *conclusie*, en zijn $Q_1,...,Q_n$ de *condities*. Soms heeft een regel in het geheel geen condities: dan wordt de regel ook een *feit* genoemd, en kortweg als "P." genoteerd. Een Horn clause drukt uit dat P geldt als we kunnen afleiden dat $Q_1,...,Q_n$ gelden.

2.7.2 Parallel programmeren

Een aantal programmeertalen levert constructies waarmee het mogelijk is processen gelijktijdig (*parallel*) uit te voeren. De hieraan ten grondslag liggende principes worden besproken in de secties *** PARALLEL PROGRAMMEREN ***.

2.7.3 Programmeren met scripts

Scripttalen zijn gericht op het snel en flexibel oplossen van allerlei problemen op het gebied van tekstverwerking, gegevens- en systeembeheer, en gebruikersinterfaces. Deze talen zijn meestal ongetypeerd en programma's worden door een interpretator uitgevoerd; ze bieden uitgebreide mogelijkheden voor tekstbewerking (stringmanipulatie, herkennen van reguliere expressies), associatieve tabellen, dynamische arrays, en geven toegang tot de volledige functionaliteit van het onderliggende bedrijfssysteem. Voorbeelden zijn Rexx, Perl, Tcl en Python.

Ter illustratie bespreken we enkele voorbeelden van de scripttaal Tcl (*Tool Control Language*) die bedoeld is als ingebedde besturingstaal voor allerlei applicaties. Aangezien Tcl meestal gebruikt wordt voor het maken van gebruikersinterfaces met behulp van Tk (*Tool Kit*) een componentenbibliotheek voor het bouwen van gebruikersinterfaces, spreekt men ook vaak over Tcl/Tk. Tcl heeft enkele uitgesproken eigenschappen die afwijken van die van meer traditionele talen:

- Alle waarden in Tcl scripts zijn strings. Bovendien zijn Tcl scripts zelf strings die kunnen optreden als argumenten van functies of als resultaat van een functie opgeleverd kunnen worden.
- Alle commando's in Tcl bestaan uit een rij *woorden* en hebben de vorm `cmnd arg1 arg2 ... argn`. Commando's leveren na uitvoering een string als resultaat op.
- Met behulp van { en } kunnen rijtjes woorden letterlijk gegroepeerd worden.
- Een rij woorden wordt opgevat als een commando door ze tussen `[` en `]` te plaatsen.
- Woorden staan voor zichzelf, tenzij voorafgegaan door `$`, dan stellen ze een variabele voor waarvan de waarde genomen moet worden.

Op grond van deze regels hebben *alle* statements in Tcl dezelfde vorm. De toekenning van `44` aan variabele `a` ziet er, b.v. uit als:

```
set a 44
```

Een while-loop is in feite het commando `while` gevolgd door twee argumenten, een test en een serie commando's:

```
while {$p>0} {
 ...
}
```

Een proceduredeclaratie, tenslotte, bestaat uit het `proc` commando gevolgd door drie argumenten, de naam van de procedure, zijn argumentenlijst en de body van de procedure:

```
proc power {base p} {
  ...
}
```

Op grond van het bovenstaande kunnen we de complete definitie van `power` geven:

```
proc power {base p} {
  set result 1
  while {$p>0} {
    set result [expr $result*$base]
    set p [expr $p-1]
  }
  return $result
}
```

Een aanroep `power 2 6` zal `64` opleveren. Tcl ondersteunt geen infix-expressies (d.w.z., expressies waarin de operator *tussen* de operanden staat zoals in $a+b$). Dergelijke expressies kunnen echter met het commando `expr` gesimuleerd worden.

De uniforme structuur van commando's maakt het eenvoudig om nieuwe commando's en controlestructuren aan Tcl toe te voegen. Tcl biedt bovendien vele ingebouwde commando's voor stringmanipulatie, formattering, herkennen van reguliere expressies, manipuleren van lijsten en tabellen, invoer/uitvoer, aanroepen van systeemcommando's en meer.

Het is ook zeer gebruikelijk dat Tcl scripts zelf weer Tcl scripts genereren. Een voorbeeld is het vastleggen van het gebruikersprofiel van, bijvoorbeeld, applicaties om nieuws of email te lezen. De gebruiker kan in dergelijke applicaties allerlei instellingen aanpassen zoals kleur, omvang en plaats van onderdelen van het gebruikersinterface. Na afloop van een sessie worden alle instellingen weggeschreven als Tcl script dat bij een volgend gebruik van de applicatie uitgevoerd wordt om de laatst gebruikte instellingen opnieuw binnen de applicatie vast te leggen.

De combinatie van Tcl met Tk geeft echter pas werkelijk indrukwekkende uitdrukkingskracht. De twee commando's:

```
button .b -text "Hello!" -command exit
pack .b
```

zijn, bijvoorbeeld, voldoende om een gebruikersinterface te creëren. Het `button`-commando definieert een knop met als interne programmanaam `.b`, als aan de gebruiker te tonen tekst `Hello!` en als uit te voeren actie het commando `exit`.

Het `pack`-commando zorgt ervoor dat de zojuist gedefinieerde knop ook op het scherm verschijnt. Indrukken van deze knop doet het gebruikersinterface verdwijnen doordat de uitvoering van het Tcl programma beëindigd wordt door het `exit` commando.

Tk biedt commando's voor het creëren van allerlei elementen van het gebruikersinterface zoals scrollbars, lijsten, tekstvelden, menu's, en tekstuele en grafische editors. Al deze commando's zijn geparameteriseerd met Tcl commando's die definiëren wat er in allerlei situaties moet gebeuren, zoals het indrukken van een knop, het indrukken of verplaatsen van de muis, of het verschuiven van een scrollbar.

Samenvattend is Tcl/Tk een scripttaal die zijn kracht ontleent aan een interpretator die het uitvoeren, genereren en manipuleren van Tcl scripts mogelijk maakt en aan de beschikbaarheid van zeer hoog niveau primitieven (zowel de ingebouwde commando's als de Tk bouwstenen). Hierdoor speelt Tcl de rol als "lijm" om deze primitieven met elkaar te combineren.

2.7.4 Visueel programmeren

Alle tot dusver besproken talen zijn *textueel* van aard. Een *visuele* taal biedt daarnaast mogelijkheden om te programmeren met behulp van iconen, keuze-menus, tekenen met behulp van de muis, enz.

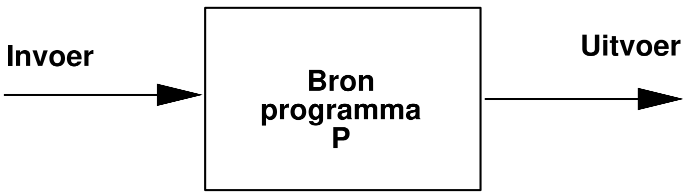
Visueel programmeren maakt het makkelijker een nieuwe taal te leren, doordat met de muis slechts uit een beperkt aantal mogelijkheden gekozen hoeft te worden. Daarnaast is visueel programmeren bijzonder geschikt voor toepassingsgebieden met een duidelijk visueel karakter, zoals het construeren van grafische user interfaces, of het grafisch modelleren van bijvoorbeeld administratieve systemen.

Een zeer bekend voorbeeld van een visuele taal is Microsoft Visual Basic. Deze taal is nauw verweven met Microsoft Windows, en biedt expliciete (visuele) ondersteuning om grafische user interfaces en database systemen te consturen. Bij het ontwerp van Visual Basic is geprobeerd de drempels om de taal te gaan gebruiken zo laag mogelijk te houden. Daarom is gekozen voor een visuele uitbreiding op de taal BASIC, die ook gemak voor de programmeur als ontwerpdoel had. In Visual Basic hoeven geen type declaraties gegeven te worden. De taal ondersteunt objecten, die verborgen en geëxporteerde methoden kunnen bezitten. Overerving is niet mogelijk.

Visual Basic draait uitsluitend op het Microsoft Windows platform. Wel maakt het maximaal gebruik van dit windows platform: het is mogelijk rechtstreeks met (onderdelen van) andere Microsoft applicaties (bijvoorbeeld uit Excel) te comminuiceren, en deze als Visual Basic object te manipuleren.

3 Uitvoeren van programma's

Op welke manier worden programma's door een computer uitgevoerd? Het beoogde effect is dat een programma P (geschreven in programmeertaal L, of kortweg, P_L) zijn *invoer* inleest en vervolgens de beoogde *uitvoer* oplevert zoals in Figuur 3.1 is te zien. We abstraheren van de aard van invoer en uitvoer; het kunnen batchgewijze ingevoerde/uitgevoerde tekstbestanden



Figuur 3.1: Programma met invoer en uitvoer

zijn, maar ze kunnen ook via een grafisch gebruikersinterface, interactief, in- of uitgevoerd worden.

Er bestaan globaal twee manieren om een programma P_L uit te voeren. Een *interpretator* (een "tolk") is een programma dat P statement voor statement inleest en elk statement direct uitvoert. Een *compiler* (een "vertaler") is een programma dat P in zijn geheel inleest, uitvoerig analyseert, en tenslotte een nieuw machinetaal programma P_M genereert dat qua functionaliteit gelijk is aan het oorspronkelijke programma P_L maar geoptimaliseerd is naar executiesnelheid. In de praktijk komen er ook hybride vormen voor waarbij een interpretator een deel van de voorbewerkingen en optimalisaties van een compiler verricht.

Voordat we dieper ingaan op interpretators en compilers is het nuttig om de voor- en nadelen van deze technieken met elkaar te vergelijken (Figuur 3.2).

De tijd voor inlezen en voorbewerken is klein voor een interpretator want er wordt direkt begonnen met het uitvoeren van programma P. Voor een compiler is deze tijd groot (inlezen, analyseren, codegenereren). Een hybride systeem ligt qua tijd tussen deze uitersten in.

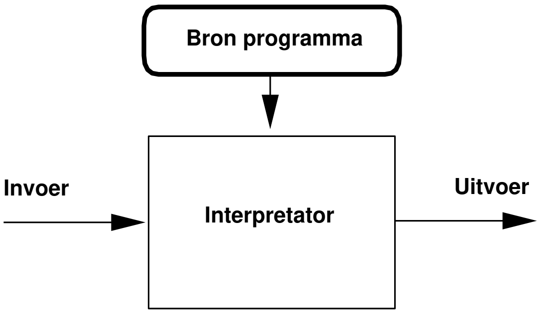
De uitvoeringssnelheid van P is laag bij een interpretator en hoog bij een compiler (dankzij de van te voren uitgevoerde analyses en optimalisaties). Een hybride systeem ligt wederom tussen deze uitersten in.

Fouten kunnen door een compiler vroegtijdig opgespoord worden (dankzij de analyse voor uitvoering), dit is bij een interpretator niet mogelijk.

Foutmeldingen tijdens het uitvoeren van P zijn het beste voor

	Pure interpretator	Compiler	Hybride
Voorbewerken	++	−/−−	−
Uitvoeringssnelheid	−−	++	−/+
Vroeg opsporen fouten	−−	++	−/+
Foutmeldingen	++	−/+	+
Eenvoud	++	−−	+
Voorbeelden	Tcl, Rexx	C	Perl, Java
Zie verder sectie	3.1	3.2	3.3

Figuur 3.2: Vergelijking van interpretator en compiler



Figuur 3.3: Struktuur van een interpretator

een interpretator (want de precieze brontekst die de fout veroorzaakt is bekend), gevolgd door een hybride implementatie (de brontekst moet uit een tussenrepresentatie gereconstrueerd worden), en zijn het slechtste voor een compiler (er is een complexe afbeelding nodig tussen uitgevoerde machine-instructies en de brontekst).

De eenvoud van interpretators en compilers loopt sterk uiteen. Pure interpretators zijn relatief eenvoudig, hybride systemen zijn iets minder eenvoudig, compilers zijn doorgaans zeer complex. Direkt gerelateerd aan eenvoud zijn flexibiliteit en onderhoudbaarheid. Dit verklaart dan ook waarom bij het experimenteren met nieuwe talen vaak eerst een interpretator wordt gebouwd.

3.1 Interpretators

Vergeleken met de aanzienlijke complexiteit van compilers, zoals we die in de komende pagina's zullen bespreken, zijn interpretators een wonder van eenvoud: een interpretator is eenvoudigweg een programma dat een gegeven bronprogramma statement voor statement inleest en uitvoert zonder verdere analyse of optimalisatie (zie Figuur 3.3). Het is nuttig om een onderscheid te maken tussen *pure interpretators* die werken zoals net geschetst en hybride systemen die een mengvorm zijn van een pure interpretator en een compiler (zie Sectie 3.3).

Omdat een pure interpretator geen enkele voorbewerking of analyse verricht kan de uitvoering van het bronprogramma direct beginnen; er is geen lange compilatietijd of edit-compileer-voeruit cyclus. Pure interpretators ondersteunen daardoor *rapid prototyping*: het snel construeren van een werkend prototype. Nadeel is wel dat eventuele fouten pas tijdens uitvoering ontdekt worden.

De uitvoering van een programma met behulp van een pure interpretator is typisch 10–100 keer trager dan uitvoering met behulp van een compiler. Hiervoor zijn diverse oorzaken. Als statements bij herhaling uitgevoerd moeten worden zal de interpretator ze keer op keer opnieuw moeten inlezen en uitvoeren. Bovendien moet de interpretator steeds opnieuw tijdens uitvoering allerlei controles uitvoeren die bij gebruik van een compiler van te voren (dus eenmalig) door de compiler verricht worden.

Tegenover het nadeel van trage uitvoering van bronprogramma's biedt de pure interpretator allerlei voordelen die een verklaring vormen voor de grote populariteit van deze techniek.

Ten eerste, is de pure interpretator veel eenvoudiger qua interne werking en struktuur dan een compiler. Dit bevordert onderhoudbaarheid en flexibiliteit.

Ten tweede, is het zeer eenvoudig om in een interpretator functionaliteit aan te bieden die een compiler veel moeilijker kan bieden, zoals het geven van goede foutmeldingen die direkt kunnen verwijzen naar de brontekst, het tijdens uitvoering (dynamisch) inladen van nieuwe programma's, en programmeerhulpmiddelen op het niveau van de brontekst zoals interactieve debugging, het traceren van de waarden van variabelen, en het zetten van breekpunten in het programma.

Ten slotte vereist een interpretator geen machine-specifieke kennis. Terwijl een compiler aangepast moet worden voor elke nieuwe machine waarvoor hij code moet opleveren, volstaat het bij een interpretator om alleen de interpretator naar de nieuwe machine over te brengen. Als de interpretator in een gestandaardiseerde, portabele, taal geschreven is, dan is dit een relatief eenvoudige taak.

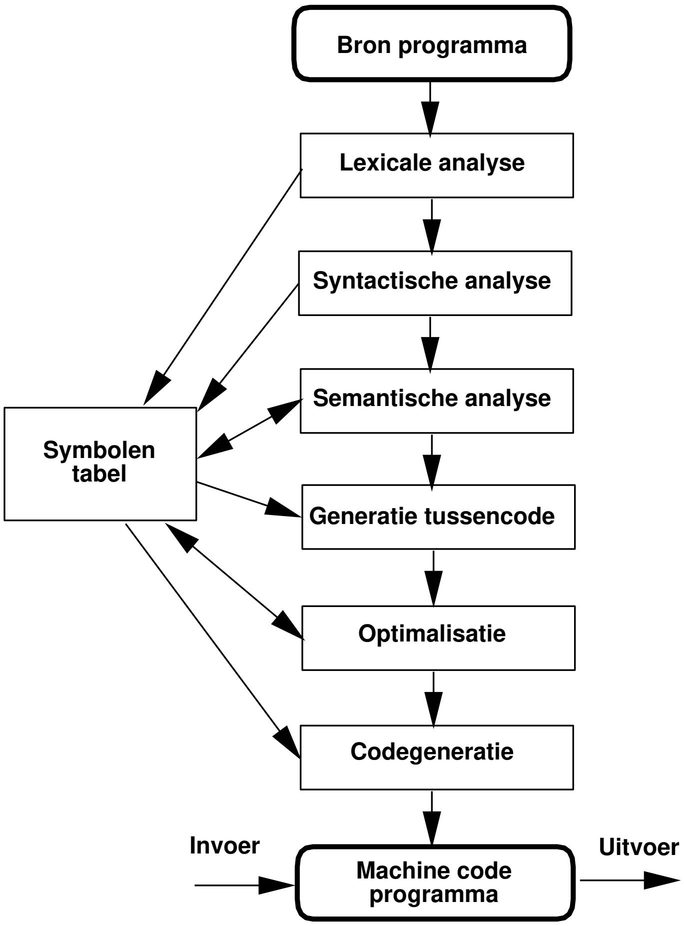
De hier genoemde voordelen (eenvoudige implementatie, grote functionaliteit, portabiliteit) verklaren de populariteit van pure interpretators als implementatietechniek voor scripting talen zoals Tcl, Awk, Rexx, en commandotalen zoals de UNIX shell. Deze voordelen zijn ook van toepassing op de *programma's* die in deze talen geschreven zijn,

3.2 Compilers

Een compiler voor taal L heeft een bronprogramma P_L als invoer en produceert uiteindelijk een machinetaal programma P_M dat direkt door de M-processoruitgevoerd kan worden (zie Figuur 3.4). Een compiler bestaat meestal uit een zestal fasen die allemaal gebruik maken van een symbolentabel. De eerste drie fasen *analyseren* de broncode, de laatste drie gebruiken de resultaten van deze analyses om machinecode te *genereren*.

3.2.1 Lexicale analyse

In eerste benadering bestaat een programma uit een rij characters in een van te voren bekende characterverzameling zoals ASCII, EBCDIC, of UNICODE. Het doel van *lexicale analyse* (ook: *lexical scanning*) is om deze rij characters om te zetten in een rij *lexicale tekens* die zinvol zijn vanuit het perspectief van de programmeertaal. Voorbeelden zijn de sleutelwoorden (key words), namen (identifiers), numerieke constanten, en layout symbolen (spaties, commentaar), zie Figuur 3.5. De uitvoer van de lexicale analyse is een rij lexicale tekens die elk eventueel van een waarde zijn voorzien, zoals de feitelijke tekst van een naam of van een numerieke constante. In sommige compilers wordt deze informatie al direkt in de symbolentabel gezet.



Figuur 3.4: Struktuur van een compiler

Teken	Voorbeelden	Omschrijving
while	`while`	het keyword "while"
ident	`x`, `flag`	identifiers
num	`0`, `3.14`	getallen
layout	`/* C commentaar */`	layout, commentaar

Figuur 3.5: Voorbeelden van lexicale tekens

De lexicale tekens van een taal worden gedefinieerd door *reguliere expressies* die tezamen de *lexicale syntax* van de taal vastleggen. Een reguliere expressie is allereerst opgebouwd uit individuele characters (die voor zichzelf staan) en characterklassen die verzamelingen van characters weergeven zoals `[a-z]` (alle kleine letters), `[a-zA-Z]` (alle letters), `[0-9]` (alle cijfers). Samengestelde expressies worden gevormd door het achterelkaar plaatsen van reguliere expressies, de alternatief-operator (`|`), de optie-operator (`?`), en de herhalingsoperatoren (`*` voor nul of meer herhalingen en `+` voor één of meer herhalingen). Haakjes kunnen gebruikt worden om subexpressies te groeperen. We geven hiervan drie voorbeelden:

- `[0-9]+`: een geheel getal bestaande uit één of meer cijfers.
- `[a-z][a-z0-9]*`: een identifier beginnend met een kleine letter, gevolgd door nul of meer kleine letters of cijfers.
- `[0-9]+.[0-9]+e(+|-)?[0-9]+`: Eenvoudige floating point getallen bestaande uit cijfers, een decimale punt, cijfers, een `e`, gevolgd door een optioneel teken en cijfers.

Lexicale analyse kan een beperkt aantal foutmeldingen produceren zoals, bijvoorbeeld, het voorkomen van tekstuele constanten (strings) of commentaren die niet goed zijn afgesloten.

3.2.2 Syntactische analyse

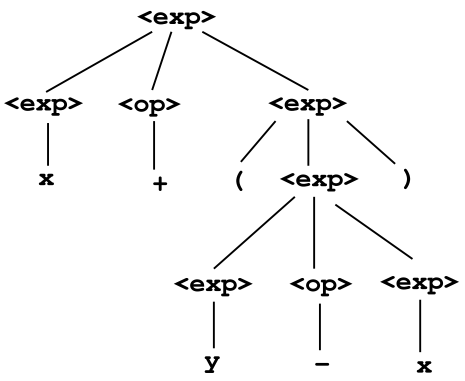
Gegeven een rij lexicale tekens die uit de fase van lexicale analyse komen, moeten we vaststellen of deze rij tekens een acceptabel programma vormt. Uit de taalkunde is bekend dat een *grammatica* (of ook wel *syntaxis*) gebruikt kan worden om te beschrijven welke opeenvolgende woorden een legale zin vormen. Op vergelijkbare wijze worden grammatica's in de informatica gebruikt: om vast te stellen welke rijen lexicale tekens een legaal programma vormen. Let op dat het hierbij puur om de *vorm* van het programma gaat; betekenis (semantiek) komt pas later aan de orde in Secties 3.2.3 en 3.2.5

Laten we aan de hand van een voorbeeld eens bekijken hoe we grammatica's kunnen gebruiken om de struktuur van rijen tekens vast te leggen. We nemen als voorbeeld eenvoudige rekenkundige expressies zoals x, x+y, x+(x+y), enz. Dit soort expressies wordt beschreven door de volgende grammatica (de regels zijn voor het gemak genummerd):

```
(1)  <exp> ::= <exp> <op> <exp>
(2)  <exp> ::= "(" <exp> ")"
(3)  <exp> ::= "x"
(4)  <exp> ::= "y"
(5)  <op>  ::= "+"
(6)  <op>  ::= "-"
```

De lexicale tekens (ook wel *terminale symbolen* genoemd omdat ze geen andere tekens kunnen voortbrengen) in deze grammatica zijn: (,), x, y, +, en -. De zogenaamde *niet-terminale symbolen* van deze grammatica zijn <exp> (die alle expressies kan voortbrengen) en <op> (die alle operatortekens kan voortbrengen). Het principe is nu dat een syntactisch correcte expressie verkregen wordt door te beginnen met het niet-terminale symbool <exp> en steeds opnieuw regels van de grammatica toe te passen zolang dat nog kan, d.w.z. niet-terminale symbolen vervangen door een rechterkant van een grammaticaregel. Op deze manier kunnen we, bijvoorbeeld, de volgende expressies krijgen: x, x+y, en x+(y-x). Dit laatste voorbeeld kunnen we alsvolgt uit <exp> afleiden (toepassen van grammaticaregel (*n*) wordt aangegeven door -*n*->):

```
<exp> -1-> <exp> <op> <exp>
      -3-> x <op> <exp>
      -5-> x + <exp>
      -2-> x + ( <exp> )
```



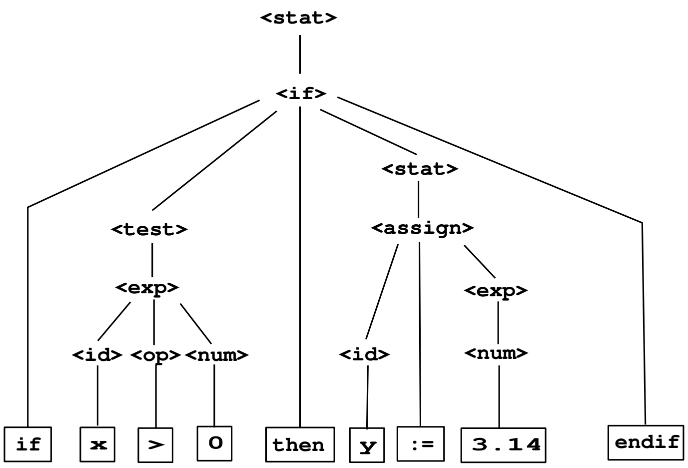
Figuur 3.6: Afleidingsboom voor `x+(y-x)`

```
-1-> x + ( <exp> <op> <exp> )
-4-> x + ( y <op> <exp> )
-6-> x + ( y - <exp> )
-3-> x + ( y - x )
```

Merk op dat we hier steeds de *linker* niet-terminaal vervangen hebben. In het algemeen kunnen we steeds een willekeurige niet-terminaal vervangen. Dezelfde afleiding, waarin de volgorde van vervanging geen rol speelt, kunnen we ook als *afleidingsboom* (engels: *parse tree*) weergeven, zie Figuur 3.6. We kunnen de afleidingsboom zien als bewijs dat de zin `x+(y-x)` volgens de gegeven grammatica door het niet-terminale symbool `<exp>` voortgebracht wordt.

In dit voorbeeld gingen we uit van het niet-terminale symbool `<exp>` en genereerden daarbij een syntactisch correcte zin. In de praktijk komt het omgekeerde echter veel vaker voor: stel vast of een gegeven zin syntactisch correct is en welk niet-terminaal symbool erbij hoort. Dit is nu precies wat *syntactische analyse* (ook: *parseren*, *ontleden*) inhoudt.

Als tweede voorbeeld geven we een grammatica voor eenvoudige statements zoals die in een programmeertaal voor zouden kunnen komen. De grammatica beschrijft twee soorten statements: conditioneel statement (`<if>`) en toekenning (`<assign>`). De lexicale definities voor identifiers (`<id>`), numerieke con-



Figuur 3.7: Afleidingsboom voor `if x > 0 then y := 3.14 endif`

stanten (<id>), en operatoren (<op>) worden hier verder niet besproken.

```
(1)  <stat>   ::= <if>
(2)  <stat>   ::= <assign>
(3)  <if>     ::= "if" <test> "then"
                  <stat> "endif"
(4)  <test>   ::= <exp>
(5)  <exp>    ::= <id>
(6)  <exp>    ::= <num>
(7)  <exp>    ::= <exp> <op> <exp>
(8)  <assign> ::= <id> ":=" <exp>
```

In Figuur 3.7 zien we een voorbeeld van een afleidingsboom voor het statement

```
if x > 0 then y := 3.14 endif
```

zoals dat in een programma zou kunnen voorkomen. Merk op dat onder in de figuur de lexicale tekens met blokjes zijn aangegeven.

Tijdens syntactische analyse kan blijken dat de gegeven tekst (d.w.z. de rij lexicale tekens) nooit tot een syntactisch correct programma kan leiden. In dat geval wordt een zo precies mogelijke foutmelding gegeven die de plaats van de fout in de tekst aangeeft en eventueel een toelichting op de oorzaak. Meestal zal geprobeerd worden om de syntactische analyse, ondanks de fout, voort te zetten om zo mogelijk nog meer fouten in de tekst op te sporen.

3.2.3 Semantische analyse

Tijdens de syntactische analyse is de hierarchische struktuur van een programma vastgesteld. Daarna blijven er nog enkele analyses over die *semantisch* van aard zijn, d.w.z. die niet de syntactische struktuur van het programma betreffen maar de betekenis ervan. De bekendste semantische analyses zijn *naamresolutie* en *typecontrole*.

Een eerste onderdeel van semantische analyse is *naamresolutie*. In het eenvoudigste geval komt dit neer op het vaststellen of een naam die in de programmatekst gebruikt wordt ook in het programma gedeclareerd is. Een complicerende factor hierbij kunnen voorwaartse referenties zijn: namen die gebruikt worden voordat ze gedeclareerd zijn. In het ingewikkelder geval van programmeertalen met blokstructuur, moet niet alleen vastgesteld worden òf een naam gedeclareerd is maar ook wélke declaratie op elke plaats in het programma van toepassing is. In eenvoudige talen zonder blokstruktuur wordt de naamresolutie vaak afgehandeld tijdens de syntactische analyse. In talen waarin zowel blokstruktuur als voorwaartse referenties voorkomen vormt naamresolutie een apart onderdeel van de semantische analyse.

Een tweede onderdeel van de semantische analyse is de *typecontrole* (Engels: *type checking*) waarin gekeken wordt of de manier waarop constanten, variabelen, functies en procedures in het programma gebruikt worden qua typering correct is. Dit houdt onder andere in dat operatoren toegepast moeten worden op operanden van het juiste type (`2+3` is goed, maar `2+"abc"` is fout omdat de +-operator numerieke operanden eist), dat variabelen met een type gedeclareerd zijn dat klopt met het type dat bij hun gebruik vereist wordt (`2+x` eist dat `x` met een numeriek type gedeclareerd is), en dat functies en procedures met het juiste aantal parameters van het juiste type aangeroepen worden. Tijdens de semantische analyse wordt ontdekt waar *impliciete typeconversies* nodig zijn: als de variabele `n` met het type integer gedeclareerd is wordt de expressies `3.14+n` in deze fase vervangen door `3.14+int_to_real(n)` om ervoor te zorgen dat de +-operator—die hier operanden van het type real eist—inderdaad een rechteroperand van het type real krijgt.

3.2.4 Generatie tussencode

De resultaten van syntactische en semantische analyse worden door sommige compilers in een aparte tussencode (Engels: *intermediate code*) vastgelegd. Deze tussencode heeft een simpele struktuur en dient ervoor om de volgende fasen van optimalisatie en codegeneratie te vereenvoudigen. In veel gevallen lijkt de tussencode op de machine-instructies voor een abstracte machine. Een veelvoorkomende tussencode is *drie-adres code* die bestaat uit: een label, naam van de operatie, gevolgd door één of twee operanden. Operanden kunnen zowel constanten zijn of resultaten van eerdere instructies die aangegeven worden met het label van de betreffende instructie. Mpgelijke operanden zijn constante waarden (b.v. `7`), variabelen (b.v. `x`), resultaten van eerdere instructies (aangegeven door het label van die eerdere instructie, b.v. `(0)`, of labels van statements waarnaartoe gesprongen wordt (b.v. `(6)`).

Het statement

```
if x > 7 then y := 3.14*2 else y:=0 endif
```

wordt, bijvoorbeeld, vertaalt naar de volgende drie-adres code:

```
(0) SUB  x, 7       # Bereken x -7
(1) BNZ  (0), (6)   # Ga verder met (6) als
                    # resultaat (0) niet nul
(2) I2R  2          # Converteer 2 naar 2.0
(3) RMUL 3.14, (2)  # Real vermenigvuldiging
(4) ASG  (3), y     # Ken resultaat van (3)
                    # toe aan y
(5) BR   (7)        # Spring over else-tak (6)
(6) ASG  0, y       # Ken 0 toe aan y
```

```
(7)                 # vertaling van volgende
                    # instructies
```

3.2.5 Optimalisatie

Op de representatie van het programma in tussencode kunnen nog allerlei optimalisaties uitgevoerd worden. We laten de belangrijkste daarvan nu de revue passeren.

- *Vereenvoudigen constante expressies* komt neer op het tijdens compilatie uitrekenen van expressies waarin alleen constanten voorkomen.
- *Vereenvoudigen van adresberekeningen* is een speciaal geval van het vereenvoudigen van constante expressies maar nu toegepast op adresberekeningen zoals die nodig zijn om het adres van elementen van een array te bepalen.
- *Elimineren van gemeenschappelijke subexpressies* komt neer op het ontdekken van verschillende voorkomens van dezelfde subexpressie in een programma; de waarde van de subexpressies wordt éénmaal berekend en in een hulpvariabele opgeslagen. Bij ieder volgend voorkomen van de subexpressie wordt de waarde van deze hulpvariabele gebruikt in plaats van de waarde van de subexpressie opnieuw uit te rekenen.
- *Algebraïsche vereenvoudigingen* zijn soms mogelijk door wetten toe te passen zoals

  ```
  x + 0 = x
  0 + x = x
  0 * x = 0
  x * 0 = 0
  1 * x = x
  x * 1 = x
  ```

- Een speciaal geval van algebraïsche vereenvoudiging is *sterktereductie*: het vervangen van duurdere operatoren door goedkopere (hierin is `**` machtsverheffen en `<<` staat voor de bitsgewijze schuifoperatie naar links):

```
x ** 2 = x * x
2.0 * x = x + x
2 * x = x << 1
4 * x = x << 2
8 * x = x << 3
```

Een ander voorbeeld is het vervangen van een dure operatie als stringconcatenatie (||) gevolgd door lengtebepaling door lengtebepaling gevolgd door optelling:

```
length(x || y) = length(x) + length(y)
```

- *Codeverplaatsing* is een techniek die vooral toegepast wordt om herhalingsconstructies (loops) te optimaliseren. Code die niet door de uitvoering van de loop beïnvloed wordt kan beter buiten de loop geplaatst worden. Bovendien kunnen complexe expressies die steeds binnen de loop berekend worden vaak gesplitst worden in een berekening die voor aanvang van de loop uitgevoerd wordt en een eenvoudigdiger berekening die binnen de loop wordt uitgevoerd. Zo kan

  ```
  while(n <= max - 1) {
    /* code die max niet wijzigt */
  }
  ```

  vervangen worden door

  ```
  tmp = max - 1;
  while(n <= tmp) {
    /* code die max en tmp
       niet wijzigt */
  }
  ```

- Een speciaal geval van codeverplaatsing is het *elimineren van dode code*, d.w.z. code die bij geen enkele uitvoering van het programma nog bereikt kan worden. Deze situatie kan ontstaan als gevolg van veelvuldig onderhoud van de code maar ook door het herdefinieren van globale constanten. Zo is het gebruikelijk om tijdens ontwikkeling van een programma een globale constante `debug` te gebruiken en alle hulpcode (extra tests, print statements) die alleen voor de programmeur van belang zijn op te nemen in if-statements die testen of `debug` waar of onwaar is. Zodra de constante `debug` op onwaar gezet wordt, veranderen al dergelijke statements in dode code die weggeoptimaliseerd kan worden.

- In sommige gevallen is het zinvol om procedure-aanroepen direkt te vervangen door de body van de betreffende procedure (Engels: *inline procedure call expansion*). Hierdoor worden de extra kosten die met een procedure-aanroep gepaard gaan vermeden. Dit kan vooral van belang zijn als de aanroep staat binnen een veelvuldig uitgevoerde loop.

Een noodzakelijke voorwaarde voor het kunnen uitvoeren van de bovenstaande optimalisaties is informatie die verzameld wordt door twee globale analysetechnieken: *Control flow analyse* en *Data flow analyse*.

Control flow analyse bepaalt wat de elementaire rijen statements (Engels: *basic blocks*) van een programma zijn: een basic block heeft één ingang, bestaat uit één of meer statements en mag zelf sprongen naar andere basic blocks bevatten. Het resultaat van control flow analyse is een verzameling basic blocks en hun onderlinge samenhang.

Data flow analyse bepaalt de afhankelijkheden tussen statements en expressies uitgaande van definitie en gebruik van variabelen. Een assignment statement fungeert hierbij typisch als een definitie van een variabele terwijl andere statements de variabele gebruiken.

Door de extra analyses die nodig zijn voor het uitvoeren van optimalisaties kunnen sommige fouten in het programma al tijdens compilatie ontdekt worden. Voorbeelden zijn het gebruik van niet-geïnitialiseerde variabelen, en het ontbreken van return-statements in functies.

3.2.6 Codegeneratie

Uitgaande van de geoptimaliseerde tussencode die door de voorgaande compilerfasen is geproduceerd resteert nog één taak: het genereren van feitelijke machine-instructies. Hierbij moeten de volgende problemen opgelost worden:

- *Allocatie van registers* is nodig om te voorkomen dat tussenresultaten van berekeningen telkens naar het geheugen worden weggeschreven en vervolgens weer moeten worden teruggelezen. Het is veel efficienter (zowel qua uitvoeringssnelheid als qua compactheid van de resulterende machine-instructies) om deze tussenresultaten in snelle machineregisters te bewaren. Het allocatieprobleem ontstaat als er meer tussenresultaten te onthouden zijn dan er registers beschikbaar zijn.
- *Selectie van instructies* is nodig om te bepalen welke machine-instructies corresponderen met de gegeven tussencode. Goede selectie van instructies maakt gebruik van zoveel mogelijk bijzondere instructies en faciliteiten van de doelmachine en genereert steeds de goedkoopste rij machine-instructies. Een typisch voorbeeld zijn optellingen waarin één van de operanden `1` of een kleine integer constante is; in dergelijke gevallen is het beter om de snelle increment-instructie te genereren in plaats van de algemene, dus langzamere, optel-instructie.
- *Lokale optimalisaties* (Engels: *peephole optimization*) zijn optimalisaties die uitgevoerd worden op het moment vlak voordat de machine-instructies feitelijk geëmitteerd worden. Het betreft hier het verwijderen van overbodige instructies (overbodige toekenningen, sprongen naar sprong-instructies) en het samenvoegen van opeenvolgende instructies door meer complexe, machine-specifieke, instructies.

3.2.7 Linkers en loaders

Conceptueel is het compilatieproces afgerond nadat de compiler het bronprogramma omgezet heeft in een machinecode programma. Er moeten echter nog een aantal extra stappen gezet worden voordat deze machinecode feitelijk uitgevoerd kan worden.

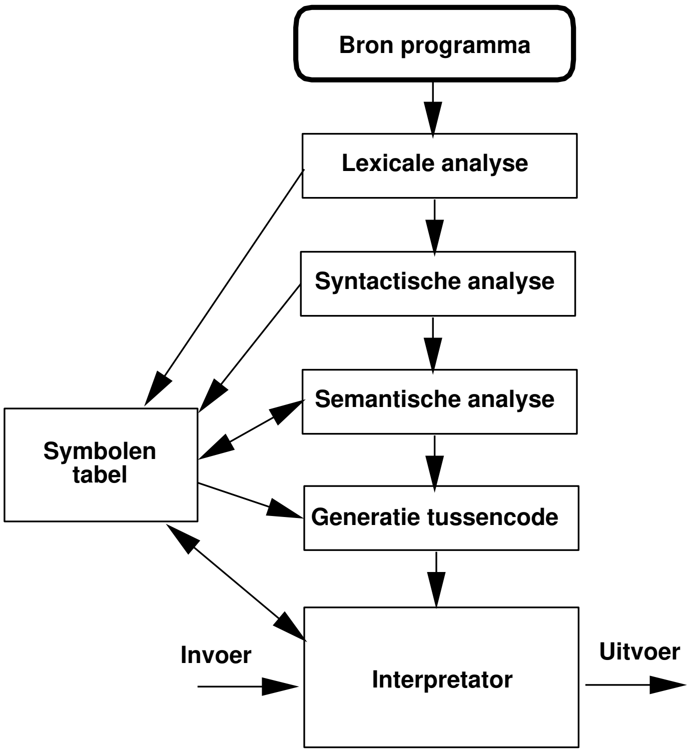
Ten eerste, is de vorm van de gegenereerde machinecode van belang. Als de compiler *assembleertaal* genereert (een voor mensen leesbare, tekstuele, representatie van de machinetaal) dan zal deze eerst door een *assembler* omgezet moeten worden in *binaire vorm* (d.w.z, een machine leesbare, bitsgewijze representatie van de machinetaal).

Ten tweede, is het van belang of een compiler toelaat dat verschillende programmamodules apart gecompileerd kunnen worden (Engels: *separate compilation*). Als dit het geval is, bestaan er nog wederzijdse verwijzingen tussen de apart gecompileerde modules (zoals namen van globale variabelen en van procedures) die opgelost moeten worden vóór uitvoering. De gangbare oplossing is dat elk programmamodule vertaald wordt naar een *objectmodule* dat uit twee delen bestaat: de gegenereerde binaire machinecode, en een symbolentabel met namen en adressen (in de binaire code) van variabelen en procedures. Het formaat van objectmodulen wordt in toenemende mate gestandaardiseerd (ELF, COFF). Een *linkage editor* gebruikt de informatie uit de symbolentabellen om een collectie objectmodulen om te vormen tot één uitvoerbaar programma (Engels: *load module*) dat direkt door het besturingssysteem geladen kan worden (door de *loader*) en in uitvoering kan worden genomen.

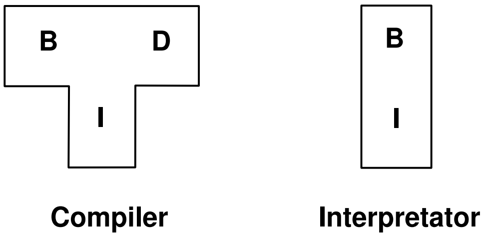
In de meeste systemen is het ook mogelijk om linkage editing en laden *dynamisch* te doen, d.w.z. dat tijdens de uitvoering van een programma andere programmamodulen ingeladen en gelinkt worden. In Windows betreft dit *Dynamic Load and Link (DLL)* files, in Unix systemen spreekt men van *Dynamic Shared Objects (DSO)*.

3.3 Hybride systemen

Hybride systemen vormen een mengvorm tussen pure interpretators en compilers (zie Figuur 3.8) waarin geprobeerd wordt de voordelen van een pure interpretator te behouden en zijn nadelen (i.h.b. de langzame uitvoering) te vermijden. Dit resultaat wordt bereikt door de compilerfasen *lexicale analyse*, *syntactische analyse*, *semantische analyse* en *generatie tussencode* in de interpretator op te nemen en de feitelijk uitvoering van het bronprogramma te baseren op interpretatie van de tussencode.



Figuur 3.8: Struktuur van een hybride systeem



Figuur 3.9: T-diagrammen voor compiler en interpretator

In sommige hybride systemen wordt een boomrepresentatie van het bronprogramma als tussencode gebruikt. In andere systemen wordt een abstracte machine code gebruikt, zoals, bijvoorbeeld, de bytecodes van de Java Virtual Machine.

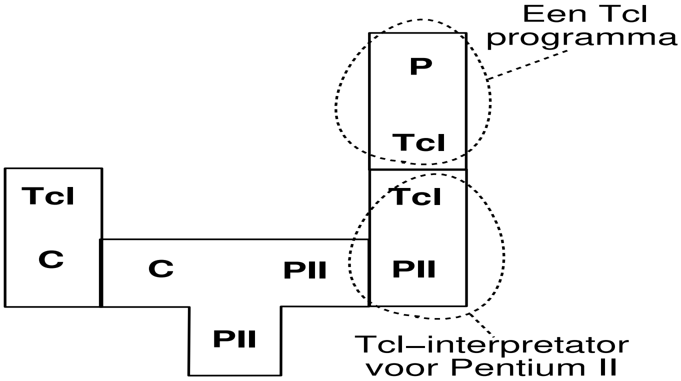
In een hybride systeem is enige tijd nodig voor de voorbewerking en analyse van het bronprogramma, maar niet zoveel als in een compiler. Fouten kunnen vroegtijdig opgespoord worden, er kunnen goede foutmeldingen gegegeven worden doordat er in de interpretator een goede symbolentabel beschikbaar is, en uitvoering is iets trager (typisch: 10 maal) dan uitvoering middels een compiler.

Bekende voorbeelden van hybride systemen zijn Lisp, Python, Perl, en Java.

3.4 T-diagrammen

In het voorafgaande hebben we gezien dat drie talen een rol spelen bij het bespreken van compilers en interpretators: de *brontaal* (B) waarin de programma's geschreven zijn die als invoer aan een compiler of interpretator aangeboden worden; de *implementatietaal* (I) waarin de compiler of interpretator zelf geschreven is; en de *doeltaal* (D) de (machine)taal waarin de uitvoer van de compiler uitgedrukt is. Bovendien spelen ook nog verschillende interne tussentalen een rol die door het éne onderdeel van de compiler of interpretator geproduceerd worden en voor het andere onderdeel weer als invoer dienen.

Deze veelheid aan talen en de rol die ze spelen in een compiler of interpretator leidt er al snel toe dat we het overzicht verliezen.

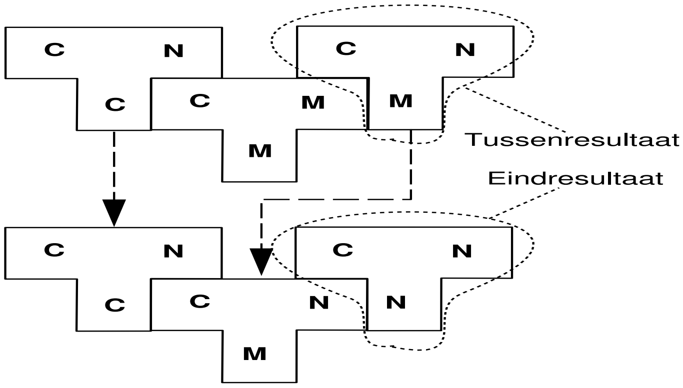


Figuur 3.10: T-diagrammen voor compilatie van Tcl interpretator

Figure 3.9 laat een grafische notatie zien die gebruikt kan worden om het vertalen en combineren van compilers en interpretators grafisch weer te geven. Links staat een T-vormig diagram (vandaar de naam van de notatie) die een compiler voorstelt: de brontaal B wordt vertaald naar een doeltaal D en de compiler zelf is geïmplementeerd in taal I. In de tekst zullen we deze compiler aanduiden als $B_I D$. Rechts zien we een interpretator voor brontaal B, geschreven in implementatietaal I. In de tekst aangeduid als B_I.

Figuur 3.10 laat nu zien hoe, bijvoorbeeld, de in C geschreven interpretator voor Tcl door een C compiler gecompileerd wordt voor een Pentium II (PII) processor. Het resultaat is een Tcl interpretator in Pentium II code die een in Tcl geschreven programma P kan interpreteren.

Figuur 3.11 laat zien hoe C compiler van machine M naar machine N geporteerd kan worden. Gegeven zijn een C compiler die M-code genereert en M als implementatietaal heeft ($C_M M$) en een nieuwe C compiler die N-code genereert en C als implementatietaal heeft ($C_C N$). Allereerst compileren we $C_C N$ met behulp van $C_M M$; het resultaat is een compiler $C_M N$ die we kunnen gebruiken om $C_C N$ nogmaals te compileren. Het resultaat is de beoogde compiler $C_N N$.



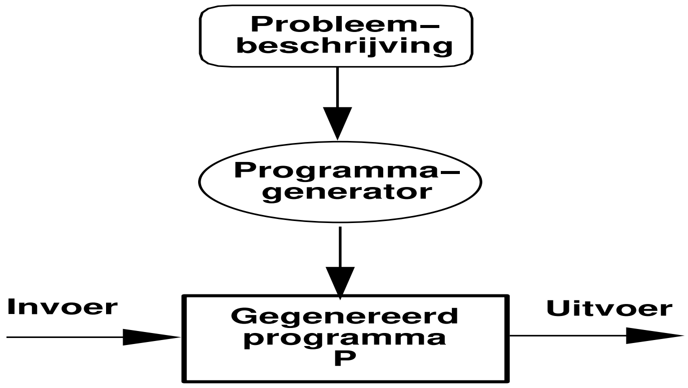
Figuur 3.11: T-diagrammen voor compilatie C compiler

4 Programmageneratoren

In de inleiding van Sectie 3 hebben we in Figuur 3.1 de globale struktuur laten zien van een programma *P* met zijn in- en uitvoer. We gingen er daarbij stilzwijgend van uit dat *P* met de hand door een programmeur geschreven was. Er is echter geen enkele noodzaak voor deze aanname, immers *P* kan ook door een *programmagenerator* zijn voortgebracht zoals in Figuur 4.1 is weergegeven. Een programmagenerator is een gewoon programma met invoer en uitvoer. Het enige bijzondere is dat de uitvoer zelf weer een programma is. De ratio achter een programmagenerator is dat deze veel kennis over een toepassingsgebied bevat die keer op keer opnieuw gebruikt kan worden. Hierdoor kan de door de programmeur te leveren invoer van de generator (de probleembeschrijving) veel korter zijn dan het gegenereerde programma.

Programmageneratoren wordt onder andere gebruikt voor het implementateren van vierde-generatie talen, database- en vraagtalen, en andere domeinspecifieke talen.

In deze sectie gaan we kort in op het gebruik van programmageneratoren voor het construeren van onderdelen van compilers, interpretators en programmeeromgevingen zoals we in de voor-



Figuur 4.1: Programmagenerator

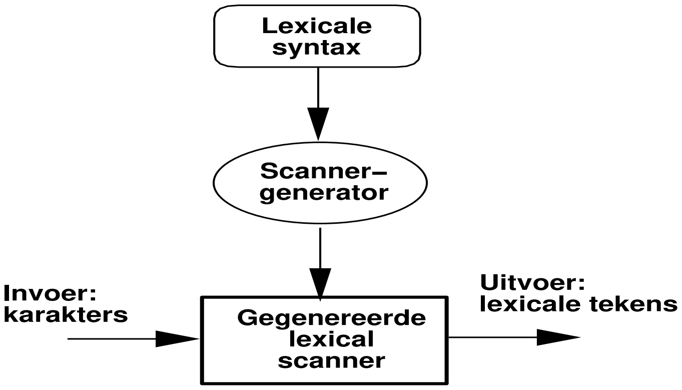
gaande secties besproken hebben.

4.1 Scannergeneratoren

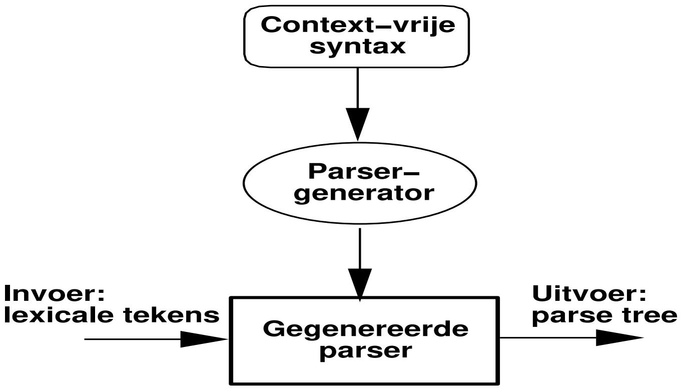
In Sectie 3.2.1 hebben we de taken van lexicale analyse uiteengezet (het combineren van de invoertekst tot lexicale tekens) evenals de beschrijvingswijze van lexicale tekens (reguliere expressies die tezamen een lexicale syntax vormen).

Een *scannergenerator* heeft tot doel om de fase van lexicale analyse (ook wel *scanner* genoemd) automatisch te genereren op basis van een lexicale syntax, zie Figuur 4.2.

De scannergenerator verricht eerst een analyse van de gegeven lexicale syntax waardoor deze omgezet kan worden in een *eindige automaat*: een beslissingstabel met een eindig aantal toestanden en toestandsovergangen die bepaald worden door (verzamelingen) invoercharacters. De uiteindelijke scanner wordt vervolgens gegenereerd door deze eindige automaat om te zetten in equivalente, efficiente, programmacode.



Figuur 4.2: Scannergenerator



Figuur 4.3: Parsergenerator

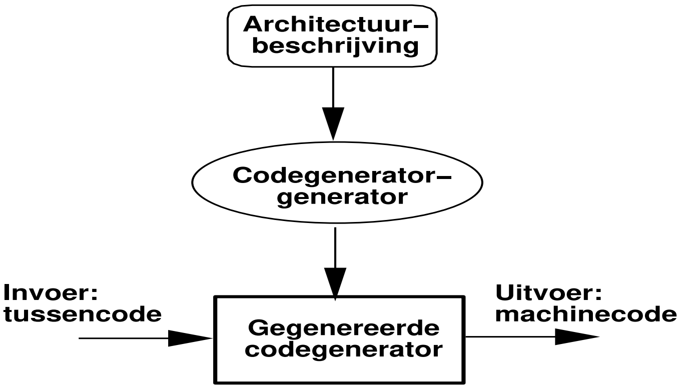
4.2 Parsergeneratoren

In Sectie 3.2.2 hebben we de taken van de syntactische analyse uiteengezet: vaststellen of een rij lexicale tekens overeenkomt met de context-vrije grammatica van de taal en, indien dit het geval is, een bijbehorende afleidingsboom (parse tree) opleveren.

Een *parsergenerator* heeft tot doel om de fase van syntactische analyse (ook wel *parser* genoemd) automatisch te genereren op basis van een context-vrije grammatica, zie Figuur 4.3.

Er is een scala aan technieken beschikbaar om parsers te genereren, die verschillen in de efficientie van de opgeleverde parsers en in de klasse van grammatica's die deze parsers aankunnen. Zoals meestal leidt grotere algemeenheid tot afnemende efficientie. We noemen enkele voorbeelden geordend van beperkt/efficient naar algemeen/minder efficient:

- LL(1), dit is een top-down parseringstechniek die al in de vroegste compilers is toegepast en daar *recursive descent* heette. Deze parsers zijn compact, snel, en kunnen redelijk goede foutmeldingen produceren. Daar staat tegenover dat de meeste grammatica's niet LL(1) zijn en dus herschreven moeten worden om ze LL(1) te maken.
- LALR(1), dit is de standaard bottom-up parseringstechniek die, bijvoorbeeld, door parsergeneratoren als YACC en BISON gebruikt wordt. Deze parsers zijn groot, redelijke snel, en kunnen matig goede foutmeldingen produceren. De meeste grammatica's passen, soms met enige aanpassingen, binnen deze klasse.
- GLR, is een zeer algemene parseringstechniek die, in tegenstelling tot de voorgaande technieken, **alle** context-vrije grammatica's aankan. Hierdoor is het mogelijk om grammatica's modulair op te bouwen en voor verschillende combinaties van modulen verschillende parsers te genereren. Dit blijkt onder andere van belang te zijn bij systeemrenovatie waar voor vele dialecten van bestaande programmeertalen (b.v. COBOL) parsers gegenereerd moeten worden. De GLR techniek wordt gebruikt in de ASF+SDF Meta-Environment.



Figuur 4.4: Codegeneratorgenerator

4.3 Codegeneratorgeneratoren

In Sectie 3.2.6 hebben we de taken van een codegenerator besproken: allocatie van register, instructieselectie, en lokale optimalisaties.

Een *codegeneratorgenerator* heeft tot doel om voor een specifieke processor, waarvan de architectuur op de een of andere wijze formeel gekarakteriseerd moet zijn, een codegenerator te genereren.

De meest gangbare methodes gaan uit van een formele architectuurbeschrijving die allerlei aspecten van een processor vastlegt, zoals globaal geheugen, registers, conditievlaggen, addresseringsmodes, en uiteraard de machine-instructies. Van de machine-instructies worden diverse aspekten gedefinieerd:

- hun pre- en postcondities, d.w.z. de voorwaarden waaronder ze van toepassing zijn en de effecten die ze hebben op de globale toestand van de processor;
- de kosten van uitvoering uitgedrukt in het aantal benodigde klokcycli;
- de lengte van hun binaire codering;

- het *effect* van elke instructie uitgedrukt als *boomherschrijfregel*, d.w.z. het toepassen van de betreffende instructie leidt tot een vereenvoudiging van het te vertalen programma zoals aangegeven door de herschrijfregel. Een `ADD` instructie kan, bijvoorbeeld, een subboom overdekken waarvan de operator een +-operator is.

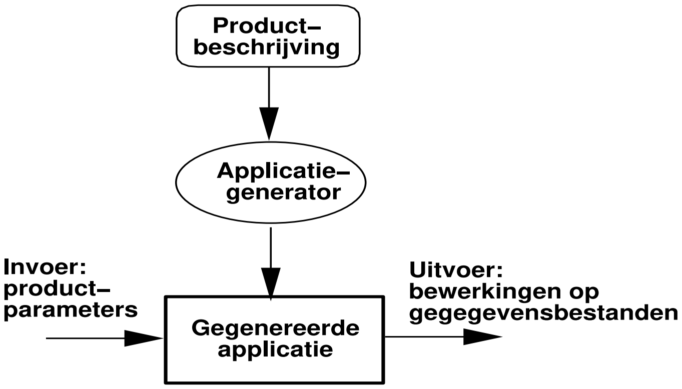
Het genereren van code komt nu neer op het kiezen van een rij machine-instructies met de volgende eigenschappen:

- de boomherschrijfregels die bij de gekozen instructies horen overdekken de gegeven tussencode (bij voorkeur ook een boom) volledig;
- de kosten in ruimte en tijd van de gekozen rij machine-instructies is minimaal.

Codegeneratorgeneratoren worden toegepast voor het construeren van codegenerators voor nieuwe generaties algemene processors en voor zeer gespecialiseerde embedded processors. In het eerste geval is de ontwikkelsnelheid van de nieuwe codegenerator doorslaggevend, in het tweede geval de reductie van de totale ontwikkelingskosten.

4.4 Applicatiegeneratoren

Programmageneratoren kunnen ook dienen voor het genereren van applicaties in velerlei andere toepassingsdomeinen. Figuur 4.5 laat de struktuur zien van een typische applicatiegenerator in het bancaire domein. Doel is om op basis van compacte *productbeschrijvingen* een applicatie te genereren die producten volgens de beschrijving afhandelt. De applicatiegenerator genereert de gewenste applicatie, die op zijn beurt als invoer bepaalde product parameters neemt (b.v. de duur van een lening) en als uitvoer de gewenste bewerkingen op de bancaire gegevensbestanden uitvoert. Merk op dat de taal waarin de productbescrhijvingen geschreven zijn vaak een domein-specifieke taal zal zijn.



Figuur 4.5: Applicatiegenerator

5 Programmeeromgevingen

Een compiler of interpretator is essentieel voor de uitvoering van een programma. Welke ondersteuning is er daarnaast voor de programmeur bij het schrijven van programma's? Hulpmiddelen voor systeemanalyse en systeemontwerp worden besproken in het hoofdstuk SOFTWARE ENGINEERING. Hier beperken we ons tot hulpmiddelen die een rol spelen tijdens het coderen en testen van programma's en een geïntegreerde programmeeromgeving bieden.

Voorbeelden van programmeeromgevingen zijn Visual C++, de Java Development Kit (JDK), en de MicroFocus COBOL Workbench.

5.1 Editors

De centrale activiteit tijdens het programmeren is *editen*: het creëren en veranderen van de brontekst van programma's. Hiervoor bestaat ondersteuning op verschillende niveau's:

- *Pure teksteditors*, zoals in gebruik om willekeurige documenten te editen, ondersteunen alleen elementaire bewerkingen op de brontekst (voeg in, verwijder, zoek).

- *Taalgevoelige* editors bevatten rudimentaire kennis van de programmeertaal en zijn behulpzaam bij het laten zien van bij elkaar passende haakjes, inspringen bij het begin van een volgend statement, en dergelijke. Ze kunnen echter geen syntaxfouten ontdekken.
- *Syntax-gestuurde editors* bevatten volledige kennis van de grammatica van de programmeertaal en waarschuwen zodra er een syntactische incorrecte constructie is ingevoerd. Bovendien helpen ze bij het invoeren van skeletten van volledige taalconstructies die alleen nog maar door de gebruiker verder ingevuld hoeven worden.

Bij een goede integratie van de editor in de programmeeromgeving zijn meldingen van compilers, debuggers en andere tools gekoppeld aan de bijbehorende regel in de brontekst en worden direct zichtbaar gemaakt in de editor. Bovendien is het mogelijk om vanuit de editor de compiler aan te roepen om modules waarin fouten hersteld zijn opnieuw te compileren.

5.2 Test tools

Het testen van programma's is een belangrijk deelgebied van software engineering dat voor een groot deel methodologisch van aard is (*design reviews*, *code inspecties*) en maar voor een klein deel door tools ondersteund wordt (prototyping, unit test, integratie test). Het blijkt dat een goede methodologische aanpak van het ontwikkelproces van groter belang is voor het opleveren van foutvrije programma's dan het gebruik van specieke test tools. De combinatie van methodologie en tools levert software op die tijdens het intern testen nog 10 tot 20 fouten per 1000 regels code kan bevatten en opgeleverde produkten die in het beste geval 0.1–0.5 fouten per 1000 regels code bevatten. In het hoofdstuk TESTEN wordt de methodologie van testen behandeld, hier beperken we ons tot enkele specieke test tools.

Test data generatoren produceren invoerdata met als doel zoveel mogelijk delen van het te testen programma uit te voeren door onverwachte of onwaarschijnlijke combinaties van invoerwaarden aan te bieden. Test data generatoren maken gebruik van control flow analyse en data flow analyse.

Overdekkingsmeters (Engels: *Coverage monitors*) meten welke delen van de code uitgevoerd zijn als resultaat van het uitvoeren van een verzameling testgevallen. Dit is typisch slechts 55% van de code, maar kan door gebruik van een coverage monitor aanzienlijk verhoogd worden. Coverage monitors werken meestal in drie fasen. Allereerst worden aan het load module van het te monitoren programma instructies toegevoegd die in een apart monitor-bestand precies aangeven welke statements uitgevoerd worden. Vervolgens worden alle testgevallen uitgevoerd met het aldus geïnstrumenteerde programma. Tenslotte worden de gegevens in het monitor-bestand gebruikt om de overdekkingsgraad te berekenen en te rapporteren.

Regressietesters hebben tot doel om het testproces zodanig te automatiseren dat na een wijziging van de code, alle voorgaande testgevallen (en hun bewaarde uitvoer) opnieuw uitgevoerd worden en oude en nieuwe uitvoer met elkaar vergeleken worden.

Test tools hebben tot doel om fouten in een programma aan te tonen. Hoe stellen we de oorzaak van de fout vast en hoe verbeteren we deze?

5.3 Debuggers

De werking van de eerste relaiscomputers kon ernstig verstoord worden doordat een insekt (Engels: *bug*) in de electromechanische circuits was doorgedrongen. Sindsdien heet het opsporen van fouten in computers (zowel in apparatuur als in programmatuur) *debugging*, in het verleden door taalpuristen ook wel aangeduid met "ontluizen".

Zodra een fout in de code vermoed wordt, is de klassieke werkwijze om print statements aan de code toe te voegen om zo de vorderingen van de berekingen te kunnen volgen. Het is echter veel effectiever om hiervoor een *hoog-niveau symbolische debugger* te gebruiken.

Een debugger biedt allereerst aan de gebruiker de mogelijkheid om *breekpunten* (Engels: *break point*) te definieren die corresponderen met specifieke regels in de brontekst of met waardeveranderingen van een specifieke variabele. Zodra tijdens uitvoering van het programma een breekpunt wordt bereikt, wordt de uitvoering onderbroken.

Tijdens een onderbreking kan de programmatoestand in detail geïnspekteerd worden: niet alleen de waarden van variabelen, pointers en dynamisch gealloceerde datastructuren kunnen in symbolische vorm bekeken worden maar ook alle procedureaanroepen die op dat moment actief zijn. Sommige debuggers bieden de mogelijkheid om de programmatoestand tijdens een breekpunt te wijzigen. Bovendien kunnen tijdens een onderbreking ook nieuwe breekpunten gedefinieerd worden.

Na een onderbreking kan de uitvoering van het programma op verschillende manieren voortgezet worden: op volle snelheid tot het volgende breekpunt, statement-voor-statement vooruit, en soms zelfs statement-voor-statement achteruit.

Uiteraard vormen debuggers geen alternatief voor goed programma-ontwerp, ze zijn echter een effectief hulpmiddel bij het opsporen van fouten.

5.4 Prestatiemetingen

Prestatieproblemen van programmatuur zijn het beste te achterhalen door hulpmiddelen te gebruiken die het *executieprofiel* van een programma tijdens uitvoering meten: welke statements zijn hoevaak uitgevoerd en hoeveel tijd kostte dat? Het executieprofiel dient enerzijds om prestatieproblemen op te sporen maar anderzijds vooral ook om vast te stellen waar verdere optimalisatie geen zin meer heeft.

Bepaling van het executieprofiel gaat meestal in drie fasen. Allereerst worden aan het load module van het te profileren programma instructies toegevoegd die in een apart profiel-bestand precies aangeven hoe vaak elk statement is uitgevoerd en hoeveel tijd dat kostte. Vervolgens wordt het aldus geïnstrumenteerde programma uitgevoerd. Tenslotte worden de gegevens in het profiel-bestand gebruikt om de executieprofielen te te berekenen en te rapporteren.

5.5 Configuratie- en versiebeheer

Grote programmatuurprojekten zijn dynamisch van aard en vergen systematische methode's om veranderingen in eisen, ontwerp, implementatie, platform, documentatie en testgevallen te beheren.

Tools voor *configuratiebeheer* houden zich bezig met het identificeren en beheren van de softwarecomponenten waaruit een systeem bestaat. Voor verschillende platforms, kunnen verschillende componenten nodig zijn of specifieke aanpassingen vereisen. Een voorbeeld is AUTOCONF.

Tools voor *versiebeheer* houden zich bezig met het registreren van veranderingen die aan de softwarecomponenten aangebracht worden. Hierdoor is het mogelijk op elk gewenst moment oudere systeemversies te reconstrueren. Voorbeelden zijn *Source Code Control System (SCCS)*, *Revision Control System (RCS)* en *Concurrent Versioning System (CVS)*.

Tools voor *systeemconstructie* zijn gebaseerd op een beschrijving van alle systeemcomponenten en hun onderlinge samenhang en genereert op basis daarvan de noodzakelijke commmando's (compileren, linken, en dergelijke) om een nieuwe, gecompileerde, versie van het systeem te bouwen. Dankzij separate compilatie (Sectie 3.2.7) hoeven alleen die systeemdelen opnieuw gecompileerd te worden die gewijzigd zijn. Een bekend voorbeel vane en tool voor systeemconstructie is MAKE.

6 Literatuur

De volgende boeken behandelen concepten van programmeertalen:

- H. E. Bal en Dick Grune, *Programming Language Essentials*. Addison-Wesley, 1994.
- B. Meyer, *Object-Oriented Software Construction*. Second edition, Prentice Hall, 1997.

- R. Sethi, *Programming Languages; Concepts and Constructs*. Second edition, 1996.
- D. A. Watt, *Programming Language Concepts and Paradigms*. Prentice Hall, 1990.

De volgende boeken gaan specifiek over één programmeertaal:

- W. B. C. Ebbinkhuijsen. *COBOL*, Vijfde druk gebaseerd op ISO-COBOL 1985. Samsom, 1990.
- D. Flanagan, *Java in a Nutshell, A Desktop Quick Reference*, Second edition, O'Reilly, 1997.
- B. W. Kernighan en D. M. Ritchie, *The C programming language*, Second edition, Prentice Hall, 1988.
- J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
- B. Stroustrup, *The C++ programming language*, Second edition, Addison-Wesley, 1991.
- S. Thompson, *Haskell, The Craft of Functional Programming*, Addison-Wesley, 1996.

Onderstaande boeken behandelen compilerbouw:

- A. Aho, R. Sethi, en J. Ullman, *Compilers — Principles, Techniques, Tools*, Addison-Wesley, 1986.
- R. Wilhelm en D. Maurer, *Compiler Design*, Addison-Wesley, 1995.

Tenslotte geven we nog enkele nuttige verwijzingen op Internet:

- European Association for Programming Languages and Systems (EAPLS): `www.cwi.nl/EAPLS`
- ACM SIGPLAN: `www.acm.org/sigplan/`
- International standardization subcommittee for programming languages: `wwwold.dkuug.dk/JTC1/SC22/`

- Catalog of Free Compilers and Interpreters: `www.idiom.com/free-compilers/`
- Programming Language Research: `www.cs.cmu.edu/~mleone/language-research.html`

A.v.D./P.K.