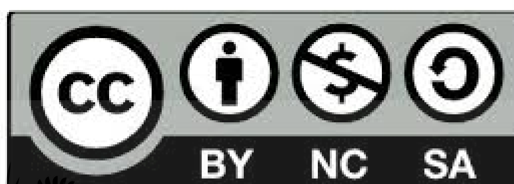
COPYRIGHT AND CITATION CONSIDERATIONS FOR THIS THESIS/ DISSERTATION



- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- NonCommercial — You may not use the material for commercial purposes.

- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

How to cite this thesis

Surname, Initial(s). (2012) Title of the thesis or dissertation. PhD. (Chemistry)/ M.Sc. (Physics)/ M.A. (Philosophy)/M.Com. (Finance) etc. [Unpublished]: University of Johannesburg. Retrieved from: https://ujdigispace.uj.ac.za (Accessed: Date).

WR10
OLIV

DIE ONDERSTEUNING VAN ABSTRAKTE DATATIPES EN TOESTELLE IN 'N PROGRAMMEERTAAL

deur

MARTIN STEPHANUS OLIVIER

VERHANDELING

voorgelê ter vervulling van die vereistes vir die graad

MAGISTER IN DIE NATUURWETENSKAPPE

in

UNIVERSITY OF JOHANNESBURG

REKENAARWETENSKAP

in die

FAKULTEIT NATUURWETENSKAPPE

aan die

RANDSE AFRIKAANSE UNIVERSITEIT
STUDIELEIER: PROF. S.H. VON SOLMS
MEDE-STUDIELEIER: PROF. R.J. VAN DEN HEEVER
MEDE-STUDIELEIER: PROF. T.H.C. SMITH

NOVEMBER 1985

Hiermee betuig ek my dank aan die WNNR
vir die finansiële steun wat ek
ontvang het vir hierdie studie.

SOLI DEO GLORIA

Inhoud.

1 Oorsig.

2 Inleiding.

3 Objekgeoriënteerde programmering.

4 Abstrakte datatipes.

5 Abstrakte datatipes in programmeertale.

6 Die wiskunde van abstrakte datatipes.

7 Data-abstraksie in programmeertale : kort gevallestudies.

8 Abstrakte toestelle in 'n programmeertaal.

9 Rekordgebaseerde toestelle.

10 Hoër-vlak toestelle : 'n voorbeeld.

11 Implementering van die vertaler.

12 Doeltreffende rekenaargebruik.

A Die programmeertaal PT.

B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

THE SUPPORT OF ABSTRACT DATA TYPES AND DEVICES IN A PROGRAMMING LANGUAGE - SUMMARY.

The peripherals of a computer system constitutes an important part of the hardware of the system. It is easy to see that there are similarities between these devices and an instance of an abstract data type - both act as encapsulated entities and are used through a well-defined interface. Objects such as RAM-disks, blur the distinction between devices and abstract data types. The concept "abstract device" is therefore defined to refer to any such encapsulated object which may only be used through its interface - the object may consist of hardware, software or a combination of both.

Objects and object oriented programming, as well as abstract data types and their support in current programming languages are examined for suitability to support abstract devices in a high-level programming language - it is argued that the usual programming language support for abstract data types provides the better solution for abstract device support.

Normal support for abstract data types are, however, not always sufficient to support abstract devices - the most important additional facility required, is the ability to support "record-based" abstract devices; a record-based abstract device is an abstract device such as a database where the information is organised in records (with a special meaning attached to certain fields of the record, e.g. key fields) and where the actual record differs from instance to instance. Since the definition of abstract devices includes the notion of abstract data types, it follows that a facility to support abstract devices will implicitly be able to support abstract data types. It is there-

fore necessary to extend the facilities for supporting abstract data types to support abstract devices as well.

The mathematical specification of both syntax and semantics of an abstract data type, has become an important field of study. It is consequently important to extend this aspect of abstract data types to abstract devices - the necessary additions are discussed and the mathematical specification of abstract devices is illustrated with a number of examples.

A programming language has been developed to demonstrate the facilities which may be provided to support abstract devices. The specification of abstract devices (in this programming language) is also discussed and illustrated with examples. A comprehensive example of an "intelligent" abstract device is also given.

UNIVERSITY
OF
JOHANNESBURG

A compiler cannot treat every instance of an abstract device like a conventional variable (although a conventional variable is, by definition, an instance of an abstract device). Since this area is the only where a compiler for a language which supports abstract devices, differs from another compiler, the internal representation of abstract devices is covered in detail.

A full specification of the above mentioned programming language is given in an appendix.

1 Oorsig.

Hierdie verhandeling ondersoek die ondersteuning wat programmeertale vir toestelle behoort te bied.

Die begrip "toestel" word eerstens uitgebrei na "abstrakte toestel", wat 'n breër veld objekte insluit as wat die geval met "toestel" is. Daarna word vasgestel watter fasiliteite in 'n programmeertaal voorsien kan word ter ondersteuning van abstrakte toestelle. Laastens word na 'n spesifieke programmeertaal (wat spesiaal vir die doel ontwikkel is) gekyk, om die ondersteuning van abstrakte toestelle in 'n programmeertaal volledig te illustreer.

Die onderwerpe wat in die onderskeie hoofstukke gedek word, is die volgende :

Hoofstuk 2 : Die begrip "abstrakte toestel" word ontwikkel en gedefinieer.

Hoofstuk 3 : Objekgeoriënteerde programmering word ondersoek om vas te stel tot watter mate dit abstrakte toestelle ondersteun.

Hoofstuk 4 : Abstrakte datatipes word ondersoek om vas te stel tot watter mate dit die implementering van abstrakte toestelle moontlik maak.

Hoofstuk 5 : Daar word kortliks gekyk na tipiese fasiliteite wat ter ondersteuning van abstrakte datatipes in programmeertale aangetref word - die fasiliteite ter ondersteuning van abstrakte toestelle word later hierop gebaseer.

UNIVERSITY OF JOHANNESBURG

Hoofstuk 6 : Eerstens word na die formele wiskundige beskrywing van abstrakte datatipes gekyk, waarna aangetoon word dat abstrakte toestelle op 'n soortgelyke wyse beskryf kan word.

Hoofstuk 7 : Hierdie hoofstuk kyk kortliks na die notasie wat in 'n paar bestaande programmeertale gebruik word om abstrakte datatipes te definieer en te gebruik - die notasie wat vir abstrakte toestelle gebruik gaan word, is hierop gebaseer.

Hoofstuk 8 : Die fasiliteite wat 'n programmeertaal behoort te voorsien om abstrakte toestelle te ondersteun, word in hierdie hoofstuk gedek.

Hoofstuk 9 : Hier word een (belangrike) aspek van abstrakte toestelle - die sogenaamde rekordgebaseerde toestelle - volledig bespreek.

Hoofstuk 10 : Hierdie hoofstuk illustreer die krag van abstrakte toestelle deur middel van 'n voorbeeld.

Hoofstuk 11 : Die interne voorstelling van abstrakte toestelle en die wyse waarop hierdie voorstelling (op 'n lae vlak) hanteer kan word, word toegelig.

Hoofstuk 12 : Hier word geredeneer dat die oorhoofse koste verbonde aan abstrakte toestelle nie te hoog is nie.

Bylaag A : Hierdie bylaag bevat die spesifikasie van die programmeertaal waarna in die vorige afdelings verwys is.

UNIVERSITY OF JOHANNESBURG

Hoofstuk 2.

2 Inleiding.
  2.1 Toestelle.
  2.2 Veralgemening van die begrip "toestel".
  2.3 Abstrakte toestelle.
  2.4 Die toestelgeoriënteerde programmeringsmetodologie.

Inhoud.

1 Oorsig.
2 Inleiding.
3 Objekgeoriënteerde programmering.
4 Abstrakte datatipes.
5 Abstrakte datatipes in programmeertale.
6 Die wiskunde van abstrakte datatipes.
7 Data-abstraksie in programmeertale : kort gevallestudies.
8 Abstrakte toestelle in 'n programmeertaal.
9 Rekordgebaseerde toestelle.
10 Hoër-vlak toestelle : 'n voorbeeld.
11 Implementering van die vertaler.
12 Doeltreffende rekenaargebruik.
A Die programmeertaal PT.
B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

2 Inleiding.

2.1 Toestelle.

Die apparatuur van 'n rekenaarstelsel bestaan, behalwe vir die rekenaar self (sentrale verwerker, geheue en beheereenheid), uit 'n verskeidenheid randapparatuur. Voorbeelde van sulke toestelle is drukkers, skyfaandrywers, kaartlesers, terminale, ensovoorts. Met elkeen van hierdie toestelle word gewoonlik 'n drywerprogram geassosieer, met behulp waarvan die toestel beheer word.

Die drywerprogram dien as koppelvlak tussen die (fisiese) toestel en ander programmatuur. Die programmatuurkant van die koppelvlak bestaan uit 'n aantal roetines (prosedures en funksies) wat geroep kan word om die toestel te gebruik. 'n Tipiese voorbeeld van so 'n koppelvlak, is 'n drywerprogram wat 'n skyfaandrywer beheer. Gewoonlik voorsien so 'n drywerprogram roetines om :

- Die data wat op 'n gespesifiseerde sektor van 'n spesifieke baan van die skyf voorkom, te lees;
- Data op 'n gespesifiseerde sektor van 'n spesifieke baan van die skyf te skryf; en
- 'n Gegewe baan van die skyf te formatteer.

Aan die toestelkant van die drywerprogram kommunikeer hierdie roetines met die (fisiese) toestel om dit te beheer; hanteer data-oordrag na en vanaf die toestel en verkry statusinligting vanaf die toestel. Raadpleeg [Mot81] en [Wes80] vir besonderhede.

So 'n toestel, saam met sy drywerprogram, word dan as 'n eenheid gesien : die enigste manier waarop ander programmatuur so 'n

eenheid (behoorlik) kan gebruik, is deur toepaslike roetines in die eenheid se koppelvlak te roep. Hierdie opset hou verskeie voordele in :

1) Programme wat die toestel wil gebruik hoef niks van die interne werking van die toestel te weet nie - dit gebruik die toestel via die logiese koppelvlak wat deur die drywerprogram voorsien word. So werk die voorafgaande toestel (volgens sy koppelvlak) in terme van grepe data, terwyl die werklike toestel in terme van magnetiese velde op 'n magnetiseerbare oppervlak werk.

2) Al die toestel-afhanklike kode is op een plek gekonsentreer - as dit dus nodig is om 'n toestel met 'n ander toestel te vervang wat dieselfde funksie as die oorspronklike toestel het, maar wat tog effens anders beheer word, is dit slegs nodig om die drywerprogram te verander.

3) As die programmatuurkoppelvlak van 'n drywerprogram geskryf word om soos diè van 'n bestaande toestel te lyk, kan die nuwe toestel deur programme wat reeds die bestaande toestel gebruik, benut word - al wat nodig is, is om elke roep na 'n drywerroetine van die bestaande toestel met 'n roep na die ooreenkomstige roetine van die nuwe toestel te vervang, waarna die ou program outomaties die nuwe toestel sal gebruik.

Die inligtingverstekingskonsep is aan Parnas [Par72] te danke. Die metode wat in die voorafgaande bespreking gebruik is om toestelle te beskryf, maak duidelik van inligtingversteking gebruik. Die voordele wat deur Parnas genoem is, wanneer die tegniek vir die ontwikkeling van programmatuurmodules gebruik word, geld ook wanneer dit by die beskrywing van toestelle

UNIVERSITY OF JOHANNESBURG

gebruik word.

Die riglyne vir die verdeling van 'n program in modules soos deur Constantine en Myers gegee (sien [Mye73] en [Jen79] pp 174-179), kan oor die algemeen nog gebruik word wanneer sommige van die "modules" toestelle is - daar behoort nog steeds gestreef te word na sterk kohesie en 'n lae mate van koppeling tussen modules, eenvoudige ontwerp, modules wat so klein as moontlik is, fout-hantering deur die modules self en 'n abstrakte ontwerp. Die oorblywende twee riglyne, naamlik dat die gedrag van modules voorspelbaar moet wees en dat die besluite wat in 'n module geneem word net die modules wat deur hierdie module gebruik word, mag beïnvloed, moet effens aangepas word - 'n toestel is dikwels in een van 'n aantal moontlike toestande en die gedrag van die toestel hang van die toestand af; die waarde wat vanaf 'n sektor van 'n skyf gelees word, hang byvoorbeeld af van wat voorheen op daardie sektor geskryf is. 'n Module sal dus nou "voorspelbaar" wees as die module, wanneer dit in 'n vaste toestand is, identiese resultate lewer as dit met identiese invoere geroep word. Verder behoort daar gekontroleer te word dat 'n module nie die gedrag van 'n onafhanklike module beïnvloed deur die toestand van 'n toestelmodule, wat deur die ander module gebruik gaan word, te verander nie.

2.2 Veralgemening van die begrip "toestel".

In die vorige afdeling is aanvaar dat 'n toestel 'n fisiese eenheid is. Die nuwe eenheid wat gevorm word deur die samevoeging van 'n toestel en 'n drywerprogram (of koppelvlak), kan as 'n nuwe toestel (op 'n hoër-vlak) gesien word, wat weer op sy beurt van 'n koppelvlak voorsien kan word.

As 'n program byvoorbeeld data semipermanent moet bêre, is 'n skyf die ideale toestel daarvoor. Die koppelvlak wat in die vorige paragraaf beskryf is, is egter baie ongerieflik vir hierdie doel. Dit is beter om 'n nuwe koppelvlak bo die bestaande koppelvlak te voorsien, wat meer toepaslike roetines bevat. Ons kan die nuwe toestel 'n "Lêer" noem en die volgende roetines in die koppelvlak voorsien :

- *Leer_OpenLees*, wat die lêer gereed maak om vanaf gelees te word;
- *Leer_OpenSkryf*, wat die lêer gereed maak om na geskryf te word;
- *Leer_Lees*, wat data vanaf die lêer lees; en
- *Leer_Skryf*, wat data op die lêer skryf.

Nie een van hierdie lêer-bewerkings verwys na spesifieke sektore op die skyf nie - dit word alles (deursigtig vir die gebruiker) deur die lêer-drywerprogram hanteer. Die lêer-toestel is dus 'n hoër-vlak toestel as die skyf-toestel in die sin dat dit die bewerkings voorsien wat die program benodig, in teenstelling met die meer primitiewe bewerkings van die skyf-toestel.

'n Interessante voorbeeld van 'n "geskepte" toestel kom voor waar programmatuur gebruik word om meer as een fisiese toestel in een logiese toestel te kombineer. Sê ons wil byvoorbeeld 'n konsole, wat uit beide 'n terminaal en 'n drukker bestaan, beheer. Alles wat op die terminaal se skerm vertoon word, moet ook deur die drukker gedruk word vir latere verwysing - die konsole-toestel is dus in werklikheid 'n enkele toestel en die twee onderliggende toestelle mag nie afsonderlik gebruik word nie. Die koppelvlak van die konsole-toestel kan die roetines *Konsole_LeesKar* en *Konsole_SkryfKar* bevat, waar *Konsole_SkryfKar* 'n karakter op beide die terminaal se skerm (met behulp van, sê,

Terminaal_SkryfKar) en op die papier in die drukker (met behulp van, sê, *Drukker_SkryfKar*) sal skryf. Soortgelyk sal *Konsole_LeesKar* 'n karakter vanaf die terminaal lees en sorg dat dit op beide die terminaal en op die drukker ge-eggo word - weer eens met behulp van roetines wat deur die onderskeie koppelvlakke van die terminaal- en drukker-toestelle voorsien word. Programme wat die konsole-toestel gebruik, gebruik dit as 'n enkele (logiese) toestel en is totaal onbewus van die feit dat daar meer as een fisiese toestel betrokke is.

Die aantal koppelvlakke wat 'n gebruikersprogram van die fisiese toestel skei is ook nie van belang nie. 'n Geval wat dit goed illustreer kom by seriële datakommunikasie voor : sommige rekenaars beskik oor toegewyde apparatuur (gewoonlik op 'n enkele vlokkie, sien byvoorbeeld [Mot78]) wat 'n greep neem en dit dan in serie versend. Dit voeg self die nodige raam- en pariteitsbisse in die stroom bisse. 'n Koppelvlak vir hierdie toestel kan, onder andere, die roetines *Serie_LeesKar* en *Serie_SkryfKar* voorsien - *Serie_SkryfKar* kan 'n karakter in 'n buffer plaas vanwaar dit versend kan word sodra al die karakters wat reeds in die buffer was, versend is. Karakters wat deur die apparatuur ontvang word, word in 'n buffer geplaas, vanwaar *Serie_LeesKar* dit kan lees.

As die rekenaar nie oor hierdie "intelligente" apparatuur beskik nie, maar wel beheer oor die afsonderlike lyne van die seriepoort het, kan 'n RS232-toestel met behulp van programmatuur geskep word, wat dan die funksie vervul wat in die voorafgaande beskrywing deur apparatuur vervul is. Die serie-toestel wat in die vorige paragraaf beskryf is, kan nou bo-op hierdie RS232-toestel gebou word.

In beide gevalle lyk die serie-toestel dieselfde vir programme

wat dit gebruik - logies is dit identiese toestelle.

Uit al die genoemde voorbeelde is dit duidelik dat dit moontlik is (en inderdaad wenslik is) om die toestelle wat in 'n program benodig word, te skep. Hierdie nuwe toestelle word uit een of meer beskikbare toestelle gekonstrueer. In al die gevalle is die nuwe toestel 'n logiese eenheid, waarvan die interne apparatuur- / programmatuursamestelling versteek is vir die programme wat dit gebruik.

2.3 Abstrakte toestelle.

Tot dusver is deurentyd aanvaar dat daar êrens in elke "toestel" 'n fisiese toestel voorkom. Dit is egter ook moontlik om 'n toestel te skep wat slegs uit programmatuur bestaan.

'n Voorbeeld van so 'n toestel wat, behalwe vir die sentrale verwerker en interne geheue, slegs uit programmatuur bestaan, is die sogenaamde LSG-skyf. 'n LSG-skyf is 'n program wat 'n gedeelte van 'n rekenaar se interne geheue soos 'n skyf laat lyk. Hierdie programmatuur voorsien 'n "koppelvlak" wat roetines soortgelyk aan die van 'n gewone skyf voorsien. Uiterlik werk die LSG-skyf presies soos 'n werklike skyf sodat programme wat skywe gebruik nie kan onderskei tussen die LSG-skyf en die ware Jakob nie.

'n Toestel wat nie van 'n fisiese toestel gebruik maak nie, hoef nie 'n fisiese toestel te emuleer, soos in die geval van die LSG-skyf, nie. In 'n omgewing waar baie vertalers en saamstellers, byvoorbeeld, gebruik word, kan die bedryfstelsel beheer oor simbooltabelle uitoefen en kan 'n program wat 'n simbooltabel benodig, dit van die bedryfstelsel aanvra net soos dit enige

ander stelselhulpbron sou aanvra. Presies so kan stapels, toue, ensovoorts, as stelselhulpbronne gesien word, wat deur die bedryfstelsel hanteer word.

Hierdie sentrale beheer oor sulke hulpbronne hou verskeie voordele in. Eerstens kan die hulpbron verbeter word met 'n gevolglike verbetering in al die programme wat die hulpbron gebruik. Verder is dit nie nodig dat elke program wat die hulpbron benodig, oor 'n kopie van die objekkode van die hulpbron beskik nie. Skyf- en geheueruimte word dus bespaar as meer as een program die hulpbron gebruik. Die algemene beskikbaarheid van so 'n hulpbron sal ook tot gevolg hê dat programmeerders die bestaande hulpbron gebruik en dus nie nodig het om hulle eie kode daarvoor te ontwikkel nie; dit sal natuurlik tot vinniger programontwikkeling lei. Dit sal ook veroorsaak dat nuwe programme voordeel kan trek uit die (moontlik heelwat) ontwikkelingswerk wat vir die hulpbron gedoen is.

Omdat daar nie fisiese toestelle by hierdie hulpbronne betrokke is nie, kan die bedryfstelsel 'n nuwe "toestel" skep wanneer so 'n toestel aangevra word, en kan die geheueruimte wat deur so 'n "toestel" beslaan is, herbruik word wanneer die toestel nie verder benodig word nie.

Dit sal egter ondoeltreffend wees om die bedryfstelsel beheer te gee oor so 'n toestel wat net deur enkele programme gebruik word. Om hierdie probleem uit te skakel kan 'n fasiliteit voorsien word sodat "privaat toestelle" deur 'n programmeerder geskep kan word wat net in 'n program waarin dit gedefinieer is, beskikbaar is. Die kode van sulke toestelle wat relatief selde benodig word, kan in 'n biblioteek gehou word sodat heelwat van die genoemde voordele behoue kan bly, sonder om die bedryfstelsel verder te belas.

Die term <u>abstrakte</u> <u>toestel</u> sal voortaan gebruik word om te verwys na enige van hierdie toestelle - dit wil sê, 'n abstrakte toestel is enige objek wat slegs deur (ander) programmatuur gebruik kan word deur die roetines wat die objek hiervoor voorsien.

2.4 Die toestelgeoriënteerde programmeringsmetodologie.

Die siening van toestelle, soos dit in die vorige afdelings bespreek is, gee op 'n natuurlike wyse aanleiding tot 'n programmeringsmetodologie wat op die konstruksie van toepaslike toestelle gebaseer is.

Op die hoogste vlak kan 'n nuwe program (of programmatuurstelsel) as 'n toestel gesien word. Nou word bepaal watter toestelle nodig is om hierdie nuwe toestel te konstrueer, waarna vasgestel word watter toestelle benodig word om elk van hierdie toestelle te konstrueer, ensovoorts. Die proses word voortgesit totdat bestaande toestelle gebruik kan word, of die toestelle eenvoudig genoeg is om maklik met behulp van roetines en basiese datastrukture te implementeer. Die nuwe toestelle kan nou van onder na bo geïmplementeer word, tot op die boonste vlak, wat die verlangde program is.

Die metode vereis dus dat die programmeerder vasstel watter abstrakte toestelle die beste by die spesifieke probleem wat hy moet oplos, pas. Dit is in teenstelling met die gewone praktyk waar gepoog word om die probleem direk met die bestaande toestelle en datastrukture op te los.

UNIVERSITY OF JOHANNESBURG

Voorbeeld 2.4.1
As 'n vertaler ontwikkel moet word, kan die volgende toestelle identifiseer word :

1) 'n Simbooltabel, wat bewerkings voorsien om simbole en beskrywings in die simbooltabel te plaas en uit die simbooltabel te onttrek. Die simbooltabel kan van ander toestelle, soos lêers gebruik maak, of kan die simbole in die geheue hou.
2) 'n Leksikale ontleder wat 'n roetine voorsien wat telkens die volgende leksikale simbool wat in die bronprogram voorkom, voorsien. Onderliggend aan die leksikale ontleder sal 'n toestel, soos 'n lêer wees, wat die bronprogram bevat.
3) 'n Objekkodehanteerder, wat 'n roetine voorsien om objekkode na die objekkodehanteerder te stuur. Intern kan die objekkodehanteerder die objekkode na 'n ander toestel, soos 'n lêer, skryf.

UNIVERSITY OF JOHANNESBURG

□

Voorbeeld 2.4.2
Veronderstel 'n inligtingstelsel moet ontwikkel word, wat inligting in 'n databasis bêre en daaruit onttrek. Toestelle wat hier nuttig sal wees, is die databasis self (wat op lêers gebou kan wees), die terminaal en die drukker.

'n Kragtiger toestel kan egter tussen die genoemde toestelle en die inligtingstelsel geplaas word - as 'n toestel ontwikkel word wat die terminaal en die databasis in 'n enkele toestel kombineer en wat dan bewerkings *Skep*, *Bywerk*, *Skrap* en *Vertoon* voorsien, is dit baie maklik om 'n inligtingstelsel met behulp van hierdie toestel te ontwikkel. As die toestel boonop in 'n toestel-

biblioteek (of as 'n stelselhulpbron) beskikbaar is, kan nuwe inligtingstelsels feitlik moeiteloos ontwikkel word.

Aangesien inligtingstelsels algemeen (veral in die kommersiële wêreld) voorkom, illustreer hierdie voorbeeld die krag en waarde van "meer intelligente" toestelle. 'n Volledige voorbeeld word in hoofstuk 10 gegee.

□

Voordele van die toestelgeoriënteerde programmeringsmetodologie en van die beskikbaarheid van standaard toestelle, hetsy as stelselhulpbronne of in toestelbiblioteke, is die volgende :

1) Die beskikbaarheid van standaard toestelle sal lei tot vinniger programmatuurontwikkeling omdat dit nie vir die programmeerder nodig is om sy eie kode te ontwikkel om die werk gedoen te kry wat deur die toestel gedoen word nie.
2) Die gebruik van standaard toestelle veroorsaak groter eenvormigheid in programkode, wat onderhoud van programme vergemaklik.
3) As daar 'n hoër-vlak toestel vir die terminaal beskikbaar is, wat spesifieke betekenisse aan toetse van die toetsbord heg (byvoorbeeld beweeg na volgende veld, beweeg na vorige veld, aanvaar data, ignoreer data, ensovoorts), sal programme wat hierdie toestel gebruik, eenvormig lyk en dit sal dus makliker wees om te leer om te dit gebruik, met 'n gevolglike vermindering van opleidingskoste.
4) As 'n standaard toestel verbeter word, verbeter al die programme wat die toestel gebruik.
5) Die afskerming van besonderhede oor die interne werking van die toestel veroorsaak dat die werking van 'n toestel maklik verander of verbeter kan word, of met 'n ander fisiese

toestel vervang kan word sonder dat dit nodig is om aan programme wat die toestel gebruik, te verander.

6) Die metodologie vereis dat benodigde toestelle geïdentifiseer moet word - dit is dus moontlik om toestelle wat gereeld benodig word, onder die beheer van die bedryfstelsel te plaas met die gepaardgaande voordele.
7) Koppelvlakke (of drywerprogramme) van toestelle wat baie gereeld gebruik word, kan in mikrokode geskryf word, wat die doeltreffendheid van baie programme sal verhoog. Dit is ook moontlik om toestelle soos, byvoorbeeld, stapels, toue en simbooltabelle in apparatuur vas te lê om hierdie toestelle meer koste-doeltreffend te maak.
8) As die koppelvlakke van twee toestelle soortgelyk is, is dit maklik om die een toestel met 'n ander te vervang - weer eens sonder dat dit nodig is om aan die programme wat dit gebruik te verander.

Voor al hierdie voordele egter gerealiseer kan word, is daar sekere fasiliteite wat voorsien moet sodat die toestelle (gerieflik) gebruik kan word. In die besonder word 'n notasie benodig waarin so 'n toestel in 'n programmeertaal beskryf kan word. Van die vereistes wat aan sulke toestelle gestel word is die volgende :

1) Algemeenheid - Die laer-vlak toestelle werk gerieflik in terme van karakters of grepe. Die hoër-vlak toestelle, soos byvoorbeeld databasisse, werk eerder in terme van groter datastrukture - oor die algemeen rekords - wat van toepassing tot toepassing verskil. Dit moet dus moontlik wees om 'n nuwe toestel te skep sonder om dit aan 'n spesifieke datastruktuur te koppel. So kan 'n lêer-toestel, byvoorbeeld, geskep word en programme wat dit wil gebruik kan dan spesifiseer dat 'n lêer van tipe T1 benodig

word, terwyl 'n ander program dieselfde lêer-toestel gebruik, maar hierdie keer as 'n lêer van tipe T2. Dit moet ook moontlik wees om 'n geparametriseerde toestel te definieer - as 'n tou van 'n vaste maksimum lengte byvoorbeeld voorsien word, moet hierdie maksimum lengte 'n parameter wees, wat deur die program wat die tou benodig, gespesifiseer kan word.

2) Afskerming tussen die interne werking van 'n toestel en die programme wat dit gebruik, moet afgedwing word deur die betrokke programmeertaal, anders kan heelwat van die voordele nie bereik word nie. As sommige programme van kennis van die interne werking van 'n toestel gebruik maak, kan die toestel nie sondermeer vervang word met, of gewysig word na 'n soortgelyke toestel nie.

3) Oorlading van roetines in toestelle se koppelvlakke moet verkieslik moontlik wees, met ander woorde, meer as een toestel moet dieselfde koppelvlakroetine voorsien, en dan moet die toestel se naam gespesifiseer word wanneer die roetine gebruik word. Hierdie vermoë sal die verwisseling van 'n toestel met 'n ander soortgelyke toestel vergemaklik. Daar behoort outomaties getoets te word dat sulke roetines met dieselfde naam, se voorkoms ook verder ooreenkom sodat dit slegs nodig is om die toestelnaam te verander as 'n ander toestel gebruik moet word. Hierdie ooreenkoms sal ook tot gevolg hê dat algemene prosedures geskryf kan word : as beide die drukker en die terminaal byvoorbeeld *SkryfKar*-roetines voorsien, kan 'n *SkryfLyn*-prosedure geskryf word, wat van *SkryfKar* gebruik maak, maar nie spesifiseer of dit vir die drukker of terminaal bedoel is nie. Die *SkryfLyn*-prosedure kan dan vir beide die drukker en terminaal gebruik word, asook vir enige ander toestel wat 'n *SkryfKar*-roetine

voorsien.

4) Dit moet moontlik wees om 'n klas van toestelle te definieer. As 'n program byvoorbeeld meer as een stapel benodig, moet dit nie nodig wees om telkens 'n stapel te definieer nie - dit moet moontlik wees om die klas "Stapel" te definieer en dan telkens wanneer 'n stapel benodig word net te spesifiseer dat die nuwe toestel (ook) van die klas "Stapel" is.

In die volgende afdelings gaan na twee tegnieke gekyk word wat (tot 'n groot mate) geskik is vir ondersteuning van die beskryfde toestelgeoriënteerde programmeringsmetodologie - eerstens word na objekgeoriënteerde programmering gekyk, waarna na data-abstraksie gekyk sal word soos dit tans in programmeertale ondersteun word.

Hoofstuk 3.

3 Objekgeoriënteerde programmering.

3.1 Objekgeoriënteerde omgewings.

3.2 Objekgeoriënteerde programmeertale.

3.3 Smalltalk.

3.4 Toestelle in die objekgeoriënteerde omgewing.

3.5 Smalltalk en datatipes.

Inhoud.

1 Oorsig.
2 Inleiding.
3 Objekgeoriënteerde programmering.
4 Abstrakte datatipes.
5 Abstrakte datatipes in programmeertale.
6 Die wiskunde van abstrakte datatipes.
7 Data-abstraksie in programmeertale : kort gevallestudies.
8 Abstrakte toestelle in 'n programmeertaal.
9 Rekordgebaseerde toestelle.
10 Hoër-vlak toestelle : 'n voorbeeld.
11 Implementering van die vertaler.
12 Doeltreffende rekenaargebruik.
A Die programmeertaal PT.
B Verwysings en bibliografie.

-oOo-

3 Objekgeoriënteerde programmering.

3.1 Objekgeoriënteerde omgewings.

'n Objekgeoriënteerde omgewing is 'n omgewing waarin alle entiteite sogenaamde objekte is. Voorbeelde van sulke objekte is prosesse, karakterstringe, getalle, programme, ensovoorts. Enige iets in so 'n omgewing wat met 'n selfstandige naamwoord beskryf kan word, is 'n objek.

Elke objek is 'n eenheid wat slegs gebruik kan word met behulp van bewerkings wat deur die objek voorsien word. Hierdie bewerkings word geaktiveer met behulp van boodskappe wat aan die objek gestuur kan word, waarop die objek reageer. Die boodskappe wat aan 'n objek gestuur kan word, vorm die koppelvlak van die objek. Sulke boodskappe word gewoonlik gestel as 'n versoek wat spesifiseer wat die sender gedoen wil hê, eerder as 'n opdrag wat aan die objek voorskryf hoe die objek moet reageer.

'n Boodskap sê eerstens aan 'n objek wat verlang word. 'n Boodskap kan ook addisionele inligting, wat met die versoek geassosieer word, spesifiseer. So moet 'n boodskap aan 'n stapel, wat vra om 'n objek op die stapel af te druk, spesifiseer watter objek op die stapel afgedruk moet word. Dit gebeur ook dikwels dat 'n boodskap om inligting vra – 'n boodskap kan byvoorbeeld aan 'n stapel gestuur word om die boonste objek van die stapel af te haal en dit aan die sender van die boodskap te lewer. Om hiervoor voorsiening te maak, kan 'n boodskap beantwoord word. Die antwoord van 'n versoek soos die genoemde voorbeeld, is die gevraagde objek.

Objekte bevat beskrywings wat bepaal hoe die objek reageer wanneer dit boodskappe ontvang. So 'n beskrywing, wat presies

beskryf hoe die bewerking wat met 'n spesifieke boodskap geassosieer word, uitgevoer moet word, staan as 'n metode bekend. Die metodes wat 'n objek gebruik om sy bewerkings uit te voer, is net binne die objek bekend - ander objekte wat boodskappe na 'n objek stuur, weet net wat die objek gaan doen, maar nie hoe die objek dit gaan doen nie.

Dikwels word 'n aantal soortgelyke objekte benodig - 'n aantal skikkings kan byvoorbeeld nodig wees. Omdat sulke objekte dieselfde gedrag moet openbaar, behoort dit net een keer nodig te wees om die gedrag van al hierdie objekte te beskryf. Om dit moontlik te maak, word objekte in klasse gegroepeer. As 'n skikking-klas, byvoorbeeld, gedefinieer is en 'n skikking word benodig, is dit net nodig om te spesifiseer dat 'n objek van hierdie klas is - en is dit nie nodig om die gedrag van elke skikking weer te beskryf nie. So 'n objek staan dan bekend as 'n voorkoms van die skikking-klas. 'n Klas is ook 'n objek en kan boodskappe aanvaar wat vir 'n voorkoms van die klas vra.

UNIVERSITY OF JOHANNESBURG

3.2 Objekgeoriënteerde programmeertale.

'n Objekgeoriënteerde programmeertaal is 'n programmeertaal wat gebruik word vir programmering in 'n objekgeoriënteerde omgewing. So 'n programmeertaal maak dan ook gebruik van die terminologie wat in 'n objekgeoriënteerde omgewing aangetref word : objekte, boodskappe, metodes, klasse en voorkomste.

In 'n objekgeoriënteerde omgewing word die vermoë om berekenings te doen, as 'n inherente eienskap van objekte gesien. As programmatuur ontwikkel moet word, moet bepaal word watter objekte benodig word, asook watter boodskappe tussen die objekte gestuur moet word om die verlangde resultate te bereik. As die

benodigde objekte nie voorkom nie, is dit nodig om die nuwe objekte by die bestaande objekte te voeg. Dit word gedoen deur 'n nuwe klas te definieer, wat gedoen word deur die naam van die klas te gee, die formaat van boodskappe wat deur objekte van die klas ontvang kan word, te beskryf en dan die metodes wat objekte van hierdie klas moet gebruik om op die boodskappe te reageer, te verskaf. Die benodigde objekte kan nou gebruik word nadat 'n boodskap aan hierdie nuwe klas gestuur is, om vir voorkomste daarvan te vra.

Die nuwe objekte se metodes word beskryf in terme van boodskappe aan bestaande objekte. Dit gebeur dikwels dat een boodskap 'n resultaat lewer, wat later weer benodig word. Om hierdie tydelike resultate (wat almal objekte is) te bêre, voorsien 'n objekgeoriënteerde programmeertaal ook veranderlikes. Die veranderlikes word dan gebruik om objekte tydelik te hou, soos veranderlikes in 'n konvensionele programmeertaal gebruik word om getalle, stringe, ensovoorts, tydelik te hou.

UNIVERSITY OF JOHANNESBURG

3.3 Smalltalk.

Smalltalk is die oorspronklike en bekendste objekgeoriënteerde programmeertaal. Die ontwikkeling van Smalltalk het in die vroeë 1970's aan die Palo Alto-navorsingsentrum van Xerox begin as deel van 'n projek, wat ten doel gehad het om 'n hoogs gesofistikeerde, maar uiters gebruikersvriendelike persoonlike rekenaar te ontwikkel.

Smalltalk is in werklikheid meer as 'n programmeertaal - dit is ook 'n programontwikkelingsomgewing. Smalltalk omskep die rekenaar in 'n virtuele Smalltalk-masjien. Alles in hierdie masjien is objekte wat deur boodskappe beheer kan word. Die

programmeerder kan 'n boodskap aan 'n teksredigeerder stuur wanneer hy veranderings aan programmatuur wil aanbring; hy kan ook 'n boodskap aan die Smalltalk-vertaler stuur as hy programme wil vertaal. Gewone Smalltalk-sintaks word gebruik vir al hierdie boodskappe - die Smalltalk-vertaler word geroep sodra 'n boodskap ingetik (of op 'n ander manier verskaf) is, om die boodskap te vertaal en aan die gespesifiseerde objek te stuur.

'n Volledige beskrywing van Smalltalk, met 'n groot verskeidenheid voorbeelde van die gebruik van Smalltalk, is deur Goldberg en Robson [Gol83] aangeteken.

3.4 Toestelle in die objekgeoriënteerde omgewing.

Dit is duidelik dat toestelgeoriënteerde programmering, soos dit in 'n vorige afdeling beskryf is, baie met objekgeoriënteerde programmering ooreenkom - die grondliggende idee is dieselfde, naamlik om nuwe toestelle of objekte met behulp van bestaande objekte of toestelle te konstrueer.

'n Fisiese toestel word egter nie in 'n objekgeoriënteerde omgewing as 'n enkele objek gesien nie. So word die rekenaar (of terminaal) se skerm gewoonlik nie as 'n objek gesien nie, maar word die skerm in "vensters" verdeel, wat elk as 'n objek gesien word. Nuwe vensters kan geskep word wanneer dit benodig word, en verwyder word wanneer dit nie verder benodig word nie. Dit het die gevolg dat ook nuwe "fisiese" objekte soos vensters, geskep kan word deur 'n boodskap aan die klas van so 'n objek te stuur.

Smalltalk voorsien sogenaamde primitiewe bewerkings om die fisiese toestelle wat aan die Smalltalk-stelsel bekend is, te

beheer. Hierdie fisiese toestelle is die skerm, toetsbord, muis en skyfaandrywer. Nuwe fisiese toestelle kan nie by die Smalltalk-stelsel gevoeg word sonder om aan die stelsel self te verander nie.

As die objekgeoriënteerde omgewing uitgebrei word, sodat die fisiese toestelle ook objekte net soos enige ander objekte in die omgewing is, is daar hoofsaaklik een probleem : Hoe reageer 'n klas wat 'n fisiese toestel verteenwoordig, as so klas 'n versoek kry om nog 'n voorkoms van so 'n objek te skep? Een moontlikheid is om elke keer 'n nuwe objek te skep wat dieselfde fisiese toestel beheer. Hierdie oplossing het egter verskeie nadele. Eerstens is daar die probleem dat 'n boodskap aan een van die objekte die gedrag van van 'n ander objek wat dieselfde toestel beheer, op 'n onverwagte wyse kan beïnvloed. Tweedens is dit byna onmoontlik om verskillende prosesse te sinchroniseer as meer as een van hierdie prosesse dieselfde toestel met behulp van verskillende objekte gebruik.

Om toestelle as objekte te hanteer, sal die klas van klasse op verskillende maniere moet reageer as dit versoek word om 'n voorkoms van, onderskeidelik, klasse wat direk met een of meer fisiese objekte geassosieer word, en klasse waarvan 'n willekeurige aantal voorkomste geskep kan word. Dit sal gevolglik ook nodig wees om vir elke nuwe klas te spesifiseer in watter van die twee kategorieë die nuwe klas tuis hoort - iets wat nie in Smalltalk voorsien word nie.

UNIVERSITY OF JOHANNESBURG

3.5 Smalltalk en datatipes.

Smalltalk voorsien geen datatipes nie. Voorstanders van Smalltalk beskou dit as 'n voordeel - dit is byvoorbeeld moontlik om 'n stapel te implementeer waarop enige objek geplaas kan word - getalle, stringe, programme, ensovoorts, kan op dieselfde stapel geplaas word. Hierdie vermoë het potensieel baie kragtige gebruike. Dit het egter ook die probleme wat in konvensionele programmeertale met outomatiese tipe-omskakeling ondervind is, en aanleiding gegee het tot die ontwikkeling van tale soos Pascal, waarin streng gekontroleer word dat net waardes van die regte tipe op enige gegewe plek gebruik word en wat ook vereis dat waardes eksplisiet na 'n ander tipe omgeskakel word wanneer so 'n omskakeling nodig is.

Die grootste probleem van "tipeloosheid" is die onvermoë van die vertaler om sekere foute op te spoor : as 'n boodskap aan 'n objek veroorsaak dat 'n verkeerde objek as 'n resultaat gelewer word, sal dit nie onmiddellik opgemerk word nie (een objek is so goed soos 'n ander). Eers wanneer 'n boodskap later aan hierdie (verkeerde) objek gestuur word, wat nie een van die geldige boodskappe aan so 'n objek is nie, sal 'n fout opgemerk word. Dit maak dit uiters moeilik om vas te stel presies waar die oorspronklike fout voorgekom het. As so 'n fout in 'n gedeelte van die kode wat selde gebruik word, voorkom, mag die fout eers lank nadat die stelsel geïmplementeer is, ontdek word. In 'n taal waarin objekte tipes het, sal die vertaler onmiddellik opmerk as 'n resultaat van 'n verkeerde tipe gelewer word, sodat 'n groot persentasie van hierdie foute reeds deur die vertaler opgemerk kan word.

Dit is dus wenslik om tipes met objekte te assosieer. Die programmeertaal wat in 'n latere afdeling beskryf sal word, sal

uitgebreide tipekontroles voorsien.

Hoofstuk 4.

4 Abstrakte datatipes.

4.1 Datastrukture.

4.2 Data-omhulling.

4.3 Abstraksie.

Inhoud.

1 Oorsig.

2 Inleiding.

3 Objekgeoriënteerde programmering.

4 Abstrakte datatipes.

5 Abstrakte datatipes in programmeertale.

6 Die wiskunde van abstrakte datatipes.

7 Data-abstraksie in programmeertale : kort gevallestudies.

8 Abstrakte toestelle in 'n programmeertaal.

9 Rekordgebaseerde toestelle.

10 Hoër-vlak toestelle : 'n voorbeeld.

11 Implementering van die vertaler.

12 Doeltreffende rekenaargebruik.

A Die programmeertaal PT.

B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

4 Abstrakte datatipes.

4.1 Datastrukture.

Die vroegste programmeertale het die programmeerder van fasiliteite voorsien om veranderlikes te verklaar wat direk met behulp van die apparatuur verwerk kan word. So kon rekenaars byvoorbeeld heelgetalle en reële getalle hanteer en, gevolglik, is hiervoor voorsiening gemaak in FORTRAN.

'n Volgende bydrae tot die voorstelling van datastrukture is deur COBOL gemaak met die rekordstruktuur. Rekords het dit moontlik gemaak om verwante data-items saam te groepeer in 'n enkele eenheid. Dit het twee groot voordele. Eerstens word program-leesbaarheid verhoog omdat items wat logies saam hoort, as 'n eenheid saam verklaar kan word en die verwantskap tussen die items sodoende beklemtoon word. Tweedens maak dit bewerkings wat op al die items as 'n eenheid werk, moontlik - dit is byvoorbeeld moontlik om so 'n rekord as geheel op 'n lêer te skryf.

Baie navorsing is ook gedoen om goeie voorstellings van algemene datastrukture te vind, asook om geassosieerde algoritmes wat hierdie voorstellings manipuleer of gebruik, te ontwikkel. Dit het aanleiding gegee tot deeglike beskrywings van datastrukture soos byvoorbeeld stapels, toue en grafieke. Hoër-vlak data-strukture, soos simbooltabelle, wat van hierdie meer elementêre datastrukture gebruik maak, is ook ontwikkel. Sien [Hor76] vir besonderhede.

Nog 'n ontwikkeling vir die voorstelling van datastrukture, was die voorsiening van 'n fasiliteit om 'n beskrywing van 'n nuwe datastruktuur te definieer, dit 'n naam te gee en dan die naam

van hierdie beskrywing te gebruik om voorkomste van die datastruktuur te verklaar. Die wyse waarop Pascal dit ondersteun met behulp van die Type-verklaring, is seker die beste voorbeeld van die gebruik van sulke beskrywings. As datastrukture (of veranderlikes) dieselfde reeks waardes kan aannneem, word gesê dat hulle van dieselfde tipe is. Die beskrywing van 'n datastruktuur word dan ook dikwels 'n tipebeskrywing genoem.

4.2 Data-omhulling.

Die werk wat in die gebied van datastrukture gedoen is, het dit dus vir die programmeerder moontlik gemaak om datastrukture te ontwikkel wat by sy spesifieke probleem pas, of om selfs beskrywings van datastrukture uit die literatuur te neem en dit net so te gebruik. Evalueringsmetodes het hom in staat gestel om sy implementering van datastrukture wiskundig met ander implementerings van soortgelyke datastrukture te vergelyk. Programmeertale is ook van fasiliteite voorsien sodat so 'n datastruktuur redelik maklik geïmplementeer kan word.

Een probleem wat egter nog gebly het, was die feit dat die voorstelling van 'n datastruktuur "oop" was vir die res van die program, met ander woorde, 'n datastruktuur kon op enige plek in die program op 'n ongeldige wyse verander word. Dit het tot gevolg dat daar (moontlik heelwat) later 'n fout kan voorkom as 'n ander gedeelte van die program die datastruktuur gebruik. Sulke foute wat nie opgemerk word op die plek waar die oorspronklike fout gemaak word nie, maar later probleme kan veroorsaak, maak dit baie moeilik om die werklike fout op te spoor - dit is dus wenslik om meganismes te vind wat die moontlikheid van sulke foute kan verminder.

'n Doeltreffende manier om die integriteit van 'n datastruktuur te beskerm, is om net enkele roetines toe te laat om die datastruktuur direk te gebruik. Die res van die program mag die datastruktuur dan slegs met behulp van hierdie roetines gebruik. Omdat daar nou op relatief min plekke in die program aan die datastruktuur verander kan word, is dit makliker om te kontroleer dat die datastruktuur op al hierdie plekke korrek gebruik word. Dit maak dit ook makliker om te kontroleer dat die datastruktuur nog steeds korrek gebruik word, nadat daar aan die datastruktuur verander is.

Dit is egter nie genoeg om van die programmeerder te verwag om datastrukture op so 'n gedissiplineerde wyse te gebruik nie - die vertaler wat hy gebruik, moet kontroleer dat die datastruktuur net deur die enkele roetines wat daarvoor ontwerp is, gebruik word. Die programmeertaal moet dit ook duidelik maak watter roetines direkte toegang tot die datastruktuur het.

'n Omhulde datatipe is 'n beskrywing van 'n datastruktuur en roetines wat die datastruktuur kan manipuleer waar die voorstelling van die datastruktuur (en gevolglik die interne werking van die roetines wat die datastruktuur manipuleer) afgeskerm is van ander programmatuur. Ander programmatuur wat die datastruktuur wil gebruik, kan dit slegs met behulp van die roetines wat deur die datatipe voorsien word, doen.

Voorbeeld 4.2.1
As 'n stapel benodig word om, sê, heelgetalle te hou, kan die datatipe Stapel gedefinieer word, waar Stapel die roetines StapelOp en StapelAf voorsien om, onderskeidelik, items op die stapel af te druk en van die stapel af te haal. Intern kan Stapel 'n skikking of 'n geskakelde lys gebruik om die stapel

UNIVERSITY OF JOHANNESBURG

voor te stel - hierdie inligting is egter net binne die datatipe bekend, en is versteek van gebruikers van die datatipe.

□

4.3 Abstraksie.

Die doel van abstraksie is om ooreenkomste te beklemtoon en verskille te ignoreer. As iets vir 'n abstraksie geld, geld dit ook vir al die konsepte of objekte wat aanleiding tot die abstraksie gegee het.

Dit is reeds lank die gebruik om programlyne wat 'n spesifieke taak verrig, saam te groepeer in 'n sogenaamde prosedure (of funksie). Die beskrywing van so 'n roetine beskryf dan wat die roetine doen - en nie hoe die roetine dit doen nie. 'n Beskrywing van 'n sorteerroetine sal byvoorbeeld spesifiseer dat die roetine 'n skikking as 'n parameter neem en dit dan in, sê, stygende orde sorteer. Die roetine kan dan gebruik word sonder dat die programmeerder weet watter sorteermetode gebruik word - die metode wat gebruik word, kan selfs verander word sonder dat dit nodig is om aan die programme wat die roetine gebruik, te verander. Die beskrywing van die roetine sluit dus dit in wat by alle sorteermetodes voorkom en laat alles wat net op 'n spesifieke metode betrekking het, uit. Die prosedurebeskrywing is dus 'n abstraksie van alle moontlike metodes wat gebruik kan word om die verlangde doel te gebruik.

In dieselfde sin kan die omhulde datatipe gebruik word om 'n abstrakte beskrywing van 'n datastruktuur te gee. Hier word die beskrywing van die datastruktuur (met behulp van beskrywings van die beskikbare bewerkings) in terme van die manier waarop die

datastruktuur hom moet gedra, gegee. Die besonderhede van die interne werking van die struktuur word dan binne die tipe versteek.

'n Omhulde datatipe word daarom ook 'n abstrakte datatipe genoem. Vir ander programmatuur is 'n abstrakte datatipe dus 'n versameling objekte met bewerkings wat op hierdie objekte gedefinieer is. Sien [Sim84] vir 'n beskrywing van abstrakte datatipes.

Die gebruik van abstrakte datatipes het heelwat voordele. Eerstens het dit natuurlik al die voordele wat by die bespreking van omhulde datatipes genoem is. Omdat 'n abstrakte datatipe nie voorskryf hoe die tipe geïmplementeer moet word nie, beteken dit dat die implementasie verander kan word, sonder dat dit nodig is om aan programme wat die abstrakte datatipe gebruik, te verander. Verder is dit moontlik om die gedrag van so 'n abstrakte datatipe in (presiese) wiskundige terme te beskryf, wat moontlike dubbelsinnighede uitskakel en wiskundige kontrolering van die korrektheid van die tipe moontlik maak.

Navorsing wat op die terrein van abstrakte datatipes gedoen is, het veral op twee gebiede gekonsentreer: daar is gesoek na geskikte voorstellings van abstrakte datatipes in programmeertale en die wiskunde van abstrakte datatipes is ondersoek. In die volgende afdeling sal na tipiese voorstellings van abstrakte datatipes in die nuwer programmeertale gekyk word, waarna die wiskunde van abstrakte datatipes kortliks onder oë geneem sal word.

Hoofstuk 5.

5 Abstrakte datatipes in programmeertale.

- 5.1 Inleiding.
- 5.2 Tipes en voorkomste van tipes.
- 5.3 Modules.
- 5.4 Geparametriseerde tipes.
- 5.5 Abstrakte toestelle en abstrakte datatipes.

Inhoud.

1 Oorsig.
2 Inleiding.
3 Objekgeoriënteerde programmering.
4 Abstrakte datatipes.
5 Abstrakte datatipes in programmeertale.
6 Die wiskunde van abstrakte datatipes.
7 Data-abstraksie in programmeertale : kort gevallestudies.
8 Abstrakte toestelle in 'n programmeertaal.
9 Rekordgebaseerde toestelle.
10 Hoër-vlak toestelle : 'n voorbeeld.
11 Implementering van die vertaler.
12 Doeltreffende rekenaargebruik.
A Die programmeertaal PT
B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

5 Abstrakte datatipes in programmeertale.

5.1 Inleiding.

Heelwat van die nuwer programmeertale voorsien 'n gerieflike manier om abstrakte datatipes te beskryf. 'n Tipiese manier waarop dit gedoen kan word, word in die volgende voorbeeld gegee.

Voorbeeld 5.1.1
'n Beskrywing van 'n abstrakte datatipe, *Stapel*, (soos dit tipies in 'n nuwer programmeertaal gegee sal word) kan soos volg lyk :

```
Struktuur Stapel is
  Bewerkings
    StapelOp, StapelAf, StapelLeeg, StapelVol, StapelInis
  Voorstelling
    S : Skikking[1..100] van Heel
    BoPunt : Heel
  VoorstEinde

  Prosedure StapelInis
  Begin
    BoPunt:= 0
  Einde

  Prosedure StapelOp(N : Heel)
  Begin
    BoPunt := BoPunt + 1
    S[BoPunt] := N
  Einde
```

UNIVERSITY OF JOHANNESBURG

```
    Funksie StapelAf : Heel
    Begin
      StapelAf := S[BoPunt]
      BoPunt := BoPunt - 1
    Einde

    Funksie StapelLeeg : Logies
    Begin
      As BoPunt = 0 dan
        StapelLeeg := Waar
      Anders
        StapelLeeg := Vals
    Einde

    Funksie StapelVol : Logies
    Begin
      As BoPunt = 100 dan
        StapelVol := Waar
      Anders
        StapelVol := Vals
    Einde

  Einde Stapel
```

□

Soos reeds vantevore genoem, kan 'n struktuur soos die stapel van voorbeeld 5.1.1, slegs deur middel van sy bewerkings gebruik word. Die bewerkings (en ook net daardie bewerkings wat in die *Bewerkings*-lys genoem is) is die enigste inligting wat in die struktuur genoem word, wat aan ander programmatuur bekend is. Die inligting in verband met 'n abstrakte datatipe wat aan ander programmatuur verskaf word, kan dus as 'n opsomming gegee word

wat net die nodige inligting bevat.

Voorbeeld 5.1.2
Die inligting van die Stapel van voorbeeld 5.1.1, wat aan ander programmatuur bekend is, kan soos volg opgesom word :

```
Koppelvlak Stapel
  Bewerkings
    StapelOp(Heel)
    StapelAf : Heel
    StapelLeeg : Logies
    StapelVol : Logies
    StapelInis
Einde Koppelvlak
```

□

Heelwat programmeertale laat dan ook 'n skeiding toe tussen die gedeelte wat beskryf hoe die tipe lyk (die koppelvlak) en die gedeelte wat die implementering van die tipe beskryf (die implementasiegedeelte). So 'n skeiding maak dit moontlik om 'n program wat 'n tipe gebruik, afsonderlik te vertaal van die implementasie van die tipe - die program wat die tipe benodig hoef slegs van die koppelvlak van die tipe voorsien te word, aangesien die koppelvlak al die inligting bevat wat deur die program benodig word. Aparte vertaling maak dit natuurlik makliker om aan die implementasie van die tipe te verander sonder dat dit enigsins nodig is om na die programme wat die tipe gebruik, te kyk. Verder maak die aparte vertaling dit moontlik om tipes in 'n biblioteek te hou, vanwaar ander programme dit kan gebruik.

Waar die koppelvlak van 'n tipe nie afsonderlik gegee word nie,

soos in die eerste voorbeeld van die *Stapel*, word die koppelvlak tog duidelik gespesifiseer. In die voorbeeld se notasie word, byvoorbeeld, vereis dat alle bewerkings wat buite die tipe gebruik kan word, eksplisiet in die *Bewerkings*-lys genoem word. Die parameters wat deur die onderskeie bewerkings benodig word, word elders in die beskrywing van die tipe gevind. Hierdie inligting is die enigste inligting wat buite die tipebeskrywing bekend is en dus deel van die (implisiete) koppelvlak vorm. Dit is ook relatief maklik om 'n vertaler te skryf wat slegs die gebruik van inligting wat in die implisiete koppelvlak voorkom, toelaat.

Die koppelvlak wat as voorbeeld 5.1.2 gegee is, beskryf net die sintaks van die tipe - met ander woorde, die formaat wat programme wat die tipe benodig, moet gebruik. Die semantiek van die tipe, dit wil sê die wyse waarop die tipe hom gedra, word wel deur die implementasie van die tipe gegee. Omdat die groot doel van abstrakte datatipes die afskerming van besonderhede van die implementering is, is dit nie voldoende nie. 'n Beskrywing van die gedrag van die tipe behoort dus saam met die koppelvlak voorsien te word. Hierdie beskrywing kan op 'n informele manier gegee word, of dit kan in 'n formele wiskundige notasie gegee word. So 'n semantiese spesifikasie van 'n tipe dien dan as definisie van die tipe se gedrag vir beide die implementeerder en die gebruiker van die tipe. Die korrektheid van die implementasie van die tipe kan bepaal word, deur dit aan die semantiese spesifikasie van die tipe te toets. Die semantiese spesifikasie van die tipe kan ook gebruik word om te verifieer dat dit korrek deur ander programmatuur gebruik word.

Voorbeeld 5.1.3
'n Informele spesifikasie van die tipe *HeelStapel*, wat 'n semantiese beskrywing insluit, kan soos volg lyk :

HeelStapel is
  Skep, Op, Af, Boonste, Vol, Leeg

HeelStapels is strukture wat gebruik word om heelgetalle te bêre op 'n laaste-in-eerste-uit grondslag. Leë HeelStapels kan geskep word (m.b.v. Skep). HeelStapels kan ook uit ander HeelStapels geskep word met die bewerkings Op en Af. Boonste lewer die boonste heelgetal op 'n HeelStapel. Vol en Leeg toets, onderskeidelik, of 'n HeelStapel vol is of geen heelgetalle bevat nie.

Skep() lewer HeelStapel
  effek : lewer 'n leë HeelStapel

Op(S : HeelStapel, N : Heel) lewer HeelStapel sein Geen_Ruimte
  effek : as genoeg ruimte lewer S met N bo-op bygevoeg
          anders sein Geen-Ruimte

Af(S : HeelStapel) lewer HeelStapel
  effek : as S leeg is, lewer S
          anders lewer S sonder die boonste element

Boonste(S : HeelStapel) lewer HeelGetal sein Bestaan_Nie
  effek : as Stapel nie leeg nie lewer boonste element van S
            (sonder om dit van die stapel te verwyder)
          anders sein Bestaan_Nie

Leeg(S : HeelStapel) lewer Logies
  effek : lewer Waar as en slegs as S geen elemente bevat nie

Vol(S : HeelStapel) lewer Logies
    effek : lewer Waar as en slegs as daar nie ruimte beskikbaar
        is om die HeelStapel wat gevorm sal word as nog 'n
        element by S gevoeg word, voor te stel nie.

□

Die voorafgaande informele spesifikasie bestaan uit drie dele. Die eerste deel is die opskrif wat die naam van die tipe gee en die bewerkings van die die tipe lys. Die tweede gedeelte is 'n kort beskrywing van die tipe as 'n geheel, terwyl die laaste gedeelte elk van die bewerkings beskryf. Die formaat van hierdie informele spesifikasie is aan Liskov [Lis90] te danke.

Formele semantiese spesifikasies van tipes sal in die volgende afdeling bespreek word.

UNIVERSITY
OF
JOHANNESBURG

5.2 Tipes en voorkomste van tipes.

Dikwels word daar in 'n program meer as een soortgelyke datastruktuur benodig, en dit is dus wenslik om 'n fasiliteit te voorsien sodat so 'n datastruktuur net een keer beskryf hoef te word, waarna hierdie tipebeskrywing gebruik kan word om meer as een voorkoms van so 'n tipe te definieer. Daar is hoofsaaklik twee maniere waarop dit gedoen kan word. Eerstens kan 'n tipebeskrywing 'n *Skep*-bewerking voorsien, soos in die voorafgaande voorbeeld van *HeelStapels*. Die *Skep*-bewerking kan dan elke keer wat 'n voorkoms van die tipe benodig word, geroep word. Nuwe voorkomste van datatipes word dus in hierdie geval tydens looptyd geskep. Die tweede manier waarop meerdere voorkomste van tipe uit 'n enkele tipebeskrywing verkry kan word, is deur die tipe-

beskrywing 'n naam te gee en die naam dan te gebruik om veranderlikes van die tipe te verklaar. As daar byvoorbeeld 'n tipebeskrywing *Stapel* (wat 'n stapel beskryf) voorkom, kan 'n veranderlike S van hierdie tipe soos volg verklaar word :

```
Verand
   S : Stapel
```

'n Verklaring soos hierdie impliseer gewoonlik dat S na 'n voorkoms van die tipe *Stapel* sal verwys. Hierdie voorkoms word geskep wanneer die prosedure (of ander blok) waarbinne die verklaring voorkom, binnegegaan word, en weer vernietig word wanneer die blok verlaat word. Die programmeerder het dus nie nodig om uitdruklik 'n voorkoms van die tipe te skep nie. Dit mag wel nodig wees om die voorkoms te inisialiseer voor dit gebruik kan word. Dit is natuurlik moontlik om 'n taal (en vertaler) so te ontwerp dat die inisialiseringskode outomaties uitgevoer word, wanneer die blok waarin die verklaring voorkom, binnegegaan word.

5.3 Modules.

Sommige programmeertale voorsien 'n fasiliteit wat gebruik kan word om gedeeltes van 'n program, soos veranderlikes, prosedures, ensovoorts, saam te groepeer. In sommige tale word die konstruksie wat hiervoor gebruik word, 'n module genoem (byvoorbeeld Modula en Modula-2) en in ander 'n pakket (byvoorbeeld Ada). Die term module sal hier gebruik word om na beide modules en pakkette, asook na ander soortgelyke konstruksies, te verwys.

Modules kan suksesvol gebruik word om abstrakte datatipes te

implementeer. Heelwat van die programmeertale wat modules voorsien, voorsien nie spesiale meganismes om abstrakte datatipes te implementeer nie - waarskynlik omdat die ontwerpers voel dat die modules reeds genoeg ondersteuning daarvoor voorsien - in die volgende paar paragrawe sal aangetoon word hoe modules hiervoor gebruik kan word.

Modules het gewoonlik 'n naam, wat gevolg word deur 'n *Uitvoer*-lys. Die *Uitvoer*-lys noem die name van alles wat binne die module verklaar word, maar ook buite die module beskikbaar is. Sommige programmeertale vereis ook 'n *Invoer*-lys, waar name wat buite die module verklaar is, en binne die module gebruik word, gegee word. Na hierdie lyste volg die definisies van die items (veranderlikes, prosedures, ensovoorts) binne die module. Sommige programmeertale maak ook voorsiening vir 'n skeiding tussen die koppelvlak en implementasie van 'n module. In sulke gevalle kan die implementasie van die module en 'n program, wat die module gebruik, afsonderlik vertaal word.

UNIVERSITY OF JOHANNESBURG

Die volgende voorbeeld illustreer hoe 'n modules gebruik kan word om 'n abstrakte datatipe te implementeer.

Voorbeeld 5.3.1
'n Stapel (waarvan daar in hierdie geval net een voorkoms sal wees) kan soos volg met behulp van 'n module geïmplementeer word :

```
  Module Stapel
  Def
    Op, Af, Leeg, Vol, Inis
```

```
Verand
  S : Skik[1:10] van Heel
  BoPunt : Heel

Prosedure Inis
Begin
  BoPunt := 0
Einde

Prosedure Op(n : Heel)
Begin
  BoPunt := BoPunt + 1
  S[BoPunt] := n
Einde

Funksie Af : Heel
Begin
  Af := S[BoPunt]
  BoPunt := BoPunt - 1
Einde

Funksie Vol : Logies
Begin
  Vol := BoPunt >= 10
Einde

Funksie Leeg : Logies
Begin
  Leeg := BoPunt = 0
Einde

Einde
```
□

UNIVERSITY OF JOHANNESBURG

As meer as een voorkoms van 'n stapel benodig word kan die tipebeskrywing "Stapel" soos in voorbeeld 5.3.2 geïmplementeer word. Die verklarings

```
Verand
  A : Stapel
  B : Stapel
```

sal dan veroorsaak dat beide A en B stapels is. Op(A, 1) sal 1 op stapel A plaas, terwyl Op(B, 2) die getal 2 op stapel B sal plaas.

Voorbeeld 5.3.2
Hierdie voorbeeld illustreer die implementering van 'n klas van toestelle met behulp van 'n module. Die *Stapel*-definisie beskryf 'n klas van stapels.

UNIVERSITY OF JOHANNESBURG

```
Module
Def
  Stapel, Inis, Op, Af, Vol, Leeg

Tipe
  Stapel = Rekord
    S : Skik[1:10] van Heel
    BoPunt : Heel
  RekEinde

Prosedure Inis(St : Stapel)
Begin
  St.BoPunt := 0
Einde
```

```
Prosedure Op(St : Stapel, i : Heel)
Begin
  St.BoPunt := St.BoPunt + 1
  St.S[St.BoPunt] := i
Einde

Funksie Af(St : Stapel) : Heel
Begin
  Af := St.S[St.BoPunt]
  St.BoPunt := St.BoPunt - 1
Einde

.
.
.

Einde
```

UNIVERSITY OF JOHANNESBURG

□

Modules kan ook gebruik word om tipes te definieer, waar voorkomste tydens looptyd geskep moet word. Die manier waarop dit gedoen word, is om 'n *Skep*-bewerking te voorsien wat 'n geheueblok (waarin die interne voorstelling van die voorkoms van die datatipe gehou gaan word) van die bedryfstelsel aanvra. Die *Skep*-bewerking doen ook enige nodige inisialiseringswerk, waarna dit 'n wyser na die geheueblok aan die roepende program lewer. Hierdie wyser word dan aan die ander bewerkings van die abstrakte datatipe gestuur, wat dit dan gebruik om die betrokke voorkoms van die datatipe te manipuleer.

5.4 Geparametriseerde tipes.

Dit gebeur dikwels dat twee (of meer) datastrukture benodig word wat so min van mekaar verskil dat dit onnodig behoort te wees om aparte beskrywings van die datastrukture te gee. Een voorbeeld van so 'n geval, is waar 'n stapel van reële getalle en 'n stapel van heelgetalle benodig word. 'n Ander geval kom voor waar 'n aantal toue (sê maar almal van heelgetalle) benodig word, maar waar die maksimum lengte van die toue verskil. In laasgenoemde voorbeeld kan die lengte van die langste tou gebruik word, maar dit kan tot 'n vermorsing van geheueruimte lei.

Die oplossing vir hierdie probleme, is om parameters toe te laat by 'n definisie van 'n abstrakte datatipe. So is dit byvoorbeeld moontlik om 'n datatipe "Stapel van tipe T" te definieer, waar T enige tipe is waarvoor 'n toewysingsbewerking gedefinieer is. 'n Voorkoms van hierdie stapel wat gebruik kan word om heelgetalle te hou, kan nou verkry word deur die volgende verklaring :

```
Verand
   SH : Stapel van Heel
```

'n Voorkoms van die stapel om reële getalle hou, kan soortgelyk verkry word deur die verklaring :

```
Verand
   SR : Stapel van Reeel
```

Om die probleem van soortgelyke datastrukture met verskillende groottes op te los, kan die grootte van 'n abstrakte datatipe 'n parameter van die beskrywing van die tipe wees. So kan die abstrakte datatipe "Tou(L)" byvoorbeeld gedefinieer word, waar L

die maksimum lengte van die tou is. Om twee voorkomste, T10 en T20, van 'n tou wat, onderskeidelik, tien en twintig elemente kan bevat, te definieer, kan die volgende verklaring gebruik word :

```
Verand
   T10 : Tou(10)
   T20 : Tou(20)
```

'n Fasiliteit wat definisies van geparametriseerde abstrakte datatipes moontlik maak, het potensieel ook baie ander gebruike. Van hierdie gebruike sal in 'n volgende paragraaf verder bespreek word.

Een van die programmeertale wat parameters by die definisie van 'n abstrakte datatipe op 'n baie netjiese wyse ondersteun is die taal CLU [Lis78]. Die generiese modules van Ada maak ook sulke definisies moontlik, maar op 'n minder elegante manier.

UNIVERSITY OF JOHANNESBURG

5.5 Abstrakte toestelle en abstrakte datatipes.

Dit is duidelik dat 'n abstrakte datatipe ook 'n abstrakte toestel is volgens die beskrywing van abstrakte toestelle, wat in afdeling 2 gegee is. Die vraag is nou tot watter mate die teendeel waar is.

In die meeste gevalle kan 'n fasiliteit wat deur 'n programmeertaal voorsien word word om abstrakte datatipes te definieer, net so gebruik word om abstrakte toestelle te definieer. Daar is egter enkele probleme wat mag voorkom.

Die eerste moontlike probleem kom voor by die siening van 'n

abstrakte datatipe as 'n versameling objekte met bewerkings op die objekte. Gewoonlik word aanvaar dat die gedrag van 'n abstrakte datatipe slegs deur sy bewerkings beïnvloed kan word - dit is die fondament van die wiskundige hantering van abstrakte datatipes. Met toestelle is dit egter nie die geval nie. 'n Drywerprogram van 'n toetsbord sal byvoorbeeld net 'n roetine *LeesKar* voorsien, wat die karakterkode van die volgende toets wat gedruk is, sal lewer. Dit is egter maklik om te sien dat die gedrag van hierdie toestel nie slegs van die bewerking *LeesKar* afhang nie. In die volgende afdeling, waar na die wiskundige hantering van abstrakte datatipes gekyk sal word, sal ook aangetoon word hoe die gedrag van abstrakte toestelle (net soos die gedrag van abstrakte datatipes) met behulp van aksiomas gespesifiseer kan word.

Programmeertale wat abstrakte datatipes ondersteun, voorsien dikwels 'n fasiliteit om meer as een voorkoms van 'n spesifieke datastruktuur te kry. Hierdie fasiliteit is wenslik by abstrakte datatipes en is ook dikwels wenslik by abstrakte toestelle. 'n Probleem kom egter voor waar 'n abstrakte toestel 'n fisiese toestel beheer; in so 'n geval kan daar nie willekeurig nuwe voorkomste van die abstrakte toestel geskep word nie. Een moontlike oplossing is om die fisiese besonderhede (soos die poortadresse van die data-, beheer- en statuspoorte) van die toestel as parameters van die definisie van die abstrakte toestel te gee, en dan te verwag dat die gebruikers van die toestel net een voorkoms van die abstrakte toestel vir elk van die fisiese toestelle sal verklaar. Hierdie oplossing is egter nie altyd wenslik nie, en gevolglik behoort daar fasiliteite te wees om 'n enkele - sowel as *meerdere* - voorkomste van abstrakte toestelle moontlik te maak.

Omdat sommige toestelle baie meer data hanteer as wat in die

interne geheue van die rekenaar gehou kan word, is kragtige metodes ontwikkel om vinnig toegang tot spesifieke data te kry – die kragtigste metodes laat die gebruik van meer as een veld van 'n rekord as 'n sleutel om die rekord vinnig op te spoor, toe. As die abstrakte toestel *DataBasis*, byvoorbeeld gedefinieer word, moet dit moontlik wees om te spesifiseer dat een of meer van die velde van die rekords wat in die databasis mag voorkom, sleutels moet wees. Verder moet daar ook 'n manier voorsien word om, wanneer 'n voorkoms van *DataBasis* verklaar word, aan te dui watter velde van die rekords wat in hierdie voorkoms gehou gaan word, sleutels is.

Hierdie laaste vorm van parametrisering, waar die abstrakte datatipe op 'n rekordtipe gebou is, en waar sommige velde van die rekord 'n spesiale betekenis het, word nie deur bestaande programmeertale ondersteun nie.

Hoofstuk 6.

6 Die wiskunde van abstrakte datatipes.

6.1 Algebras.

6.2 Algebraïese spesifikasie van 'n abstrakte datatipe.

6.3 Die sintaktiese spesifikasie.

6.4 Die semantiese spesifikasie.

6.5 Aksiomas : volledigheid en konsistensie.

6.6 Aksiomas en abstrakte toestelle.

Inhoud.

1 Oorsig.

2 Inleiding.

3 Objekgeoriënteerde programmering.

4 Abstrakte datatipes.

5 Abstrakte datatipes in programmeertale.

6 Die wiskunde van abstrakte datatipes.

7 Data-abstraksie in programmeertale : kort gevallestudies.

8 Abstrakte toestelle in 'n programmeertaal.

9 Rekordgebaseerde toestelle.

10 Hoër-vlak toestelle : 'n voorbeeld.

11 Implementering van die vertaler.

12 Doeltreffende rekenaargebruik.

A Die programmeertaal PT.

B Verwysings en bibliografie.

-oOo-

6 Die wiskunde van abstrakte datatipes.

6.1 Algebras

'n <u>Algebra</u> is 'n geordende paar <A;F>, waar A 'n nie-leë versameling is en F 'n nie-leë versameling funksies van A^n, $n \geq 0$, in A is. By die definisie van abstrakte datatipes is genoem dat 'n abstrakte datatipe 'n versameling objekte met 'n versameling bewerkings op hierdie objekte is. Die ooreenkoms tussen 'n algebra en 'n abstrakte datatipe is duidelik. 'n Algebra voorsien dus 'n gerieflike basis vir 'n wiskundige hantering van abstrakte datatipes.

Die bewerkings van 'n abstrakte datatipe werk egter nie altyd net op die objekte van die datatipe nie. As S byvoorbeeld 'n stapel van heelgetalle is, sal ons die volgende bewerkings benodig :

UNIVERSITY
OF
JOHANNESBURG

Skep : -> S
Op : S x Heel -> S
Af : S -> S
Bo : S -> Heel

Die funksies Op en Bo werk nie net op elemente van S nie - dit werk ook op elemente van Heel (die versameling heelgetalle). Dit is dus gewoonlik gerieflik om te aanvaar dat sommige datatipes reeds bestaan wanneer 'n nuwe datatipe gedefinieer word - soos in die bostaande voorbeeld aanvaar is dat die datatipe Heel reeds bestaan. Aangesien die bewerkings van 'n algebra <A;F> net op die versameling A (of A^n, $n \geq 0$) gedefinieer is (en dus nie van die bestaan van 'n ander algebra gebruik kan maak nie) is die gewone (homogene) algebra nie heeltemal geskik vir die beskrywing van 'n abstrakte datatipe nie.

Die heterogene algebras van Birkhoff en Lipson [Bir70] los egter hierdie probleem op. 'n Heterogene algebra A = <P;F> is 'n geordende paar, waar P 'n familie van nie-leë versamelings is en F 'n versameling van bewerkings is. Die definisiegebied van enige van hierdie bewerkings is 'n kruisproduk van n van die elemente van P ($n \geq 0$, n eindig), terwyl die waardegebied een van die elemente van P is. In [Bir70] word ook aangetoon dat heelwat van die basiese stellings wat vir homogene algebras geld, ook vir heterogene algebras geld.

6.2 Algebraïese spesifikasie van 'n abstrakte datatipe.

Die algebraïese spesifikasie van 'n abstrakte datatipe word volledig deur Guttag en Horning in [Gut78] bespreek. Hierdie spesifikasies is op die heterogene algebras gebaseer.

Die taal wat vir algebraïese spesifikasies gebruik word, maak van vyf "primitiewe" gebruik :

1) Funksionele samestelling
2) 'n Gelykheidsrelasie
3) 'n Konstante "Waar"
4) 'n Konstante "Vals"
5) 'n Onbegrensde aantal vrye veranderlikes

'n Algebraïese spesifikasie bestaan uit twee dele : 'n sintaktiese spesifikasie en 'n semantiese spesifikasie. Die sintaktiese spesifikasie noem die beskikbare bewerkings asook die definisie- en waardeversamelings van elk van die bewerkings. Die semantiese spesifikasie beskryf die werking (of effek) van die bewerkings.

UNIVERSITY OF JOHANNESBURG

'n Funksie wat dikwels benodig word is die *AsDanAnders*-funksie, wat soos volg gedefinieer word :

AsDanAnders(Waar, a, b) = a
AsDanAnders(Vals, a, b) = b

Ter wille van leesbaarheid word die *AsDanAnders*-funksie gewoonlik geskryf

As V Dan a Anders b

met dieselfde betekenis as

AsDanAnders(V, a, b).

6.3 Die sintaktiese spesifikasie.

Soos voorheen genoem, gee die sintaktiese spesifikasie al die beskikbare bewerkings, saam met hulle definisie- en waardegebiede.

UNIVERSITY OF JOHANNESBURG

Voorbeeld 6.3.1
Die sintaktiese spesifikasie van 'n tou (van heelgetalle) kan soos volg lyk :

Skep : -> Tou
Byvoeg : Tou x Heel -> Tou
Verwyder : Tou -> Tou
Voor : Tou -> Heel
Leeg? : Tou -> Logies

□

6 Die wiskunde van abstrakte datatipes.

Die bewerkings van 'n datatipe kan in twee kategorieë verdeel word :

1) Die generatorbewerkings; en
2) Die navraagbewerkings.

Die generatorbewerkings is daardie bewerkings waarvan die waardegebied die datatipe is wat gedefinieer word. In bogenoemde voorbeeld is *Skep*, *Byvoeg* en *Verwyder* dus generatorbewerkings. Enige tou kan egter met die bewerkings Skep en Byvoeg gegenereer word. Hierdie bewerkings, wat enige van die waardes wat 'n nuwe datatipe kan aanneem, kan genereer, staan as die basiese generatorbewerkings bekend.

Die waardes in die gebied van 'n datatipe, word gewoonlik nie as 'n lys gegee nie. So word die verskillende toue wat mag voorkom nie eksplisiet genoem nie. Die waardes word implisiet met behulp van die generatorbewerkings gegee : die waardes wat kan voorkom, is daardie waardes wat deur (herhaalde) toepassing van die generatorbewerkings verkry kan word. Guttag aanvaar in sy werk dat die waardes van 'n abstrakte datatipe altyd op so 'n implisiete manier gegee sal word. Dit is dus belangrik dat daar altyd minstens een generatorbewerking is.

Die navraagbewerkings verskaf inligting oor die abstrakte datatipe. Die bewerkings *Voor* en *Leeg?* van voorbeeld 6.3.1 is navraagbewerkings : *Voor* verskaf die voorste element van die tou, terwyl *Leeg?* gebruik kan word om vas te stel of die tou leeg is of nie. Omdat navraagbewerkings die enigste manier is om tussen verskillende waardes van 'n datatipe te onderskei, moet daar minstens een navraagbewerking voorkom - as daar nie tussen twee waardes van 'n datatipe onderskei kan word nie, word die twee waardes as 'n enkele waarde gesien. Die volgende voorbeeld

illustreer dit.

Voorbeeld 6.3.2
Veronderstel die datatipe PosHeel (positiewe heelgetalle) word soos volg gedefinieer :

```
Een  : -> PosHeel
Volg : PosHeel -> PosHeel
Ewe? : PosHeel -> Logies
```

Hier lewer die bewerking Een() die heelgetal 1. As n enige positiewe heelgetal is, lewer Volg(n) die positiewe heelgetal wat op n volg en Ewe?(n) lewer Waar as n ewe is, en Vals andersins. In hierdie definisie is dit egter onmoontlik om tussen enige twee ewe getalle of enige twee onewe getalle te onderskei - Ewe?(n) van enige ewe getal n is Waar ongeag watter ewe getal dit is en Ewe?(n) van enige onewe getal n is Vals ongeag watter onewe getal dit is. Volg kan ook nie gebruik word om tussen twee ewe of twee onewe getalle te onderskei nie.

As ons egter die "="-bewerking,

= : PosHeel x PosHeel -> Logies,

byvoeg, wat Waar lewer as en slegs as die twee positiewe heelgetalle dieselfde is, kan tussen enige twee getalle onderskei word - hierdie bewerking veroorsaak dus dat PosHeel oneindig veel waardes voorsien, in teenstelling met die eerste geval waar PosHeel slegs twee onderskeibare waardes voorsien het.

□

6.4 Die semantiese spesifikasie.

In paragraaf 6.3 is van twee maniere gebruik gemaak om die effek van 'n bewerking te beskryf - die name van bewerkings is so gekies dat dit (intuïtief) beskryf wat die bewerking doen en soms is 'n woordelikse beskrywing gegee om die effek van 'n bewerking te spesifiseer. Albei hierdie tegnieke kan egter maklik tot dubbelsinnighede lei. Dit is dan ook die beweegrede vir die wiskundige spesifikasie van 'n abstrakte datatipe - 'n presiese, ondubbelsinnige wiskundige spesifikasie kan die genoemde probleme uitskakel en gevolglik tot meer betroubare programmatuur lei.

Die hulpmiddel wat gebruik gaan word om die effek (dit wil sê die semantiek) van die bewerkings te beskryf, is aksiomas. Die aksiomas word gebruik om al die navraagbewerkings en die nie-basiese generatorbewerkings te beskryf Die volgende voorbeelde illustreer die gebruik van aksiomas.

UNIVERSITY OF JOHANNESBURG

Voorbeeld 6.4.1

Die semantiek van die tou van voorbeeld 6.3.1 word deur die volgende aksiomas gegee :

```
Vir enige tou t en enige heelgetal i moet die volgende geld

Leeg?(Skep()) = Waar
Leeg?(Byvoeg(t, i)) = Vals
Verwyder(Byvoeg(t, i)) = As Leeg?(t) Dan Skep()
                         Anders Byvoeg(Verwyder(t), i)
Voor(Byvoeg(t, i)) = As Leeg?(t) Dan i
                     Anders Voor(t)

Voor(Skep()) = Fout
Verwyder(Skep()) = Fout
```

Die laaste twee aksiomas dui op toestande wanneer die bewerkings Voor en Verwyder nie gebruik mag word nie. Hierdie aksiomas word <u>beperkings</u> genoem.

□

Voorbeeld 6.4.2

'n Volledige spesifikasie van PosHeel, soos in voorbeeld 6.3.2 gegee, met die byvoeging van 'n paar nuwe bewerkings, sal soos volg lyk :

```
Tipe PosHeel

  Een   : -> PosHeel
  Volg  : PosHeel -> PosHeel
  Vorig : PosHeel -> PosHeel
  +     : PosHeel x PosHeel -> PosHeel
  Ewe?  : PosHeel -> Logies
  =     : PosHeel x PosHeel -> Logies
  Een?  : PosHeel -> Logies

Vir i, j : PosHeel

  Een?(Een()) = Waar
  Een?(Volg(i)) = Vals
  Vorig(Volg(i)) = i
  Ewe?(i) = As Een?(i) Dan Vals
            Anders Nie(Ewe?(Vorig(i)))
  =(i, j) = As Een?(i) Dan Een?(j)
            Anders As Een?(j) Dan Vals
                   Anders =(Vorig(i), Vorig(j))
  +(i, j) = As Een?(j) Dan Volg(i)
```


UNIVERSITY OF JOHANNESBURG

Anders Volg(+(i, Vorig(j)))

Vorig(Een()) = Fout

Die Nie-funksie wat in hierdie voorbeeld gebruik is, word vir 'n b, b ∈ {Waar; Vals} soos volg gedefinieer :

Nie(b) = As b Dan Vals Anders Waar.

□

As optelling met die voorafgaande aksiomas gedoen moet word - sê ons wil +(2, 3) bereken - sal dit soos volg gedoen word :

```
+(2, 3) = Volg(+(2, Vorig(3)))
        = Volg(+(2, 2))
        = Volg(Volg(+(2, Vorig(2))))
        = Volg(Volg(+(2, 1)))
        = Volg(Volg(Volg(2)))
        = Volg(Volg(3))
        = Volg(4)
        = 5
```

In hierdie berekening is 1 gebruik om Een() voor te stel, 2 om Volg(Een()) voor te stel, 3 vir Volg(Volg(Een())), ensovoorts.

Die voorafgaande berekening illustreer 'n groot verskil tussen 'n wiskundige benadering tot berekenings en die manier waarop berekenings op 'n rekenaar gedoen word : die aksiomas gee 'n gerieflike notasie om die die gedrag van die bewerkings te beskryf, maar die beskryfde metode sal baie rekenaartyd vermors. Die manier waarop 'n rekenaar berekenings doen (met behulp van binêre getalle en logiese bisbewerkings) is baie vinniger, maar weer moeiliker om in aksiomas te gebruik. Om hierdie rede is

UNIVERSITY OF JOHANNESBURG

dit nie prakties moontlik om 'n aksiomatiese spesifikasie te gebruik om 'n abstrakte datatipe in 'n programmeertaal te beskryf nie. As dit egter nodig is om 'n datatipe vinnig te implementeer sodat dit getoets kan word, is dit egter wel moontlik om die aksiomatiese spesifikasie direk te gebruik.

6.5 Aksiomas : volledigheid en konsistensie.

Twee probleme wat steeds kan voorkom as aksiomas gebruik word, is onvolledige of onkonsistente aksiomas. Aksiomas is onvolledig as daar 'n waarde in die gebied van 'n datatipe is, waarvoor die effek van 'n bewerking ongedefinieerd is. Aksiomas is onkonsistent as daar 'n waarde in die gebied van 'n datatipe is, waarvoor 'n spesifieke bewerking meer as een moontlike effek kan hê.

Guttag het aangetoon dat dit oor die algemeen nie moontlik is om vas te stel of 'n stel aksiomas volledig en konsistent is nie.

UNIVERSITY OF JOHANNESBURG

6.6 Aksiomas en abstrakte toestelle.

As die gedrag van abstrakte toestelle met aksiomas beskryf moet word, soos in die geval van abstrakte datatipes, is daar hoofsaaklik twee probleme :

1) Die gedrag van die toestelle moet volledig deur aksiomas beskryf word. Dit kan veral 'n probleem wees by 'n toestel soos 'n toetsbord, waar 'n mens betrokke is : die gedrag van die toetsbord hang af van watter toets die gebruiker druk.
2) Die toestel moet generatorbewerkings verskaf. Die probleem hier is hoe 'n generatorbewerking "Skep" analoog aan die

Skep-bewerking van, byvoorbeeld, 'n tou om 'n leë tou te skep, op 'n fisiese toestel hanteer moet word.

Hierdie probleme kan opgelos word deur twee nuwe soorte bewerkings toe te laat.

Eerstens kan bewerkings toegelaat word, wat nie deur die program gebruik word nie, maar deur die gebruiker. Die toetsbord kan byvoorbeeld die bewerkings *LeesKar* en *DrukToets* voorsien, waar LeesKar op die normale manier deur die program gebruik word, en DrukToets implisiet deur die gebruiker gebruik word wanneer hy 'n toets op die toetsbord druk.

Tweedens kan skynbewerkings toegelaat word wat denkbeeldig uitgevoer word wanneer die rekenaar geïnisialiseer word. Die "Skep"-bewerking van 'n fisiese toestel kan dan so 'n bewerking wees. So kan byvoorbeeld aanvaar word dat die toestel *Drukker* 'n bewerking *SkepDrukker* toelaat, wat so denkbeeldig uitgevoer word om die drukker te "skep". Die *SkepDrukker*-bewerking word dan nie werklik voorsien nie, maar word wel ter wille van die aksiomatiese spesifikasie by laasgenoemde spesifikasie ingesluit.

Dit gebeur dikwels dat 'n toestel inisialiseringskode benodig of toelaat. 'n Bandaandrywer kan byvoorbeeld 'n bewerking *LaaiBand* toelaat. 'n Drukker kan 'n inisialiseringsroetine verskaf wat geroep kan word om die drukker in 'n vaste (bekende) toestand te plaas - die roetine sal veroorsaak dat die aantal lyne per bladsy na 'n sekere konstante gestel word, dat die drukkop na links beweeg word, en dat die huidige lyn as die begin van 'n nuwe bladsy beskou word. Hierdie inisialiseringskode kan ook gebruik word as 'n generatorbewerking om 'n "leë" toestel te "skep". So 'n bewerking is egter nie 'n nullêre funksie soos die gewone *Skep*-bewerking nie, maar 'n unêre funksie - dit neem 'n toestel

en "lewer" die toestel in 'n bekende toestand. Die probleem is egter nie onoorkomelik nie - daar moet aanvaar word dat 'n toestel van die tipe wat gedefinieer word, wel beskikbaar is. As 'n *Drukker*-toestel gedefinieer word, met 'n bewerking *InisDrukker*, en daar word aanvaar dat daar 'n drukker d bestaan, sal *InisDrukker(d)* gebruik word, waar *Skep()* normaalweg gebruik is.

Omdat die verskillende bewerkings van 'n abstrakte toestel nou vanuit meer as een bron geïnisieer kan word, is dit nodig om te vereis dat net een bewerking op enige gegewe oomblik enige gegewe voorkoms van 'n toestel gebruik. As 'n gebruiker byvoorbeeld 'n bewerking op 'n voorkoms aktiveer terwyl 'n program besig is met 'n bewerking op dieselfde voorkoms, sal dit nodig wees om te vereis dat die gebruikersgeaktiveerde bewerking eers geïnisieer word wanneer die ander bewerking afgehandel is.

Voorbeelde van aksiomatiese spesifikasies van 'n aantal abstrakte toestelle sal vervolgens gegee word.

UNIVERSITY OF JOHANNESBURG

Voorbeeld 6.6.1

In hierdie voorbeeld word 'n aksiomatiese spesifikasie van 'n toetsbord gegee. Daar word aanvaar dat die gebruiker op enige oomblik 'n toets kan druk en dat die waarde van daardie toets in 'n buffer geplaas sal word tot dit deur die program wat die toestel gebruik, gelees word.

```
Toestel ToetsBord

   Inis       : ToetsBord -> ToetsBord
   DrukToets  : ToetsBord x Karakter -> ToetsBord
   Gereed?    : ToetsBord -> Logies
   VolgKar    : ToetsBord -> Karakter
   LeesKar    : ToetsBord -> ToetsBord

Vir t : ToetsBord, k : Karakter

   Gereed?(Inis(t)) = Vals
   Gereed?(DrukToets(t, k)) = Waar
   VolgKar(DrukToets(t, k)) = As Gereed?(t) Dan VolgKar(t)
                              Anders k
   LeesKar(DrukToets(t, k)) =
           As Gereed?(t) Dan DrukToets(LeesKar(t), k)
           Anders Inis(t)

   VolgKar(Inis(t)) = Fout
   LeesKar(Inis(t)) = Fout
```

In hierdie voorbeeld is *DrukToets* 'n bewerking wat deur die gebruiker uitgevoer word. Die ander bewerkings kan almal deur die program wat die ToetsBord gebruik, geroep word.

Die *ToetsBord* is 'n voorbeeld van 'n toestel wat nie 'n *Skep*-bewerking voorsien nie, maar wel 'n *Inis*-bewerking wat soortgelyk aan 'n *Skep*-bewerking gebruik word – beide lewer 'n voorkoms van 'n toestel in 'n vaste bekende toestand. *Inis* benodig egter 'n voorkoms van die toestel, terwyl dit nie die geval met *Skep* is nie. Daar kan egter aanvaar word dat *ToetsBord* 'n skynbewerking *Skep* voorsien wat aanvanklik deur die bedryfstelsel geroep word om die enkele voorkoms van *ToetsBord* te lewer.

UNIVERSITY OF JOHANNESBURG

In hierdie voorbeeld is die *VolgKar-* en *LeesKar*-bewerkings wat, onderskeidelik, die volgende karakter in die buffer lewer en die voorste karakter uit die buffer verwyder, as aparte bewerkings gegee. In 'n werklike implementering sal LeesKar tipies die volgende karakter lewer en dit terselfdertyd uit die buffer verwyder. In so 'n geval sal die sintaktiese spesifikasie soos volg verander :

```
LeesKar : ToetsBord -> <Karakter, ToetsBord>,
```

terwyl die *VolgKar*-bewerking uit die sintaktiese spesifikasie verwyder word.

Projeksiefunksies sal dan in die aksiomas gebruik word om die benodigde element van die geordende paar te isoleer. As die projeksiefunksies, *P1* en *P2*, soos volg gedefinieer word :

```
P1(<a, b>) = a en P2(<a, b>) = b vir enige a en b,
```

kan die aksiomas

```
VolgKar(DrukToets(t, k)) = As Gereed?(t) Dan VolgKar(t)
                           Anders k
LeesKar(DrukToets(t, k)) =
        As Gereed?(t) Dan DrukToets(LeesKar(t), k)
        Anders Inis(t)
```

vervang word met die aksioma

```
LeesKar(DrukToets(t, k)) =
        As Gereed?(t) Dan
            <P1(LeesKar(t)), DrukToets(P2(LeesKar(t)), k)>
        Anders <k, Inis(t)>
```

Omdat afsonderlike funksies oor die algemeen makliker leesbaar is en verder die onderskeid tussen navraag- en generatorbewerkings duideliker maak, sal afsonderlike funksies ook in die volgende voorbeelde gebruik word.

□

Voorbeeld 6.6.2

In die vorige voorbeeld is aanvaar dat die gebruiker op enige tydstip 'n toets op die toetsbord kon druk. 'n Onderbrekingsfasiliteit van die apparatuur kon dan gebruik word om die gedrukte karakter in die buffer te plaas. Die gebruiker word egter nie altyd toegelaat om 'n bewerking enige tyd uit te voer nie - op sommige toestelle is dit nodig dat die program wat dit gebruik, die toestel eers in 'n sekere toestand moet bring voordat die gebruiker bewerkings op die toestel mag uitvoer. Tipiese gevalle waar dit gebeur, is waar 'n invoertoestel deur die rekenaar gepols word - as daar invoer deur die toestel gedoen is, moet daar gewag word totdat die toestel deur die rekenaar gepols is en die data na die rekenaar oorgedra is voordat die volgende invoer gedoen kan word.

In hierdie voorbeeld word na so 'n toestel gekyk - 'n toetsbord wat in hierdie geval deur die rekenaar gepols word.

```
Toestel ToetsBord

  Inis       : ToetsBord -> ToetsBord
  DrukToets : ToetsBord x Karakter -> ToetsBord
  LeesKar    : ToetsBord -> ToetsBord
  VolgKar    : ToetsBord -> Karakter
  Lees?      : ToetsBord -> Logies

Vir t : ToetsBord, k : Karakter

  LeesKar(DrukToets(t, k)) = Inis(t)
  VolgKar(DrukToets(t, k)) = k
  Lees?(DrukToets(t, k)) = Waar
  Lees?(Inis(t)) = Vals

  VolgKar(Inis(t)) = Fout
  LeesKar(Inis(t)) = Fout
```

UNIVERSITY OF JOHANNESBURG

Volgens hierdie spesifikasie word die gebruiker wel toegelaat om 'n toets te druk voordat die vorige toets gelees is. In so 'n geval word die vorige toets geïgnoreer en die laaste toets wat gedruk is voor die program 'n karakter vanaf die toetsbord lees, is die karakter wat gelewer word.

'n Alternatief sou wees om die aksioma

```
VolgKar(DrukToets(t, k)) = k
```

met die aksioma

```
VolgKar(DrukToets(t, k)) =
  As Lees?(t) Dan Fout
  Anders k
```

te vervang. In hierdie geval word die gebruiker dan nie toegelaat om 'n toets te druk voordat die vorige toets gelees is nie - as hy dit wel doen, kom 'n fout voor wanneer daar weer 'n karakter gelees word. Die implementering van hierdie spesifikasie kan egter probleme lewer, omdat die fout opgemerk moet word en die apparatuur van die toetsbord in staat moet wees om dit te doen aangesien die rekenaar nie bewus is wat by die toestel gebeur tussen opeenvolgende polse nie.

Dikwels word die eerste toets wat gedruk word, gelewer tydens die volgende lees vanaf die toetsbord en nie die laaste toets soos in bostaande spesifikasie nie. As die spesifikasie aangepas word om die eerste toets te lewer, kan 'n bewerking *Tik* bygevoeg word. *Tik* is nou die bewerking wat die gebruiker uitvoer wanneer hy 'n toets druk. Die bewerking wat bygevoeg word, is die volgende :

Tik : ToetsBord x Karakter -> ToetsBord

UNIVERSITY
JOHANNESBURG

met die nuwe aksioma

Tik(t, k) = DrukToets(Inis(t), k)

DrukToets is nou 'n bewerking wat slegs deur *Tik* gebruik word.

□

Voorbeeld 6.6.3

Hierdie voorbeeld toon hoe 'n drukker beskryf kan word. In hierdie spesifikasie sal *SkryfKar* 'n karakter na die drukker stuur; *KryKar* is die bewerking is wat deur die apparatuur uitgevoer word om die volgende karakter wat gedruk moet word, te kry, terwyl *DrukKar* deur die apparatuur uitgevoer word wanneer 'n

karakter fisies gedruk word. *Gereed?* word deur die drukker-apparatuur gebruik word om vas te stel of die volgende karakter wat gedruk moet word, beskikbaar is.

```
Toestel Drukker

  Inis     : Drukker -> Drukker
  SkryfKar : Drukker x Karakter -> Drukker
  DrukKar  : Drukker -> Drukker
  KryKar   : Drukker -> Karakter
  Gereed?  : Drukker -> Logies

Vir d : Drukker, k : Karakter

  Gereed?(Inis(d)) = Vals
  Gereed?(SkryfKar(d, k)) = Waar
  DrukKar(SkryfKar(d, k)) =
          As Gereed?(d) Dan SkryfKar(DrukKar(d), k)
          Anders Inis(d)
  KryKar(SkryfKar(d, k)) =
          As Gereed?(d) Dan KryKar(d)
          Anders k

  KryKar(Inis(d)) = Fout
  DrukKar(Inis(d)) = Fout
```

Die laaste twee beperkings kan ook soos volg geskryf word :

```
KryKar(Inis(d)) = Wag
DrukKar(Inis(d)) = Wag
```

Omdat voorkomste van 'n toestel deur meer as een gebruiker gebruik word (byvoorbeeld 'n program en die drukkerapparatuur),

UNIVERSITY OF JOHANNESBURG

kan 'n ongeldige bewerking wat deur een gebruiker op 'n voorkoms toegepas word, 'n oomblik later wel geldig wees wanneer 'n ander gebruiker 'n ander bewerking op daardie voorkoms toegepas het. In die geval van die drukker kan die drukker byvoorbeeld 'n karakter vra om te druk, voordat die program 'n karakter na die drukker gestuur het. Die drukker hoef nou net te wag tot die program wel 'n karakter druk voor hierdie karakter deur die drukker gedruk kan word. Die woord *Wag* in die aksiomas word gebruik om sulke beperkings, wat dui op toestande waar 'n bewerking nie onmiddellik uitgevoer kan word nie, aan te dui. Omdat net een bewerking op 'n keer toegelaat word om 'n voorkoms van 'n abstrakte toestel te gebruik, is dit nodig dat 'n bewerking wat nie onmiddellik uitgevoer kan word nie, veroorsaak dat daar "buite" die bewerking gewag word totdat dit wel uitgevoer kan word.

□

UNIVERSITY OF JOHANNESBURG

Voorbeeld 6.6.4

Hierdie laaste voorbeeld gee 'n aksiomatiese spesifikasie van 'n skyfaandrywer. 'n Interessante probleem kom voor by skyf-aandrywers wat verwyderbare skywe gebruik : in so 'n geval is die gedrag van die skyfaandrywer afhanklik van die skyf wat in die aandrywer is wanneer die bewerkings van die skyfaandrywer gebruik word. 'n Oplossing vir hierdie probleem is om die skyf, eerder as die skyfaandrywer, as 'n toestel te sien. As so 'n skyf-aandrywer egter bewerkings voorsien om 'n skyf in die aandrywer te laai of om 'n skyf uit die aandrywer te verwyder, sal die skyf en die skyfaandrywer as afsonderlike toestelle gesien moet word.

Die spesifikasie van 'n skyf (-aandrywer) wat in hierdie voorbeeld gegee word, aanvaar dat die datatipe *Sektor* reeds gedefinieer is (dit is maar net 'n skikking van grepe). Daar

word ook aanvaar dat daar 'n spesifieke sektorwaarde is wat deur die inisialiseringsbewerking op elke sektor van die skyf geskryf word - hierdie waarde word deur die *NulSektor*-bewerking van die *Sektor*-toestel gelewer. Laastens word aanvaar dat elke sektor op 'n skyf deur 'n unieke heelgetal identifiseer kan word.

```
Toestel Skyf

  Inis         : Skyf -> Skyf
  SkryfSektor  : Skyf x Sektor x Heel -> Sektor
  LeesSektor   : Skyf x Heel -> Sektor

Vir d : Skyf, s : Sektor, i, j : Heel

  LeesSektor(SkryfSektor(d, s, i), j) =
              As i = j Dan s
              Anders LeesSektor(d, j)
  LeesSektor(Inis(d), j) = NulSektor()
```

□

UNIVERSITY OF JOHANNESBURG

Hoofstuk 7.

7 Data-abstraksie in programmeertale : kort gevallestudies.

- 7.1 Inleiding.
- 7.2 Modula-2.
- 7.3 Ada.
- 7.4 CLU.
- 7.5 Simula.

Inhoud.

1 Oorsig.

2 Inleiding.

3 Objekgeoriënteerde programmering.

4 Abstrakte datatipes.

5 Abstrakte datatipes in programmeertale.

6 Die wiskunde van abstrakte datatipes.

7 Data-abstraksie in programmeertale : kort gevallestudies.

8 Abstrakte toestelle in 'n programmeertaal.

9 Rekordgebaseerde toestelle.

10 Hoër-vlak toestelle : 'n voorbeeld.

11 Implementering van die vertaler.

12 Doeltreffende rekenaargebruik.

A Die programmeertaal PT.

B Verwysings en bibliografie.

-oOo-

7 Data-abstraksie in programmeertale : kort gevallestudies.

7.1 Inleiding.

In hierdie afdeling sal kortliks gekyk word na metodes wat in bestaande programmeertale voorsien word om data-abstraksie te ondersteun.

Ellis Horowitz [Hor84] het die vrae wat oor 'n data-abstraksie fasiliteit van 'n programmeertaal gevra behoort te word, soos volg opgesom :

1) Wat is die sintaktiese vorm van die konstukt?
2) Wat is die strekkingsreëls en en wat is die leeftyd van die objekte wat geskep word?
3) Laat die fasiliteit aanvangs- en finale kodesegmente toe?
4) Is dit moontlik om meer as een voorkoms van 'n datatipe te verkry, dit wil sê, is die fasiliteit 'n tipekonstruktor?
5) Is dit moontlik om die definisie van 'n abstrakte datatipe te parametriseer?
6) Kan data tussen voorkomste gedeel word?

7.2 Modula-2.

Abstrakte datatipes word in Modula-2 met behulp van modules gedefinieer. Modules van Modula-2 lyk soos die modules wat in hoofstuk 5 bespreek is. Die module begin met die opskrif MODULE, wat gevolg word deur die naam van die module. Na die naam van die module word IMPORT- en EXPORT-lyste gegee wat, onderskeidelik, spesifiseer watter identifiseerders van buite die module ook binne die module bekend is, en watter interne identifiseerders van die module ook buite die module bekend is.

7 Data-abstraksie : kort gevallestudies.

Die voorstelling van die abstrakte datatipe wat met 'n Modula-2 module geïmplementeer is, begin bestaan wanneer die prosedure wat die module bevat, geaktiveer word en die bestaan van die voorstelling word beëindig wanneer dié prosedure verlaat word. Die objekte wat aan 'n module behoort, wat nie in 'n prosedure bevat is nie, bestaan vandat die hoofprogram geaktiveer word, totdat die program klaar uitgevoer het.

Modula-2 voorsien aanvangkodesegmente, wat uitgevoer word wanneer die prosedure wat die module bevat, geaktiveer word - dit wil sê, wanneer die leeftyd van die objekte begin.

'n Module is nie 'n tipekonstruktor nie, alhoewel dit gebruik kan word om klasse tipes en dan voorkomste van die klas te definieer, soos dit in hoofstuk 5 beskryf is.

Modules, soos dit deur Modula-2 ondersteun word, maak dit nie moontlik om geparametriseerde abstrakte datatipes te definieer nie (sien egter [Wie85]).

Die moontlikheid van meer as een voorkoms van 'n abstrakte datatipe hang af van die manier waarop die programmeerder die abstrakte datatipe implementeer. Soortgelyk kan data tussen sulke voorkomste gedeel word, as die ontwerp van die programmeerder dit toelaat.

Modula-2 ondersteun ook sogenaamde DEFINITION- en IMPLEMENTATION-modules, wat dit moontlik maak om die inligting wat buite 'n module beskikbaar is, van die werklike implementering te skei. Dit maak ook afsonderlike vertaling van die implementasie van 'n abstrakte datatipe en programmatuur wat die abstrakte datatipe gebruik, moontlik.

UNIVERSITY OF JOHANNESBURG

Meer besonderhede in verband met Modula-2 kan in [Wir82] gevind word.

Aspekte van Modula, die voorganger van Modula-2, word in [Wir77a], [Wir77b] en [Wir77c] bespreek.

7.3 Ada.

Ada voorsien die PACKAGE-konstruksie om abstrakte datatipes te ondersteun. 'n PACKAGE is soortgelyk aan modules, wat vantevore bespreek is.

Ada voorsien ook generiese pakkette. 'n Generiese pakket is 'n pakket met parameters. Om 'n spesifieke voorkoms van so 'n generiese pakket te kry, word nog 'n pakket verklaar as 'n nuwe voorkoms van die generiese pakket en terselfdertyd word die waardes van die parameters vir die nuwe voorkoms gespesifiseer.

Generiese pakkette voorsien dus 'n gerieflike metode om geparametriseerde abstrakte datatipes te implementeer.

[Ich82] en [Fre82] kan vir verdere inligting geraadpleeg word.

7.4 CLU.

CLU is 'n taal wat ontwerp is met 'n verskeidenheid abstraksie-meganismes as oogmerk - CLU voorsien dan ook 'n goed-ontwerpte data-abstraksiefasiliteit.

Die CLU-eenheid wat vir data-abstraksie gebruik word, is die

sogenaamde CLUSTER. 'n CLUSTER word gebruik om 'n klas van objekte te definieer. Die bewerkings wat deur die klas van objekte voorsien word, word saam met die naam en enige parameters van die abstrakte datatipe, in die opskrif van die CLUSTER gegee. Binne die CLUSTER word die voorstelling van 'n voorkoms van die abstrakte datatipe gegee, terwyl die bewerkings (wat natuurlik toegang tot hierdie voorstelling het) ook volledig binne die CLUSTER voorkom.

CLU-veranderlikes kan gesien word as wysers na voorkomste van abstrakte datatipes (of objekte in CLU-terminologie). Abstrakte datatipes in CLU voorsien normaalweg 'n CREATE-bewerking wat gebruik kan word om voorkomste van die datatipe te verkry - die CREATE-bewerking lewer 'n wyser na die objek wat geskep is, wat dan in 'n veranderlike gehou kan word. 'n Objek bly teoreties vir ewig bestaan nadat dit geskep is. In die praktyk word sulke objekte egter outomaties geskrap wanneer daar nie meer na die objek verwys kan word nie, omdat daar nie meer wysers (in veranderlikes) na die objek bestaan nie.

Omdat objekte eksplisiet geskep word met die CREATE-bewerking, kan die CREATE-bewerking gebruik word om enige inisialiseringskode wat deur die abstrakte datatipe benodig word, uitgevoer te kry voor die objek gebruik word. Dit is egter nie moontlik om kode outomaties te laat uitvoer voor die objek geskrap word nie - 'n objek word immers nooit geskrap as die looptydstelsel nie spesiaal daarvoor voorsiening maak nie.

Die CREATE-bewerking van 'n abstrakte datatipe word (normaalweg) so geskryf dat dit 'n onbeperkte aantal kere gebruik kan word. 'n Onbeperkte aantal voorkomste kan dus van 'n enkele datatipe-spesifikasie verkry word.

CLUSTERS laat parameters in hulle definisies toe. Hierdie parameters kan ook tipes wees, sodat dit moontlik is om, byvoorbeeld, 'n stapel van tipe t te definieer, waar t 'n parameter van die stapel is. Wanneer 'n veranderlike van die stapeltipe verklaar word, word dan ook gespesifiseer watter "waarde" t in die spesifieke geval aanneem - t kan byvoorbeeld die waarde "int" (die tipe heelgetal van CLU) aanneem, wat sou beteken dat die veranderlike na stapels van heelgetalle kan wys.

Besonderhede van CLU kan in [Lis77] gevind word.

7.5 Simula.

Simula was die eerste taal wat 'n meganisme voorsien het om objekte saam te groepeer en dit dan met behulp van bewerkings te manipuleer. Simula het dit egter nie ten doel gehad om abstrakte datatipes in die algemeen te ondersteun nie - dit is voorsien om simulasie moontlik te maak.

Waar 'n program 'n werklike situasie simuleer, moet dit voorsiening maak vir objekte wat op enige oomblik kan begin bestaan, vir 'n tydperk bestaan en die simulasie beïnvloed, en daarna mag ophou bestaan - 'n mens kan byvoorbeeld gebore word, 'n bydrae tot sy omgewing lewer, waarna hy mag sterf. Aangesien dit moeilik was om sulke situasies, waar die leeftyd van 'n objek nie bepaal word deur die prosedure waarin dit voorkom nie, maar waar dit eksplisiet deur die program geskep en geskrap word, in konvensionele programmeertale te ondersteun, is hierdie nuwe fasiliteit in Simula voorsien.

Alhoewel abstrakte datatipes dus 'n sterk ooreenkoms met die Simula CLASS het en, volgens Wulf (sien [Hor84], p234), hulle

oorsprong aan die CLASS-konstuksie te danke het, verskil die doel waarvoor dit gebruik word tog.

Hoofstuk 8.

8 Abstrakte toestelle in 'n programmeertaal.

8.1 Inleiding.

8.2 Een-van-'n-soort-toestelle.

8.3 Klasse toestelle.

8.4 Oorlading van operasies.

8.5 Toestelle onder beheer van die bedryfstelsel.

8.6 Dinamiese toestelle.

8.7 Rekordgebaseerde toestelle.

8.8 Uitsonderings.

8.9 Gelyktydigheid.

Inhoud.

1 Oorsig.

2 Inleiding.

3 Objekgeoriënteerde programmering.

4 Abstrakte datatipes.

5 Abstrakte datatipes in programmeertale.

6 Die wiskunde van abstrakte datatipes.

7 Data-abstraksie in programmeertale : kort gevallestudies.

8 Abstrakte toestelle in 'n programmeertaal.

9 Rekordgebaseerde toestelle.

10 Hoër-vlak toestelle : 'n voorbeeld.

11 Implementering van die vertaler.

12 Doeltreffende rekenaargebruik.

A Die programmeertaal PT,

B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

8 Abstrakte toestelle in 'n programmeertaal.

8.1 Inleiding.

Die fasiliteite wat 'n programmeertaal kan voorsien om abstrakte toestelle te ondersteun, kom baie ooreen met die fasiliteite wat deur moderne programmeertale voorsien word om abstrakte datatipes te ondersteun. Die basiese vereiste is dieselfde : die programmeertaal moet 'n meganisme voorsien sodat die implementering van die prosedures en datastrukture wat gebruik word om die abstrakte toestel of abstrakte datatipe te definieer, versteek kan word vir ander prosedures wat die abstrakte datatipe of abstrakte toestel gebruik. Daar moet verkieslik ook 'n meganisme voorsien word sodat meer as een voorkoms van 'n abstrakte datatipe of 'n abstrakte toestel verkry kan word.

Die belangrikste addisionele vereistes wat aan 'n programmeertaal wat abstrakte toestelle ondersteun, gestel word, is die volgende :

1) Daar moet fasiliteite bestaan om sowel een-van-'n-soort toestelle, as toestelle wat meer as een voorkoms van die toestel toelaat, te definieer.

2) 'n Meganisme moet voorsien word om 'n toestel wat sy data in rekords groepeer, maar waar die spesifieke rekord van voorkoms tot voorkoms van die toestel kan verskil, te beskryf.

'n Programmeertaal, PT, is vir hierdie verhandeling ontwikkel om die ondersteuning van abstrakte toestelle deur 'n programmeertaal te illustreer. Die notasie wat gebruik sal word wanneer daar voortaan na abstrakte toestelle verwys word, is die notasie wat

deur PT gebruik word. 'n Volledige spesifikasie van PT word in bylaag A gegee.

8.2 Een-van-'n-soort toestelle.

Een-van-'n-soort toestelle is veral nuttig in die volgende gevalle :

1) As 'n toestel beskryf word, wat definitief net een voorkoms sal hê. Dit kan gebeur as 'n toestel soos 'n drukker beskryf word op 'n rekenaar wat net een drukker kan hanteer. As dit moontlik is om meer as een voorkoms van so 'n drukker in verskillende subprogramme te verklaar, sal dit feitlik onmoontlik wees om die gebruik van die drukker deur die verskillende subprogramme te koördineer. In sommige gevalle is dit wel wenslik om meer as een voorkoms van so 'n fisiese toestel toe te laat. Dit is byvoorbeeld die geval waar die stelselprogrammatuur wat die drukker moet beheer, geskryf word. In so 'n geval sal dit beteken dat dieselfde kode gebruik kan word om nog 'n drukker te beheer wanneer nog 'n (soortgelyke) drukker later by die stelsel gevoeg word. Om dit te bewerkstellig, sal die inligting wat een drukker van die ander drukker onderskei (soos die poort-adresse waaraan dit gekoppel is) parameters van die spesifikasie van die abstrakte toestel moet wees.

2) Wanneer 'n nuwe soort toestel gebruik word. Wanneer die drywerprogram van 'n nuwe soort toestel geskryf word, mag dit moontlik nog nie duidelik wees watter inligting van voorkoms tot voorkoms van die toestel gaan verskil en dus parameters van 'n beskrywing van die klas van hierdie nuwe toestelle moet wees, nie. Dit is makliker om 'n een-van-

'n-soort spesifikasie te verander wanneer ondervinding van die gebruik van die nuwe soort toestel opgedoen word.

Waar daar nie 'n goeie rede bestaan waarom die beskrywing van 'n abstrakte toestel 'n een-van-'n-soort beskrywing moet wees nie, behoort 'n beskrywing van 'n klas van abstrakte toestelle gegee te word. As net een so 'n abstrakte toestel benodig word, word net een voorkoms van die klas van toestelle verklaar. Hierdie gebruik sal veroorsaak dat, indien dit later blyk dat nog soortgelyke toestelle nuttig sal wees, dit slegs nodig sal wees om nog voorkomste van die toestel te verklaar. Die implementering van die vertaler van 'n programmeertaal behoort dus so te wees dat dit nie 'n verskil aan die doeltreffendheid van programme sal maak wat 'n voorkoms van 'n klas toestelle, eerder as een-van-'n-soort toestelle, gebruik nie.

Die module-konstruksie is waarskynlik die eenvoudigste konstruksie wat gebruik kan word om een-van-'n-soort toestelle te implementeer. Die formaat van 'n module, soos dit in die programmeertaal PT gebruik sal word, is soos volg :

```
MODULE [ naam ]
DEF defitem [, defitem ]...
  .
  . ( Liggaam van module )
  .
 EINDE
```

In die geval van 'n een-van-'n-soort toestel, sal die items in die DEF-lys die bewerkings wat die toestel toelaat, wees. Die DEF-lys kan uit een of meer items bestaan.

Die naam van die module is opsioneel. As die naam voorkom, moet

dit gebruik word om items in die DEF-lys te kwalifiseer : as 'n module 'n *Skryf*-bewerking voorsien en die naam van die module M is, sal na die bewerking verwys word as *M.Skryf*. As die module nie 'n naam het nie, word daar slegs na *Skryf* verwys wanneer die *Skryf*-bewerking gebruik word.

Voorbeelde van modules sal later gegee word.

8.3 Klasse toestelle.

Dit gebeur dikwels dat 'n hele aantal soortgelyke toestelle benodig word. In so 'n geval behoort dit nie nodig te wees om meer as een beskrywing van die toestel te gee nie - dit behoort eerder moontlik te wees om 'n klasbeskrywing te gee en dan voorkomste van die klas te verklaar wanneer so 'n toestel benodig word. Die programmeertaal PT voorsien dan ook 'n meganisme om so 'n klas van toestelle te beskryf. Die basiese struktuur van hierdie meganisme is die volgende :

```
TIPEDEF tipenaam [parameters] [basistipe]
DEF defitem [, defitem]...
    .
    . (Beskrywing van abstrakte toestel)
    .
EINDE
```

Die items in die DEF-lys is weer eens die name van die bewerkings wat deur die toestel voorsien word. In die geval van 'n TIPEDEF kontroleer die vertaler dat net bewerkings in die DEF-lys voorkom.

UNIVERSITY OF JOHANNESBURG

'n Voorkoms van die toestel kan nou met behulp van die verklaring

```
VERAND
  voorkoms : tipenaam
```

verkry word.

Voorbeeld 8.3.1
Hierdie voorbeeld illustreer die implementering van die klas *Stapel* in PT.

```
TipeDef Stapel
Def Inis, Op, Af, Vol, Leeg

  Verand
    St : Skik[1:10] Heel
    BoPunt : Heel

  Prosedure Inis(S : DefTipe)
  Begin
    S.BoPunt := 0
  Einde

  Prosedure Op(S : DefTipe, n : Heel)
  Begin
    S.BoPunt := S.BoPunt + 1
    S.St[S.BoPunt] := n
  Einde
```

UNIVERSITY OF JOHANNESBURG

```
    Funksie Af(S : DefTipe) : Heel
    Begin
      S.BoPunt := S.BoPunt - 1
      Waarde is S.St[S.BoPunt+1]
    Einde

    Funksie Vol(S : DefTipe) : Logies
    Begin
      Waarde is S.BoPunt >= 10
    Einde

    Funksie Leeg(S : DefTipe) : Logies
    Begin
      Waarde is S.BoPunt = 0
    Einde

  Einde
```

Die "DefTipe"-parameter wat by elk van die bewerkings voorkom, is natuurlik nodig omdat die bewerkings vir elke voorkoms van 'n toestel van die klas moet kan werk en dus die betrokke toestel kan identifiseer. So 'n DefTipe-parameter word gevolglik by elke roetine van 'n tipedefinisie vereis. PT vereis dat die eerste parameter van elke roetine binne 'n tipedefinisie 'n DefTipe-parameter is.

'n Voorkoms, S, van die klas van stapels, kan nou verkry word deur die verklaring

```
  Verand
    S : Stapel
```

Die program wat S gebruik sal dit tipies soos volg doen :

```
Inis(S)
 .
 .
Op(S, 1)
 .
 .
Op(S, 2)
 .
 .
As Nie Leeg(S) Dan
  Skryf(Af(S))
AsEinde
 .
 .
As Nie Leeg(S) Dan
  Skryf(Af(S))
AsEinde
 .
 .
```

□

Voorbeeld 8.3.2

Hierdie voorbeeld toon hoe 'n meer algemene klas van stapels gedefinieer kan word, wat nie beperk is tot die hantering van heelgetalle soos in die geval van voorbeeld 8.3.1, nie. Die klas neem ook 'n parameter om die maksimum grootte van die stapel te definieer.

UNIVERSITY OF JOHANNESBURG

```
TipeDef Stapel(n : Heel) van TipeMet(:=)

  Def Inis, Op, Af, Vol, Leeg

  Verand
    St : Skik [1:n] van BasisTipe
    BoPunt : Heel

  Prosedure Inis(S : DefTipe)
  Begin
    S.BoPunt := 0
  Einde

  Prosedure Op(S : DefTipe, i : BasisTipe)
  Begin
    S.BoPunt := S.BoPunt + 1
    S.St[S.BoPunt] := i
  Einde

  Funksie Af(S : DefTipe) : BasisTipe
  Begin
    S.BoPunt := S.BoPunt - 1
    Waarde is S.St[S.BoPunt+1]
  Einde

  Funksie Vol(S : DefTipe) : Logies
  Begin
    Waarde is S.BoPunt >= S.n
  Einde
```

```
    Funksie Leeg(S : DefTipe) : Logies
    Begin
      Waarde is S.BoPunt = 0
    Einde

  Einde
```

Die woord "van" in die TipeDef-lyn is die aanduiding dat die tipe waarop die toestel gebaseer word, gespesifiseer sal word wanneer voorkomste van die klas verklaar word. Binne die toestel-spesifikasie word na hierdie tipe verwys as die "BasisTipe". Na die woord "van" word beperkings op hierdie basistipe gelê : in die voorbeeld word vereis dat die basistipe 'n toewysings-bewerking moet voorsien; die toewysingsbewerking is natuurlik nodig want in Op word dit eksplisiet gebruik en waar Af as 'n funksie van hierdie tipe verklaar word, word die toewysings-bewerking implisiet aanvaar.

UNIVERSITY OF JOHANNESBURG

Voorkomste van hierdie stapel kan soos volg verklaar word :

```
  Verand
    SH : Stapel(10) van Heel
    SR : Stapel(20) van Reeel
```

□

Die term <u>datatipe</u> sal voortaan gebruik word om na 'n klas van abstrakte toestelle te verwys. 'n <u>Veranderlike</u> sal na 'n voorkoms van 'n klas van abstrakte toestelle verwys.

8.4 Oorlading van bewerkings.

Aangesien die eerste parameter van elke roetine van 'n tipe-definisie 'n DefTipe-parameter is, is dit maklik om tussen roetines met dieselfde naam, maar wat vir verskillende toestelle gedefinieer is, te onderskei. As S 'n voorkoms van 'n klas van stapels is en T 'n voorkoms van 'n klas van toue, waar beide klasse 'n *Inis*-roetine voorsien, sal *Inis(T)* die *Inis*-roetine wat vir die klas van toue gedefinieer is, roep en *Inis(S)* sal die roetine wat vir die klas van stapels gedefinieer is, roep. In so 'n geval word gesê dat die *Inis*-roetine oorlaai is.

Dit is dikwels wenslik om 'n roetine te kan oorlaai - dit maak dit moontlik om algemene roetines te skryf. Beskou byvoorbeeld die volgende *SkryfString*-roetine :

```
Prosedure SkryfString(Toestel : TipeMet(SkryfKar), L : String)
Verand
  i : Heel

Begin
  Vir i := 1 tot Len(L) Herhaal
    Skryf(Toestel, L[i])
  LusEinde
  SkryfLyn(Toestel)
Einde
```

As *Drukker* nou 'n drukker is en *Stipper* 'n stipper, waar albei *SkryfKar*- en *SkryfLyn*-roetines voorsien, kan dieselfde roetine gebruik word om 'n string op beide die drukker en die stipper te skryf. So sal *SkryfString(Drukker, 'ABC')* die string 'ABC' op die drukker skryf, terwyl *SkryfString(Stipper, 'XYZ')* die string 'XYZ' op die stipper sal skryf.

Oorlaaide roetines maak dit ook moontlik om meer algemene toestelle te definieer. Dit kan gesien word in voorbeeld 8.3.2 waar die basistipe van die stapel enige tipe kan wees wat oor 'n toewysingsbewerking beskik.

Waar 'n roetine oorlaai word, is dit belangrik dat die formaat wat gebruik moet word om die roetine te roep, vir al die verskillende toestelle dieselfde is (in die onderskeie roetines verwys DefTipe natuurlik na die toestelklas waarbinne dit gedefinieer is). Dit is nodig sodat die vertaler reeds teen vertaaltyd kan kontroleer dat 'n "algemene" roetine die oorlaaide roetine korrek gebruik.

Die PT-programmeertaal voorsien die tipe

TIPEMET(roetine [, roetine]...)

om die skryf van "meer algemene" roetines en toestelle toe te laat. 'n Roetine wat so 'n tipe gebruik sal op enige toestel gebruik kan word indien die toestel die bewerkings wat in die TIPEMET-lys gegee word, voorsien.

8.5 Toestelle onder beheer van die bedryfstelsel.

'n Programmeertaal wat abstrakte toestelle ondersteun, behoort ook 'n meganisme te voorsien om toestelle wat onder die beheer van die bedryfstelsel staan, te gebruik. Sulke toestelle moet net soos toestelle wat deur die program self geskep is, gebruik kan word.

Voorbeeld 8.5.1

Hierdie voorbeeld illustreer hoe 'n skyfaandrywer, wat deur die bedryfstelsel beheer word, vanuit 'n PT-program gebruik kan word.

```
TipeDef Skyf
  Def Open, Sluit, LeesSektor, SkryfSektor

  Tipe
    SektorTipe : Skik[1:128] van Kar
    SkyfIdTipe : Heel

  Verand
    SkyfId : SkyfIdTipe

  Prosedure Open(S : DefTipe)
    Ekstern SkfAanVra

  Prosedure Sluit(S : DefTipe)
    Ekstern SkfBevry

  Prosedure LeesSektor(S : DefTipe, Sekt : SektorTipe Uit,
                       SektorNr : Heel In)
    Ekstern SkfLeesSekt

  Prosedure SkryfSektor(S : DefTipe, Sekt : SektorTipe In,
                        SektorNr : Heel In)
    Ekstern SkfSkrfSekt

Einde
```

'n Program wat 'n skyfaandrywer benodig, sal 'n veranderlike van tipe *Skyf* verklaar en dit dan aanvra deur die *Open*-roetine te roep. Daarna sal dit *LeesSektor* en *SkryfSektor* roep om die

UNIVERSITY OF JOHANNESBURG

verlangde lees- en skryfwerk te doen. Wanneer dit die aandrywer nie verder benodig nie, sal dit die bedryfstelsel laat weet deur die *Sluit*-bewerking uit te voer.

Die woord "Ekstern" by 'n roetineverklaring dui aan dat die roetine in 'n biblioteek voorkom en deur 'n binder by die res van die program wat hierdie tipebeskrywing gebruik, gevoeg word. So sal die eksterne roetine *SkfAanVra* geroep word wanneer die *Open*-roetine vir bogenoemde *Skyf*-toestel uitgevoer word. Die parameters wat aan die *Open*-roetine gestuur word, sal net so aan die *SkfAanVra*-roetine gestuur word, sodat laasgenoemde roetine die nommer van die skyfaandrywer wat aan die program beskikbaar gestel word, in die *SkyfId*-veranderlike van die betrokke voorkoms van die *Skyf*-toestel kan plaas. Die ander bedryfstelselroetines sal soortgelyk van die genoemde parameters voorsien word, sodat hulle die betrokke skyfaandrywer, baan en sektor kan gebruik.

□

UNIVERSITY
OF
JOHANNESBURG

8.6 Dinamiese toestelle.

Die term dinamiese_toestel sal gebruik word om na 'n abstrakte toestel te verwys wat 'n *Skep*-bewerking voorsien. Die *Skep*-bewerking kan dan tydens looptyd geroep word om nog 'n voorkoms van so 'n toestel te verkry. In baie programmeertale word voorkomste van abstrakte datatipes altyd op so 'n manier verkry. Om redes wat vantevore gegee is, is besluit dat abstrakte toestelle nie altyd op so 'n manier geskep kan word nie. Dit behoort egter moontlik wees om sulke dinamiese toestelle te kan beskryf.

Daar is besluit om nie 'n spesiale konstruksie in PT te voorsien

om dinamiese toestelle te ondersteun nie aangesien gewone abstrakte toestelle en modules voldoende ondersteuning bied - die volgende voorbeeld illustreer dit :

Voorbeeld 8.6.1

Hierdie voorbeeld toon aan hoe 'n dinamiese toestel in PT gedefinieer kan word. Dit aanvaar dat daar 'n datatipe "Stapel" (soos in voorbeeld 8.3.1) gedefinieer is.

Die *StapelSkep*-bewerking kan gebruik word om 'n voorkoms van 'n stapel te kry - as S 'n veranderlike van tipe *StapelTipe* is, sal die funksieroep StapelSkep(S) 'n voorkoms van 'n stapel kry as dit moontlik is; die funksie sal dan die waarde *Waar* lewer om aan te dui dat dit suksesvol was. As dit nie moontlik is om nog 'n voorkoms van 'n stapel te skep nie, sal *StapelSkep* die waarde *Vals* lewer.

UNIVERSITY OF JOHANNESBURG

```
Module
  Def
    StapelSkep, StapelSkrap, StapelTipe, StapelOp,
    StapelAf, StapelVol, StapelLeeg, StapelsInis

  Tipe
    StapelTipe = Heel

  Verand
    Stapels : Skik[1:10] van Stapel
    Beskikbaar : Skik[1:10] van Logies

  Prosedure StapelsInis;
  Verand
    i : Heel
```

```
Begin
  Vir i := 1 tot 10 herhaal
    Beskikbaar[i] := Waar
  LusEinde
Einde

Funksie StapelSkep(n : Heel Uit) : Logies
Verand
  i : Heel

Begin
  i := 0
  Solank i < 10 herhaal
    i := i + 1
  TotDat Beskikbaar[i]
  As i <= 10 dan
    Beskikbaar[i] := Vals
    Inis(Stapels[i])
    Waarde is Waar
  Anders
    Waarde is Vals
Einde

Prosedure StapelSkrap(n : Heel In)
Begin
  Beskikbaar[i] := Waar
Einde

Prosedure StapelOp(S : StapelTipe In, i : Heel)
Begin
  Op(Stapels[S], i)
Einde
```

UNIVERSITY OF JOHANNESBURG

```
    Funksie StapelAf(S : StapelTipe In) : Heel
    Begin
      Waarde is Af(Stapels[S])
    Einde

    Funksie StapelVol(S : StapelTipe In) : Logies
    Begin
      Waarde is Vol(Stapels[S])
    Einde

    Funksie StapelLeeg(S : StapelTipe In) : Logies
    Begin
      Waarde is Leeg(Stapels[S])
    Einde

  Einde
```

□

In die voorafgaande voorbeeld is 'n skikking van die verlangde toestel verklaar en 'n element van die skikking is gebruik om 'n voorkoms van die toestel wat deur 'n *Skep*-bewerking aangevra is, te hou. Hierdie metode het natuurlik een groot nadeel : die grootte van die skikking bepaal die maksimum aantal voorkomste wat van die toestel aangevra kan word. Verder word die hoeveelheid geheue wat nodig is om die maksimum aantal voorkomste van die toestel voor te stel, deurentyd deur die skikking beslaan.

'n Beter oplossing sou natuurlik moontlik wees, as dit moontlik was om die nodige geheue aan te vra wanneer dit benodig word en weer vry te stel wanneer dit nie verder benodig word nie. Om

dit toe te laat, kon PT 'n wysertipe soortgelyk aan die van Pascal toelaat. As *Stapel* dan 'n toestelklas is, kan 'n veranderlike *StapelW*, soos volg verklaar word :

```
Verand
  StapelW : ^Stapel
```

en dan kon 'n voorkoms van die stapel verkry word deur die bewerking

```
Skep(StapelW)
```

uit te voer. Daar kan dan na die voorkoms van die stapel verwys word as *StapelW^*. Die voorkoms van die stapel kan geskrap word deur die bewerking

```
Skrap(StapelW)
```

uit te voer. *Skep* en *Skrap* is bewerkings wat vir alle wysertipes voorsien word en dus nie deur die definisie van 'n toestel gegee hoef te word nie.

Wyserveranderlikes is nie in die huidige weergawe van PT ingesluit nie. Dit is egter vir die gebruiker (van die huidige weergawe waar dit op enige bedryfstelsel gebruik word, wat 'n roetine voorsien om geheue vanaf die bedryfstelsel aan te vra en aan die bedryfstelsel terug te besorg) moontlik om wyser-veranderlikes te definieer. Wyserveranderlikes behoort egter in 'n volgende weergawe van PT ingesluit te word.

UNIVERSITY OF JOHANNESBURG

8.7 Rekordgebaseerde toestelle.

Die grootste verskil tussen abstrakte datatipes, soos dit tans deur bestaande programmeertale voorsien word, en abstrakte toestelle soos dit in PT ondersteun word, is die fasiliteit om 'n abstrakte toestel te verklaar wat op 'n rekord gebaseer is en, wanneer 'n voorkoms van die toestel verklaar word, te spesifiseer watter rekord vir die spesifieke voorkoms gebruik moet word. Sulke rekordgebaseerde toestelle sal volledig in die volgende hoofstuk bespreek word.

8.8 Uitsonderings.

Uitsonderings word nie in PT ondersteun nie. Daar word van die gebruiker van 'n toestel verwag om die navraagbewerkings (wat voorsien behoort te word) te gebruik om vas te stel of bewerkings (wat nie altyd toegelaat word nie) wel toegelaat word wanneer dit benodig word. So behoort 'n roetine wat 'n item vanaf 'n stapel wil verwyder (en nie seker is of daar nog items op die stapel is nie) 'n bewerking uit te voer wat uitvind of daar nog items op die stapel is.

As alternatief kan bewerkings 'n logiese parameter, *Suksesvol*, voorsien wat onderskeidelik, die waardes *Waar* en *Vals* kan aanneem, om aan te dui of die bewerking suksesvol was of nie.

Daar is heelwat rekenaarwetenskaplikes wat voel dat 'n programmeertaal voorsiening vir sulke onvoorsiene situasies moet maak deur 'n uitsonderingsmeganisme te voorsien - dit is dan ook waarom nuwer programmeertale soos Ada en CLU (sien [Lis77] en [Ich82]) uitsonderingsmeganismes voorsien. Daar is egter ook mense soos Wirth [Pou85] wat voel dat uitsonderings 'n onding is.

Aangesien uitsonderings nie nodig is nie en, moontlik, selfs nie wenslik is nie, maak PT geensins daarvoor voorsiening nie.

8.9 Gelyktydigheid.

Toestelle verskil natuurlik ook van abstrakte datatipes omdat dit dikwels deur meer as een proses gelyktydig gebruik kan word - 'n skyfaandrywer kan byvoorbeeld op 'n gegewe tydstip deur 'n hele aantal programme gebruik word. Selfs in die geval van 'n enkel-taak rekenaar wat oor 'n drukker beskik, sal beide die program wat deur die rekenaar uitgevoer word, sowel as die drukker-apparatuur die drywerprogram van die drukker (gelyktydig) gebruik.

In sommige gevalle is dit maklik om hierdie gelyktydigheid te beheer. In die geval van die enkeltaak rekenaar met die drukker kan daar 'n meester-slaaf verhouding bestaan tussen die dele wat die drywerprogram gebruik - die slaaf sal wag terwyl die meester werk; daarna sal die meester beheer aan die slaaf oordra en wag terwyl die slaaf werk; wanneer die slaaf sy werk afgehandel het, sal die slaaf weer beheer aan die meester oordra.

Daar is egter gevalle waar die presiese oomblikke wanneer die gebruikers van 'n toestel die toestel gaan benodig, nie voorspelbaar is nie. In so 'n geval kan die gebruikers gesinchroniseer word deur net een gebruiker op 'n gegewe oomblik toe te laat om 'n spesifieke toestel te gebruik. Omdat al die bewerkings wat die toestel mag gebruik, binne 'n toesteldefinisie saamgroepeer is, is dit relatief maklik om onderlinge uitsluiting te bewerkstellig - onderlinge uitsluiting kan presies soos in die geval van monitors (sien [Bri73] en [Hoa74]) hanteer word.

8 Abstrakte toestelle in 'n programmeertaal.

PT maak nie voorsiening vir gelyktydigheid nie, alhoewel die taal so ontwerp is dat dit redelik maklik, bygevoeg kan word.

Hoofstuk 9.

9 Rekordgebaseerde toestelle.
- 9.1 Rekords.
- 9.2 Toestelle en rekords.
- 9.3 Rekordgebaseerde toestelle.
- 9.4 Attribute.
- 9.5 Rekords en skikkings.

Inhoud.

1 Oorsig.
2 Inleiding.
3 Objekgeoriënteerde programmering.
4 Abstrakte datatipes.
5 Abstrakte datatipes in programmeertale.
6 Die wiskunde van abstrakte datatipes.
7 Data-abstraksie in programmeertale : kort gevallestudies.
8 Abstrakte toestelle in 'n programmeertaal.
9 Rekordgebaseerde toestelle.
10 Hoër-vlak toestelle : 'n voorbeeld.
11 Implementering van die vertaler.
12 Doeltreffende rekenaargebruik.
A Die programmeertaal PT.
B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

9 Rekordgebaseerde toestelle.

9.1 Rekords.

Rekords was aanvanklik deel van 'n gerieflike model van die handmetodes, wat gebruik is om inligting te organiseer. In 'n handstelsel word inligting oor 'n enkele onderwerp gewoonlik op 'n kaart of bladsy gehou. So kan 'n winkel die besonderhede van elk van sy klante op 'n (rekord-) kaart hou en die kaarte dan saam in 'n lêer bêre. Soortgelyk kan hulle al die inligting oor elk van die produkte wat hulle verkoop, op 'n (rekord-) kaart hou en hierdie kaarte dan in 'n lêer saamgroepeer. Die rekord word nou op 'n soortgelyke manier op 'n rekenaar gebruik : al die inligting in verband met 'n klant of in verband met 'n produk word in 'n rekord gehou. Hierdie rekords kan dan in 'n rekenaarlêer saamgroepeer word.

Daar is gou vasgestel dat rekords oor die algemeen 'n gerieflike metode voorsien om onderdele van 'n datastruktuur wat saam hoort, saam te groepeer. So kan die reële en imaginêre dele van 'n komplekse getal in 'n rekord saamgroepeer word, wat dan as 'n eenheid 'n komplekse getal vorm. In 'n bedryfstelsel kan die inligting wat 'n proses beskryf ook in 'n rekord ('n "proses-beheerblok") saamgroepeer word.

Benewens die voordeel wat deur hierdie logiese saamgroepering van items wat saam hoort, gegee word, het die feit dat hierdie samehorende items gewoonlik ook saamgroepeer is in die interne voorstelling van die rekord, dit ook moontlik gemaak dat sekere bewerkings hierdie rekord as 'n eenheid kon sien. So was dit byvoorbeeld moontlik om al die inligting wat in 'n rekord bevat is, met 'n enkele skryfbewerking op 'n lêer te skryf.

9.2 Toestelle en rekords.

In sommige gevalle is 'n abstrakte toestel (of ook 'n abstrakte datatipe) 'n veralgemening van 'n rekord. Die geval waar die komplekse getal as 'n rekord met twee velde - een vir die reële gedeelte en een vir die imaginêre gedeelte - voorgestel word, is tipies. Dit is duidelik dat dit beter is om komplekse getalle as abstrakte toestelle of datatipes te sien - die presiese voorstelling van die komplekse getal word binne die toestel versteek en kan slegs deur die bewerkings wat deur die abstrakte toestel voorsien word, gebruik word.

Abstrakte toestelle kan rekords egter nie in alle gevalle vervang nie. Dit is nog steeds gerieflik om inligting in verband met 'n enkele onderwerp in 'n rekord saam te groepeer. Dit is byvoorbeeld gerieflik om die inligting oor 'n produk (sê kode, prys en verskaffer) in so 'n rekord saam te groepeer - dit is nie nodig (of wenslik) om die inhoud van hierdie rekord te omhul in 'n abstrakte toestel nie. Verder is daar nie "spesiale" bewerkings vir hierdie produkrekord nodig nie - die bewerkings wat benodig word, is die bewerkings wat oor die algemeen deur alle rekords benodig word : toewysing, lees en skryf van rekords en toegang tot die afsonderlike velde van die rekord.

Die *Van*-gedeelte van 'n *TipeDef*-spesifikasie in PT voorsien dikwels voldoende ondersteuning vir rekords. As die abstrakte toestel *Lêer* reeds gedefinieer is, en *ProdukRekord* 'n rekordtipe is, sal die verklaring

```
Verand
   P : Leer van ProdukRekord
```

'n voorkoms van 'n lêer van hierdie rekord verklaar. In hierdie geval word *ProdukRekord* dus net soos enige ander datatipe gesien – vir die definisie van die *Lêer* toestel is enige tipe (wat 'n skryfbewerking voorsien) aanvaarbaar.

Dit is egter nie altyd voldoende vir 'n abstrakte toestel om 'n rekord as net nog 'n tipe te behandel nie. Dit gebeur byvoorbeeld dikwels dat daar 'n spesiale betekenis aan sekere velde van 'n rekord geheg word. So kan sommige velde van 'n rekord wat in 'n databasis gehou word, as sleutels (om die rekord vinnig te onttrek) gebruik word. Die abstrakte toestel *DataBasis* sal dus spesifiseer dat een of meer velde van 'n rekord waarvoor 'n voorkoms van *DataBasis* verklaar word, sleutels moet wees. Verder sal die bewerkings van *DataBasis* die sleutelvelde van die rekord waarop die betrokke voorkoms van *DataBasis* gebaseer is, moet kan gebruik.

9.3 Rekordgebaseerde toestelle.

Rekordgebaseerde toestelle kan die onderliggende rekord op twee spesiale maniere gebruik :

1) Die bewerkings van die toestel kan toegang verkry tot gespesifiseerde velde (byvoorbeeld sleutelvelde); en

2) Die bewerkings van die toestel kan toegang verkry tot alle velde van die rekord wat van 'n gespesifiseerde tipe is.

Die gebruik van velde van 'n gespesifiseerde tipe kan byvoorbeeld nuttig wees in 'n *SkryfRekord*-bewerking, wat al die velde van die rekord wat van 'n tipe is wat 'n *Skryf*-bewerking voorsien, op die terminaalskerm sal skryf.

Soms is daar 'n beperking op die maksimum of minimum aantal velde wat uitgesonder kan word vir spesiale behandeling : die databasisbeheerstelsel kan byvoorbeeld vereis dat daar minstens een en hoogstens agt sleutelvelde mag voorkom. Dit mag ook wees dat hierdie spesiale velde net van sekere tipes mag wees. Die programmeertaal behoort dit moontlik te maak om sulke beperkings af te dwing. PT maak hiervoor voorsiening.

9.4 Attribute.

In PT kan 'n veld gemerk word deur 'n attribuut daarmee te assosieer. So sal die *DataBasis*-toestel spesifiseer dat *Sleutel*-attribute toegelaat word vir die velde van die onderliggende rekord. As 'n *DataBasis* nou benodig word om Skolier-Rekords te hou, waar 'n *SkolierRekord* soos volg verklaar is

```
SkolierRekord : Rekord
  Nommer : Heel
  Naam : String(30)
  Standerd : Heel
RekEinde
```

en waar *Nommer* en *Naam* sleutels is, sal die *Sleutel*-attribuut met *Nommer* en *Naam* geassosieer word. Daar is twee moontlike maniere waarop dit gedoen kan word. Die eerste is soos volg :

```
Verand
   DB : DataBasis van Rekord
                               Nommer : Heel Sleutel
                               Naam : String(30) Sleutel
                               Standerd : Heel
                            RekEinde
```

Omdat dit nie altyd wenslik is om die struktuur van die rekord te beskryf wanneer 'n voorkoms van 'n toestel wat op die rekord gebaseer is, verklaar word, nie, kan die attribute ook in 'n (attribuut-) lys na die verklaring van die rekord gegee word. Dit word soos volg gedoen :

```
Tipe
   SkolierRekord : Rekord
      Nommer : Heel
      Naam : String(30)
      Standerd : Heel
   RekEinde
```

```
Verand
   DB : DataBasis van SkolierRekord
          Sleutel(Nommer, Naam)
```

Laasgenoemde manier is dan ook hoe attribute in PT met velde van 'n rekord geassosieer word.

Daar is in die vorige afdeling genoem dat dit soms wenslik is om al die velde van 'n rekord wat van 'n sekere tipe is te kan bereik. Om hierdie rede laat PT implisiete attribute toe. 'n Implisiete attribuut is 'n attribuut wat outomaties met al die velde wat aan die gespesifiseerde vereistes voldoen, geassosieer word. As *DataBasis* die implisiete attribuut *Skryfbaar* definieer

vir alle velde van tipe

TipeMet(Skryf)

sal die attribuut *Skryfbaar* met al die velde van 'n *SkolierRekord* wat na 'n bewerking van *DataBasis* gestuur word, geassosieer word, sonder dat *Skryfbaar* by die verklaring van *DB* hierbo genoem word. As 'n attribuut *H* soortgelyk implisiet met alle velde van tipe *Heel* geassosieer word, sal dit in hierdie geval net met *Nommer* en *Standerd* geassosieer word.

Wanneer 'n bewerking van die abstrakte toestel nou 'n veld met die attribuut van belang wil gebruik, hoef hy net die naam van die attribuut te gee, gevolg deur 'n indeks wat sê watter veld met die attribuut benodig word. Laat *LeesItem* byvoorbeeld 'n bewerking van *DataBasis* wees, waar *LeesItem* soos volg verklaar is :

Prosedure LeesItem(D : DefTipe In Uit, Rek : BasisTipe In)

In hierdie verklaring is BasisTipe die onderliggende rekordtipe waarop die abstrakte toestel gebaseer is. As *Skolier* nou 'n veranderlike van tipe *SkolierRekord* is, en *LeesItem* word soos volg geroep :

LeesItem(DB, Skolier)

dan kan daar binne *LeesItem* na *Rek.Sleutel[1]* verwys word om die *Nommer*-veld van die *Skolier*-rekord te gebruik en na *Rek.Sleutel[2]* om die *Naam*-veld van die *Skolier*-rekord te gebruik. Soortgelyk sal *Rek.Skryfbaar[1]*, *Rek.Skryfbaar[2]* en *Rek.Skryfbaar[3]* onderskeidelik met *Nommer*, *Naam* en *Standerd* geassosieer word.

9.5 Rekords en skikkings.

Dit is dikwels vir 'n program nodig om inligting in verband met die onderskeie velde van 'n rekord aan te teken. Dit sal tipies die geval wees waar 'n program moet onthou watter velde van 'n rekord reeds geldige waardes bevat. PT maak dit moontlik om hierdie en soortgelyke inligting in verband met rekordvelde te hou, deur dit moontlik te maak om skikkings te verklaar wat deur rekordvelde "geïndekseer" word. 'n Skikking van logiese items, met een item per veld van 'n rekordtipe r, kan byvoorbeeld verkry word met die volgende verklaring :

```
Verand
  Vlae : Skik [1:#r] van Logies
```

Die konstante "#r" is die aantal velde van die rekordtipe r. Op 'n soortgelyke wyse sal "#r.v" die nommer van die veld v binne die rekordtipe r wees – die velde van 'n rekord word genommer vanaf die begin van die rekord met een die nommer van die eerste veld.

Die voorafverklaarde funksie *VeldNr* kan gebruik word om die veldnommer van 'n veld waarmee 'n spesifieke attribuut geassosieer word, te verkry.

Voorbeeld 9.5.1
Aanvaar dat *DataBasis* soos in afdeling 9.4 gedefinieer is. Nou sal die funksieroep

```
VeldNr(Rek.Sleutel[1])
```

binne die prosedure *LeesItem*, die waarde 1 lewer, omdat die eerste veld van die *SkolierRekord* ook die eerste sleutel is.

□

'n Uitgebreide voorbeeld van 'n hoër-vlak toestel wat op rekords gebaseer is, sal in die volgende hoofstuk gegee word.

UNIVERSITY
OF
JOHANNESBURG

Hoofstuk 10.

10 Hoër-vlak toestelle : 'n voorbeeld.

- 10.1 Inleiding.
- 10.2 'n Beskrywing van die voorbeeld.
- 10.3 Die voorbeeld.
- 10.4 Gebruik van *DataBasis*.
- 10.5 Moontlike uitbreidings aan die voorbeeld.

Inhoud.

1 Oorsig.
2 Inleiding.
3 Objekgeoriënteerde programmering.
4 Abstrakte datatipes.
5 Abstrakte datatipes in programmeertale.
6 Die wiskunde van abstrakte datatipes.
7 Data-abstraksie in programmeertale : kort gevallestudies.
8 Abstrakte toestelle in 'n programmeertaal.
9 Rekordgebaseerde toestelle.
10 Hoër-vlak toestelle : 'n voorbeeld.
11 Implementering van die vertaler.
12 Doeltreffende rekenaargebruik.
A Die programmeertaal PT.
B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

10 Hoër-vlak toestelle : 'n voorbeeld.

10.1 Inleiding.

In hierdie hoofstuk sal 'n volledige voorbeeld gegee word om die gebruik van abstrakte toestelle te illustreer. Die voorbeeld sal die krag van abstrakte toestelle wat wel van fisiese toestelle gebruik maak, maar wat deur 'n aantal programmatuurlae van die apparatuur geskei is, aantoon. Die voorbeeld is in PT geskryf.

10.2 'n Beskrywing van die voorbeeld.

Hierdie voorbeeld definieer 'n *DataBasis*-toestel wat van twee ander toestelle gebruik maak : 'n *Skerm*- en 'n *Isam*-toestel.

Die *Skerm*-toestel aanvaar dat skerms (of vorms) vooraf deur 'n skermredigeerder opgestel word. Wanneer die skerm gedefinieer word, word al die datavelde wat op die skerm kan voorkom beskryf - die posisie en lengte van die velde word vasgestel en met elke veld word 'n nommer geassosieer. Hierdie nommer sal later gebruik word om die veld te identifiseer. Daar word ook aanvaar dat 'n aantal roetines om sulke skerms te manipuleer reeds beskikbaar is - hierdie roetines sluit opdragte in om 'n vooraf-opgestelde skerm op die rekenaarkonsole (of terminaalskerm) te vertoon, asook roetines om data na 'n spesifieke veld op die skerm te skryf of om die gebruiker toe te laat om 'n spesifieke veld in te vul en dan die data wat die gebruiker ingevul het, in te lees.

Hierdie voorbeeld aanvaar dat die veld van die skerm met nommer n met die n-de veld van die rekord waarvoor die skerm gebruik gaan

word ooreenkom. Verder aanvaar die voorbeeld dat daar 'n veld nommer nul op die skerm voorkom - hierdie veld word vir boodskappe gebruik.

Die *Isam*-toestel is 'n gewone indeks-sekwensiële lêer. Dit voorsien bewerkings om 'n rekord na die lêer te skryf, data daarvan te lees en om 'n spesifieke rekord te skrap. Een of meer van velde van die rekordtipe waarvoor die lêer gedefinieer is, word as sleutels gebruik. Die betrokke rekord word in elke geval met behulp van een van hierdie sleutels opgespoor. In hierdie voorbeeld word aanvaar dat enige van die sleutels 'n rekord uniek kan identifiseer; in die werklikheid word daar gewoonlik voorsiening gemaak vir sleutel waar dieselfde sleutel-waarde in meer as een rekord mag voorkom - sien 10.5.

Ter wille van eenvoud is daar aanvaar dat 'n verskeidenheid ander roetines (ekstern) beskikbaar is. Die doel van die roetines blyk uit hulle name.

UNIVERSITY OF JOHANNESBURG

10.3 Die voorbeeld.

```
TipeDef DataBasis van Rekord RekEinde
  Attr Sleutel
         Tipes(TipeMet(NaStr, VanStr, Niks))
         Minstens (1)
         Hoogstens(8)
       Opsioneel
         Tipe (TipeMet(NaStr, VanStr, Niks))
       Skryfbaar soos Skerm
         Tipe (TipeMet(NaStr, VanStr, Niks))
         Implisiet
```

```
Def
  OpenDB, SluitDB, DbBywerk, DbVertoon, DbSkrap, DbInvoeg

TipeDef Skerm van Rekord RekEinde
  Attr Skryfbaar soos DataBasis,
       Opsioneel soos DataBasis

Def
  OpenSkerm, SluitSkerm, LeesSkerm, SkryfSkerm,
  LeesVeld, SkryfVeld, SkryfBoodskap

Prosedure LeesVeld(VeldNr : Heel In, Str : String Uit)
  Ekstern LeesVeld

Prosedure SkryfVeld(VeldNr : Heel In, Str : String In)
  Ekstern SkryfVeld
```

```
Prosedure VertoonSkerm(Naam : String In)
  Ekstern VertoonSkerm

Prosedure MaakSkermSkoon
  Ekstern MaakSkermSkoon

Prosedure LeesKar(K : Kar)
  Ekstern LeesKar

Prosedure OpenSkerm(S : DefTipe, Naam : String In)
Begin
  MaakSkermSkoon
  VertoonSkerm(Naam)
Einde
```

```
Prosedure SluitSkerm(S : DefTipe)
Begin
  MaakSkermSkoon
Einde

Prosedure LeesSkerm(S : DefTipe, R : BasisTipe)
Verand
  i : Heel
  Str : String(80)
  Verplig : Skik[1:#BasisTipe] van Logies

Begin
  Vir i := 1 tot BoGrens(R.Skryfbaar) herhaal
    Verplig[VeldNr(R.Skryfbaar[i])] := Waar
  LusEinde
  Vir i := 1 tot Bogrens(R.Opsioneel) herhaal
    Verplig[VeldNr(R.Opsioneel[i])] := Vals
  LusEinde
  i := 1
  Solank i <= BoGrens(R.Skryfbaar) herhaal
    LeesVeld(i, Str)
    As Str <> "" dan
      R.Skryfbaar[i] := Str -> VanStr
      i := i + 1
    AsAnders nie Verplig[i] dan
      i := i + 1
    AsEinde
  LusEinde
Einde
```

```
Prosedure SkryfSkerm(S : DefTipe, R : BasisTipe)
Verand
  i Heel

Begin
  Vir i := 1 tot BoGrens(R.Skryfbaar) herhaal
    SkryfVeld(i, R.Skryfbaar[i] -> NaStr)
  LusEinde
Einde

Prosedure SkryfBoodskap(S : DefTipe, Boodskap : String In)
Verand
  K : Kar

Begin
  SkryfVeld(0, Boodskap)
  LeesKar(K)
Einde

Einde % Skerm


TipeDef ISAM van Rekord RekEinde
  Attr
    Sleutel soos DataBasis

Def
  SkryfIsam, LeesIsam, OpenIsam, SluitIsam, SkrapIsam

Verand
  Nr : Heel    % Word gebruik om die nommer van die betrokke Isam-
               % leer te hou
```

UNIVERSITY OF JOHANNESBURG

```
Funksie OpenIsamLeer(IsamNr : Heel Uit, Naam : String In)
        : String(80)
  Ekstern OpenIsamLeer
  % Lewer lee string as leer suksesvol geopen,
  % lewer foutboodskap andersins

Prosedure SluitIsamLeer(IsamNr : Heel In)
  Ekstern SluitIsamLeer

Funksie SkrIsamRek(IsamNr : Heel In, Rek : TipeMet(:=) In)
        : String(80)
  Ekstern SkrIsamRek
  % Lewer lee string as rekord suksesvol geskryf is,
  % lewer foutboodskap andersins

Funksie KryIsamRek(IsamNr : Heel, Rek : TipeMet(:=) Uit,
                   SleutelNr : Heel In) : String(80)
  Ekstern KryIsamRek
  % Lewer lee string as rekord suksesvol gelees is,
  % lewer foutboodskap andersins

Funksie SkrapIsamRek(IsamNr : Heel, Rek : TipeMet(:=) Uit,
                     SleutelNr : Heel In) : String(80)
  Ekstern SkrapIsamRek
  % Lewer lee string as rekord suksesvol geskrap is,
  % lewer foutboodskap andersins
```

UNIVERSITY OF JOHANNESBURG

```
Prosedure SkryfIsam(D : DefTipe, R : BasisTipe,
                    Fout : String Uit)
Verand
  i : Heel

Begin
  Fout := SkrIsamRek(D.Nr, R)
Einde

Prosedure LeesIsam(D : DefTipe, R : BasisTipe, SleutelNr : Heel,
                   Fout : String Uit)
Begin
  Fout := LeesIsamRek(D.Nr, R, i)
Einde

Prosedure SkrapIsam(D : DefTipe, R : BasisTipe, SleutelNr : Heel,
                    Fout : String Uit)
Begin
  Fout := SkrapIsamRek(D.Nr, R, i)
Einde

Prosedure OpenIsam(D : DefTipe, Naam : String In,
                   Fout : String Uit)
Begin
  Fout := OpenIsamLeer(D.Nr, Naam)
Einde

Prosedure SluitIsam(D : DefTipe)
Begin
  SluitIsamLeer(D.Nr)
Einde

Einde % Isam
```

UNIVERSITY OF JOHANNESBURG

```
% Die veranderlikes wat deur DataBasis benodig word, word nou
% verklaar.

Verand
  I : Isam van BasisTipe
        Sleutel soos BasisTipe
  S : Skerm van BasisTipe
        Opsioneel soos BasisTipe
        Skryfbaar soos BasisTipe

Prosedure VindRekord(D : DefTipe, R : BasisTipe Uit,
                      SNr : Heel Uit)
Verand
  Gevind, Ingevul : Logies

Begin
  SNr := 1
  Gevind := Vals
  Herhaal
    LeesVeld(VeldNr(D.Sleutel[SNr]), Str)
    As Str = "" dan
      Ingevul := Vals
    Anders
      Ingevul := Waar
      NaStr(D.Sleutel[SNr], Str)
    AsEinde
    As Ingevul dan
      LeesIsamRek(D.I, R, SNr, Gevind)
    Einde
    As nie Ingevul dan
      SNr := (SNr mod BoGrens(D.Sleutel)) + 1
    AsAnders nie Gevind dan
```

```
      SkryfBoodskap("Rekord nie gevind nie - druk enige toets.")
    AsEinde
  Totdat Gevind
Einde

Prosedure OpenDb(D : DefTipe, SkermNaam : String In,
                 IsamNaam : String In, Fout : String Uit)
Begin
  OpenIsam(D.I, IsamNaam, Fout)
  VertoonSkerm(D.S, SkermNaam)
Einde

Prosedure SluitDb(D : DefTipe)
Begin
  SluitIsam(D.I)
  SluitSkerm(D.S)
Einde
```

UNIVERSITY
OF
JOHANNESBURG

```
Prosedure DbBywerk(D : DefTipe)
Verand
  R : BasisTipe
  Fout : String(80)

Begin
  Herhaal
    VindRekord(D, R)
    LeesSkerm(D.S, R)
    SkryfIsam(D.I, R, Fout)
    As Fout <> "" dan
      SkryfBoodskap(D.S, Fout)
    AsEinde
  TotDat Fout = ""
Einde
```

```
Prosedure DbSkrap(D : DefTipe)
Verand
  R : BasisTipe
  Fout : String(80)

Begin
  Herhaal
    VindRekord(D, R)
    SkryfSkerm(D.S, R)
    SkryfBoodskap(D.S, "Druk enige toets om die rekord te skrap.")
    SkrapIsam(D.I, R, Fout)
    As Fout <> "" dan
      SkryfBoodskap(D.S, Fout)
    AsEinde
  TotDat Fout = ""
Einde
```

```
Prosedure DbVertoon(D : DefTipe)
Verand
  R : BasisTipe

Begin
  VindRekord(D, R)
  SkryfSkerm(D.S, R)
  SkryfBoodskap(D.S, "Druk enige toets om voort te gaan.")
Einde
```

```
Prosedure DbInvoeg(D : DefTipe)
Verand
  Fout : String(80)

Begin
  Herhaal
    LeesSkerm(D.S, R)
    SkryfIsam(D.I, R, Fout)
    As Fout <> "" dan
      SkryfBoodskap(D.S, Fout)
    AsEinde
  TotDat Fout = ""
Einde

Einde % DB
```

UNIVERSITY OF JOHANNESBURG

10.4 Gebruik van *DataBasis*.

'n Voorkoms van *DataBasis* word op die gewone manier verkry – 'n veranderlike van hierdie tipe word verklaar. Veronderstel die volgende verklarings kom voor :

```
  Tipe
    SkolierT = Rekord
      Naam : String(30)
      SkolierNr : Heel
      Standerd : Heel
      SportSoort : String(30)
      RekEinde
```

```
Verand
  SkolierDb : DataBasis van SkolierT
    Sleutel(SkolierNr)
    Opsioneel(SportSoort)
```

Dit is nou moontlik om die volgende prosedureroep uit te voer :

```
DbBywerk(SkolierDb)
```

Dit sal veroorsaak dat die gebruiker toegelaat word om die sleutelwaarde (*SkolierNr*) in te tik, waarna die rekord vanaf die databasis geneem sal word, en op die skerm vertoon word. Hierna kan die gebruiker enige veranderings aanbring, waarna die rekord na die databasis teruggeskryf sal word.

Dit is ook moontlik om 'n spyskaartroetine te skryf wat 'n keuse uit die opsies *Bywerk*, *Vertoon*, *Skrap* en *Invoeg* sal aanvaar, waarna dit die toepaslike *DataBasis*-bewerking sal roep. Aangesien dit net nodig sal wees om die verklaring van die rekord te verander (asook die skermuitleg en definisie van Isamlêer) om so 'n program vir 'n nuwe stel inligting te laat werk, beteken dit dat 'n nuwe inligtingstelsel binne minute geskryf sal kan word.

10.5 Moontlike uitbreidings aan die voorbeeld.

'n Hele aantal uitbreidings aan die genoemde voorbeeld is moontlik. Die volgende twee is veral belangrik :

1) Duplikaatwaardes vir sleutels kan toegelaat word. In so geval sou die rekords wat met die genoemde sleutel ooreenkom een na die ander vertoon word, en daar sal aan die gebruiker

gevra word of dit die rekord is waarna hy soek.

2) Sekere toetse kan spesiale betekenisse kry. As die gebruiker die IGNOREER-toets druk, kan veranderings geïgnoreer word. Daar kan ook spesiale toetse wees om na die volgende en vorige velde op die skerm te beweeg.

As die veranderings aangebring word, sal dit natuurlik onmiddellik beskikbaar wees vir alle voorkomste van *DataBasis*.

UNIVERSITY OF JOHANNESBURG

Hoofstuk 11.

11 Implementering van die vertaler.

11.1 Inleiding.

11.2 Interne voorstelling van abstrakte toestelle.

11.3 Algemene bewerkings.

11.4 Bewerkingsparameters.

11.5 Attribute.

11.6 Skikkings.

11.7 Parameterprotokol van bewerkings.

Inhoud.

1 Oorsig.
2 Inleiding.
3 Objekgeoriënteerde programmering.
4 Abstrakte datatipes.
5 Abstrakte datatipes in programmeertale.
6 Die wiskunde van abstrakte datatipes.
7 Data-abstraksie in programmeertale : kort gevallestudies.
8 Abstrakte toestelle in 'n programmeertaal.
9 Rekordgebaseerde toestelle.
10 Hoër-vlak toestelle : 'n voorbeeld.
11 Implementering van die vertaler.
12 Doeltreffende rekenaargebruik.
A Die programmeertaal PT.
B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

11 Implementering van die vertaler.

11.1 Inleiding.

In hierdie afdeling sal gekyk word na moontlike maniere waarop 'n vertaler die nuwe strukture van PT kan vertaal. 'n Vertaler vir PT kan die ander aspekte van PT hanteer soos 'n Pascal-vertaler die ooreenkomkomstige aspekte van Pascal sou hanteer - sien [Aho78] vir 'n beskrywing van tegnieke wat vir vertaler-konstruksie gebruik kan word.

11.2 Interne voorstelling van abstrakte toestelle.

'n Abstrakte toestel kan uit twee dele bestaan :

1) Apparatuur wat aan spesifieke poorte (of geheueadresse) gekoppel is; en
2) Inligting in verband met die toestel wat in die primêre geheue gehou word.

Een of albei hierdie dele kan by 'n abstrakte toestel voorkom - 'n stapel sal slegs uit die geheuegedeelte bestaan, terwyl 'n drukker moontlik slegs aan 'n poort verbind sal wees. 'n Toetsbord kan moontlik uit 'n buffer in die primêre geheue sowel as 'n poortverbinding met die apparatuur bestaan.

Die wyse waarop die vertaler 'n gedeelte van 'n abstrakte toestel wat met apparatuur via 'n poort gekoppel is, hanteer, is eenvoudig - kommunikasie met die apparatuur word bewerkstellig met behulp van leesopdragte vanaf die poort en skryfopdragte na die poort. In gevalle waar 'n poortadres 'n parameter van die abstrakte toestel is, is die vertaling van die toesteldefinisie

nie so voor die hand liggend nie - dit sal egter later gedek word wanneer daar na die voorstelling van geparametriseerde toestelle gekyk word.

Die voorstelling van die geheuegedeelte van 'n toestel sonder parameters en wat ook nie 'n basistipe het nie, lewer nie probleme op nie. Sulke toestelle kan voorgestel word soos 'n rekord deur vertalers vir bestaande programmeertale voorgestel word - die verskillende dele waaruit so 'n geheuegedeelte van 'n abstrakte toestel bestaan, kan by opeenvolgende geheueadresse gehou word. Die datastruktuur wat deur hierdie geheuegedeelte gevorm word, sal voortaan die veranderlikeblok van die toestel genoem word.

Bewerkings van geparametriseerde abstrakte toestelle moet soms verwys na die waardes van parameters van die voorkoms van die toestel waarop dit werk. Die waardes van die parameters moet dus tydens looptyd beskikbaar wees. Een moontlike voorstelling van 'n toestel sal die parameters net soos enige ander veranderlikes van die toestel voorstel. Die interne voorstelling van die toestel sal dus uit twee (afsonderlike of aangrensende) geheuegedeeltes bestaan - die eerste deel sal die waardes van parameters bevat, terwyl die volgende deel (die veranderlikeblok) die waardes van veranderlikes sal bevat. 'n Optimerende vertaler kan natuurlik vasstel watter parameterwaardes wel deur die bewerkings van die toestel benodig word, en dan net daardie waardes by die interne voorstelling van die toestel insluit. Die datastruktuur wat die waardes van die parameters bevat, word die parameterblok van die toestel genoem.

Wanneer 'n toesteldefinisie parameters bevat, kan ook heelwat probleme voorkom omdat die presiese interne samestelling van so 'n abstrakte toestel van die parameters kan afhang; die grootte

van skikkings binne so 'n abstrakte toestel kan byvoorbeeld van 'n parameter afhang met die gevolg dat die adresse van geheuegedeeltes wat op so 'n skikking volg ook van die parameter afhang. As die verskillende dele van 'n geparametriseerde abstrakte toestel by opeenvolgende geheueadresse gehou word, kan hierdie afhanklikheid van parameters veroorsaak dat heelwat berekeninge gedoen moet word om te bepaal by watter adres 'n spesifieke gedeelte van die voorstelling van die toestel voorkom. As hierdie berekeninge tydens looptyd gedoen moet word, kan dit die tyd wat die program benodig om uit te voer, aansienlik verleng.

Een moontlike oplossing om uitvoertyd te verminder, is om telkens wanneer 'n nuwe voorkoms van 'n toestel verklaar word, 'n nuwe stel bewerkings te genereer wat spesifiek by hierdie nuwe voorkoms van die toestel pas. Dit sou dit natuurlik ook onnodig maak om die parameterwaardes van elke voorkoms saam met die veranderlikes van die voorkoms te hou - die bewerkings vir 'n spesifieke voorkoms van 'n toestel, kan die parameters van die voorkoms as konstantes hanteer. Dit is dan selfs moontlik om elke keer die bewerkings vir die spesifieke voorkoms te optimeer. Hierdie oplossing het natuurlik die nadeel dat heelwat geheue vermors word omdat die kode vir die bewerkings tot 'n groot mate gedupliseer word (alhoewel dit moontlik is om hierdie vermorsing te beperk deur 'n goeie optimeerder).

Dit is egter nie nodig om sulke drastiese stappe te neem om uitvoertyd te verbeter nie. 'n Alternatief wat die duplisering van kode onnodig maak, sal soos volg werk : Die vertaler sal 'n meganisme hê wat sal vasstel of enige gegewe veranderlike van 'n toestelparameter afhanklik is. Indien dit wel die geval is, word 'n wyser na die voorstelling van die spesifieke veranderlike gebruik waar die veranderlike normaalweg in die veranderlikeblok

gehou sou word, terwyl die veranderlike self aan die einde van die veranderlikeblok gehou word. Omdat so 'n wyser van 'n konstante grootte is, sal die adresse van die volgende veranderlikes nie deur parameters beïnvloed word nie. Die metode impliseer wel dat 'n bewerking wat 'n veranderlike, wat van 'n parameter afhanklik is, gebruik, twee stappe sal benodig om die veranderlike te bereik : in die eerste stap sal die adres van die wyser na die veranderlike gebruik word om die wyser na die veranderlike te vind, wat dan in die volgende stap gebruik sal word om die werklike veranderlike te bereik. Die addisionele tyd wat hierdeur benodig word is baie klein - veral as die rekenaar 'n indirekte adresseringsmodus voorsien. Verder neem die wysers natuurlik ruimte in beslag, maar hierdie oorhoofse koste is baie klein as dit vergelyk word met die addisionele geheue wat deur duplisering van die kode van bewerkings benodig sou word.

Komplikasies soortgelyk aan die van gewone geparametriseerde abstrakte toestelle, kan voorkom wanneer toestelle met 'n basistipe gedefinieer word - die geheuegrootte wat deur veranderlikes van die basistipe benodig word, sal van die een basistipe tot die volgende basistipe verskil. Die oplossing wat aan die hand gedoen is vir die probleme met geparametriseerde toestelle, los ook hierdie probleme by 'n toestel met 'n basistipe op.

Die voorstelling van attribute sal later bespreek word. Dit is egter moontlik dat die aantal rekordvelde waarmee 'n spesifieke attribuut geassosieer word, van voorkoms tot voorkoms kan verskil. Die geheuegrootte van die voorstelling van 'n attribuut sal dus van die voorkoms afhang, sodat wysers wat na die voorstelling van die attribuut wys, ook in hierdie geval gebruik sal word om die probleem van datastrukture, waarvan die lengte nie in alle gevalle dieselfde is nie, te oorkom.

11.3 Algemene bewerkings.

Met 'n <u>algemene bewerking</u> word 'n bewerking bedoel wat nie deur 'n spesifieke abstrakte toestel voorsien word nie, maar 'n bewerking wat gebruik kan word om enige abstrakte toestel (wat aan gestelde beperkings voldoen) te manipuleer. Die algemene bewerkings het nie toegang tot die interne voorstelling van die toestelle, wat dit kan manipuleer, nie en kan dit gevolglik slegs deur middel van die bewerkings wat deur die onderskeie toestelle voorsien word, gebruik word.

Daar is twee metodes wat gebruik kan word om algemene bewerkings te definieer :

1) So 'n bewerking kan parameters van 'n tipe aanvaar waarvan sommige (tipe-) parameters nie gespesifiseer is nie. Dit is byvoorbeeld die geval waar 'n bewerking 'n string manipuleer, maar waar die lengte van die string nie vooraf vasgestel word nie.
2) 'n Algemene bewerking kan ook parameters aanvaar wat sekere gespesifiseerde bewerkings voorsien. So kan 'n *SkryfString*-bewerking, byvoorbeeld, geskryf word vir enige toestel wat 'n *Skryf*-bewerking voorsien.

Dit is relatief maklik om bewerkings te implementeer wat parameters van 'n geparametriseerde tipe aanvaar waarvan nie alle parameters gespesifiseer is nie - omdat die waardes van parameters tydens uitvoertyd beskikbaar is (in 'n parameterblok) kan enige bewerking die waardes van die parameters bekom wanneer dit benodig word.

Bewerkings wat parameters aanvaar waarop sekere gespesifiseerde bewerkings uitgevoer kan word, het 'n manier nodig om enige parameter met die ooreenkomstige roetine te assosieer. As 'n bewerking gedefinieer word wat, byvoorbeeld, bruikbaar is vir enige tipe wat 'n optellingsbewerking voorsien, sal die bewerking die heeltallige optellingsbewerking moet gebruik wanneer dit met heeltallige parameters geroep word. Soortgelyk sal dit die reële optellingsbewerking moet gebruik wanneer dit met reële parameters geroep word. 'n Metode wat gebruik kan word om die regte bewerking vir enige parameter wat die bewerking voorsien, te assosieer, is om saam met die parameter nog 'n skikking van adresse te stuur. Die adresskikking bevat dan die beginadresse van al die roetines wat benodig word. As 'n prosedure geskryf word wat, sê, optellings- en aftrekkingsbewerkings benodig en dan parameters aanvaar wat hierdie bewerkings voorsien, sal, wanneer die prosedure gebruik word, die adresse van toepaslike optellings- en aftrekkingsroetines saam met die werklike parameter aan die prosedure gestuur word. Aangesien oorlaaide bewerkings dieselfde geroep word, ongeag die datatipe waarvoor dit gedefinieer is, is die beginadres van 'n bewerking al wat deur 'n ander roetine benodig word om die bewerking te gebruik.

Die skikking wat die beginadresse van die gespesifiseerde bewerkings bevat, word 'n bewerkingsblok genoem.

11.4 Bewerkingsparameters.

Oor die algemeen word van twee maniere gebruik gemaak om parameters aan prosedures en funksies te stuur : die adres van die werklike parameter word gestuur ('n verwysingsparameter, Eng. call-by-reference parameter) of die waarde van die werklike parameter word na 'n spesiale area gekopieer, vanwaar die

prosedure of funksie dit kan gebruik ('n waardeparameter, Eng. call-by-value parameter). Aangesien 'n groot verskeidenheid parameters aan bewerkings van toestelle en 'n (nog groter) verskeidenheid aan algemene bewerkings gestuur kan word, word slegs adresse van interne voorstellings van toestelle in PT as parameters gestuur.

PT voorsien drie soorte parameters : "In"-, "Uit"- en "In Uit"-parameters.

In die geval van "In Uit"-parameters word die voorstelling van die werklike parameter (argument) gebruik wanneer die formele parameter binne die prosedure gebruik word. Waar veranderings binne die prosedure aan die waarde van die formele parameter aangebring word, word dit dus outomaties aan die werklike parameter aangebring. Dit kan gedoen word omdat die adres van die werklike parameter aan die prosedure gestuur word, en nie die waarde nie.

UNIVERSITY
OF
JOHANNESBURG

"In"-parameters is die waardeparameters van PT, dit wil sê die parameters waarvan die waardes binne die prosedure gebruik kan word, maar nie gewysig kan word nie. In PT word die adres van die werklike parameter (argument) weer eens aan die prosedure gestuur. Alhoewel die parameter net soos "In Uit"-parameters hanteer word, is dit nou die werk van die vertaler om te kontroleer dat die waarde van die parameter nie in die prosedure verander word nie. Dit impliseer ook dat so 'n parameter nie as 'n "In Uit"-parameter aan 'n ander prosedure gestuur mag word nie.

"Uit"-parameters is soortgelyk aan "In Uit"-parameters en word soortgelyk hanteer. Die vertaler kontroleer egter dat "Uit"-parameters se waarde nie binne die prosedure waaraan dit behoort,

gebruik word nie. "Uit"-parameters word gebruik waar parameters benodig word om resultate aan die roepende prosedure te lewer. Die voordeel van "Uit"-parameters is dat die roepende prosedure nie werklike parameters hoef te verskaf as dit nie die betrokke resultate benodig nie. Omdat die waarde van die parameter nêrens gebruik word nie, en dit maklik is om te kontroleer of 'n werklike parameter voorsien is voordat 'n waarde aan die formele parameter toegewys word, is dit duidelik dat dit nie nodig is om 'n werklike parameter te voorsien nie.

11.5 Attribute.

Attribute word primêr deur bewerkings gebruik om spesifieke velde van veranderlikes (rekords) van die basistipe, te bereik. 'n Bewerking sal dus toegang tot 'n rekord en tot die interne voorstelling van die attribuut hê en daarmee moet die n-de veld met die spesifieke attribuut, vir 'n geldige n, bereik kan word.

Die gekose voorstelling van attribute bestaan uit 'n skikking met een element vir elke veld van die rekord waarmee die attribuut geassosieer word. Die eerste veld van die n-de element van die skikking bevat die afset binne die rekord van die n-de rekordveld met die attribuut. Om die adres van die n-de rekordveld te bepaal, is dit net nodig om die inhoud van die eerste veld van die n-de element van die attribuutskikking by die beginadres van die betrokke rekord te tel. Die tweede veld van elke element van die skikking bevat 'n wyser na die parameterblok van die rekordveld waarna die eerste veld wys. Die parameterblok word natuurlik benodig deur bewerkings wat gebruik gaan word om die veld te manipuleer. Die derde veld van elke item bevat die nommer van die rekordveld waarmee die attribuut geassosieer word, sodat die *VeldNr*-funksie ondersteun kan word. Aangesien dit

nodig is om te kontroleer dat daar nie na meer velde wat die attribuut bevat as wat daar in werklikheid is, verwys word nie, word element nul van die attribuutskikking gebruik om die aantal velde waarna hierdie attribuutskikking wys, te hou.

As daar beperkings op die tipes van die velde waarmee 'n gegewe attribuut geassosieer mag word, gelê is, voorsien die beskryfde voorstelling in die meeste gevalle voldoende inligting dat die velde deur geskikte bewerkings gebruik kan word - die waardes van parameters, sowel as die veranderlikes van die toestel, is beskikbaar en kan aan bewerkings wat dit benodig, verskaf word. Soos in die geval van algemene bewerkings (11.3) is dit egter ook nodig om addisionele inligting te voorsien as die beperking vereis dat die attribuut met velde geassosieer mag word wat sekere gespesifiseerde bewerkings voorsien. In sulke gevalle sal die elemente van die attribuutskikking uit vier velde bestaan. Die eerste drie velde van elke element sal weer eens die afset van die veld binne die rekord, 'n wyser na die parameterblok van die veld en die veldnommer bevat. Die volgende veld sal 'n wyser na 'n bewerkingsblok wees. Die bewerkingsblok sal, soos voorheen, die beginadresse van al die gevraagde bewerkings van die tipe van die betrokke rekordveld bevat. 'n Benodigde bewerking kan dan geaktiveer word deur 'n sprong na die betrokke adres uit te voer.

Die attribuutskikking sal voortaan 'n <u>attribuutblok</u> van die toestel genoem word. Die attribuutblokke van die toestel vorm deel van die parameterblok van die toestel en word aan die einde van die parameterblok gehou. Wysers na elk van die attribuutblokke van die toestel word voor die attribuutblokke in die parameterblok gehou, sodat die attribuutblokke vinnig bereik kan word.

UNIVERSITY OF JOHANNESBURG

11.6 Skikkings.

Aangesien die parameterblok van al die elemente van 'n skikking dieselfde sal lyk, sal dit duidelik ondoeltreffend wees om die parameterblok vir elke element van die skikking te dupliseer. Om hierdie rede kan die voorstelling van 'n skikking uit die volgende dele bestaan :

Parameterblok van die skikking :

1) 'n Wyser na die parameterblok van die elemente van die skikking;
2) Die waarde van die onderste grens van die skikking; en
3) Die waarde van die boonste grens van die skikking.

Veranderlikeblok van die skikking :

1) Die veranderlikeblokke van die elemente van die skikking, in volgorde.

PT hanteer n-dimensionele skikkings as 'n een-dimensionele skikking van (n-1)-dimensionele skikkings. Dit is hoe dit ook in Pascal (onder andere) hanteer word. Hierdie metode veroorsaak dat die parameterblok van 'n skikking altyd net vir twee grense (onder en bo) voorsiening hoef te maak.

Skikkings is nie die enigste geval waar dit moontlik is om ruimte te bespaar deur meer as een veranderlike dieselfde parameterblok te laat gebruik nie - 'n optimerende vertaler kan telkens wanneer 'n nuwe veranderlike verklaar word, uitvind of daar reeds 'n geskikte parameterblok vir die veranderlike bestaan en, indien wel, die bestaande blok gebruik. In die geval van skikking is die potensiële vermorsing van geheue, as vir elke element van die skikking 'n afsonderlike parameterblok geskep word, so groot dat

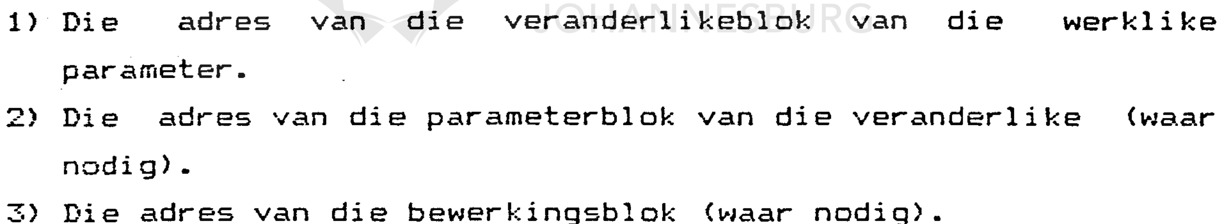
UNIVERSITY OF JOHANNESBURG

enige vertaler in hierdie geval net die een parameterblok behoort te skep.

11.7 Parameterprotokol van bewerkings.

Al die inligting in verband met enige veranderlike, wat deur die bewerkings van so 'n veranderlike benodig word, is bevat in die veranderlike- en parameterblokke van die veranderlike - die parameterblok is natuurlik net nodig waar die tipe van die veranderlike wel parameters toelaat. Bewerkings wat enige parameter aanvaar, solank dit net sekere bewerkings voorsien, benodig ook die bewerkingsblok van die veranderlike wat as die parameter gestuur word.

Parameters kan gevolglik soos volg aan 'n bewerking oorgedra word :



1) Die adres van die veranderlikeblok van die werklike parameter.
2) Die adres van die parameterblok van die veranderlike (waar nodig).
3) Die adres van die bewerkingsblok (waar nodig).

In gevalle waar 'n konstante as 'n parameter aan 'n bewerking gestuur word, sal die wyser wat normaalweg na die veranderlike-blok van die parameter wys, na die voorstelling van die konstante wys. Waar die tipe van die konstante parameters benodig (soos die geval by karakterstringe is), sal 'n parameterblok vir die konstante geskep word. Die wyser wat normaalweg na die parameterblok van die parameter wys, sal nou na hierdie parameterblok wys.

In die geval van tipes wat nie parameters voorsien nie, is dit moontlik om die wyser na die parameterblok weg te laat. As die wyser egter wel ingesluit word ter wille van 'n eenvormige protokol, kan die wyser enige waarde bevat, aangesien dit nie deur die bewerkings van die tipe gebruik sal word nie.

Die adres van die bewerkingsblok word natuurlik net benodig in die geval van *TipeMet*-parameters. Dit is egter weer eens moontlik om hierdie adres vir alle parameters te verskaf, in welke geval dit ook enige waarde mag hê, aangesien dit nie gebruik sal word nie.

Parameterblokke kan as konstantes in die vertaalde kopie van die program gehou word, en wysers na hierdie vaste areas kan aan die bewerkings wat dit benodig, gestuur word. In die meeste gevalle is dit ook moontlik om bewerkingsblokke as sulke konstantes te hanteer. Daar is egter gevalle waar die inhoud van 'n bewerkingsblok nie tydens vertaaltyd bekend is nie, en die blok tydens looptyd gekonstrueer moet word. Dit is maklik om aan te toon dat die inligting wat tydens looptyd benodig word om hierdie blokke te konstrueer, wel beskikbaar sal wees.

As dit nodig is om 'n bewerkingsblok tydens looptyd te konstrueer, beteken dit dat die adresse van die bewerkings van die tipe, waarvoor die bewerkingsblok gekonstrueer moet word, onbekend was tydens vertaaltyd. Dit kan net gebeur as die tipe nie tydens vertaaltyd bekend is nie, wat net kan gebeur as die betrokke tipe enige tipe is wat gespesifiseerde bewerkings voorsien en dus moet daar 'n bewerkingsblok vir die betrokke tipe tydens looptyd beskikbaar wees. Verder moet al die bewerkings wat in die nuwe bewerkingsblok benodig word in die bewerkingsblok van die werklike parameter voorkom. Dit is dus net nodig om die adresse van die bewerkings wat in die nuwe bewerkingsblok hoort,

vanuit die bewerkingsblok van die werklike parameter te kopieer om die verlangde bewerkingsblok te konstrueer.

UNIVERSITY
OF
JOHANNESBURG

Hoofstuk 12.

12 Doeltreffende rekenaargebruik.
- 12.1 Inleiding.
- 12.2 Programmeerderproduktiwiteit.
- 12.3 Hulpbronproduktiwiteit.

Inhoud.

1 Oorsig.
2 Inleiding.
3 Objekgeoriënteerde programmering.
4 Abstrakte datatipes.
5 Abstrakte datatipes in programmeertale.
6 Die wiskunde van abstrakte datatipes.
7 Data-abstraksie in programmeertale : kort gevallestudies.
8 Abstrakte toestelle in 'n programmeertaal.
9 Rekordgebaseerde toestelle.
10 Hoër-vlak toestelle : 'n voorbeeld.
11 Implementering van die vertaler.
12 Doeltreffende rekenaargebruik.
A Die programmeertaal PT.
B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

12 Doeltreffende rekenaargebruik.

12.1 Inleiding.

Uit die vorige hoofstuk is dit duidelik dat die abstraksie-meganismes wat deur 'n taal soos PT voorsien word, heelwat oorhoofse koste vereis. Alhoewel dit algemeen aanvaar word dat programmeerderdoeltreffendheid belangriker is as om rekenaar-hulpbronne so doeltreffend moontlik te gebruik, is dit tog belangrik om te kontroleer dat die oorhoofse koste wat aan nuwe programmeringshulpmiddels verbonde is, nie so hoog is dat die hulpmiddel nie werklik prakties is nie.

In hierdie hoofstuk sal aangetoon word dat die gebruik van abstrakte toestelle in 'n program nie sal lei tot minder produktiewe gebruik van rekenaarhulpbronne as wat die geval sou wees as die program van meer konvensionele tegnieke gebruik sou maak nie, terwyl die gebruik van abstrakte toestelle programmeerderproduktiwiteit kan verhoog.

12.2 Programmeerderproduktiwiteit.

Abstrakte toestelle, soos dit in PT ondersteun word, verhoog programmeerderproduktiwiteit op twee maniere.

Eerstens maak die vermoë om die dinge wat saam hoort saam te groepeer en dit dan van die res van die programmatuur af te skerm, dit makliker om in die eerste plek korrekte programmatuur te ontwikkel en in die tweede plek is dit ook makliker om programmatuur wat so georganiseer is, te wysig wanneer omstandighede verander.

Tweedens veroorsaak die fasiliteit om 'n klas van toestelle te definieer dat dit moontlik is om een stuk kode te ontwikkel en dieselfde kode dan vir 'n aantal toestelle te gebruik, wat koderingstyd bespaar.

Die gebruik van 'n programmeertaal soos PT waarborg egter nie hierdie voordele nie - dit voorsien slegs die fasiliteite wat deur die programmeerder gebruik kan word as hy hierdie voordele wil geniet.

12.3 Hulpbronproduktiwiteit.

Oorhoofse koste verbonde aan abstrakte toestelle kan in drie kategorieë ingedeel word :

1) Addisionele geheue wat benodig word om inligting aangaande abstrakte toestelle te hou, byvoorbeeld geheue wat benodig word om die grense van skikkings te hou;
2) Rekenaartyd wat benodig word om inligting aangaande toestelle te onttrek en te gebruik;
3) Tyd wat benodig word om velde van rekords via attribute te gebruik, asook die rekenaargeheue wat vir hierdie attribute benodig word.

Die geheue wat benodig word vir inligting aangaande abstrakte toestelle word dikwels ook in konvensionele programmeertale gebruik om inligting aangaande datastrukture te hou. So word die onder- en bogrense van skikkings dikwels in ander hoë-vlak tale as deel van die interne voorstelling van die skikking gehou - soms vir foutkontroledoeleindes en soms omdat dit deur looptyd-funksies, wat hierdie grense van enige skikking lewer, benodig word. In die geval van abstrakte toestelle is daar dikwels meer

inligting wat in verband met die toestel gehou moet word - oor die algemeen sal dit geld dat hoe meer abstrak 'n toestel is, hoe meer parameters sal dit benodig en, gevolglik, hoe meer geheueruimte sal benodig word om die parameters te hou. As 'n meer abstrakte toestel egter gebruik word, sal een operasie voldoende wees om enige voorkoms van die toestel 'n sekere funksie te laat uitvoer - die verskillende voorkomste, sou in 'n taal wat nie abstrakte toestelle voorsien nie, as verskillende datastrukture geïmplementeer moes word, in welke geval afsonderlike prosedures dan vir elk van hierdie datastrukture geskryf sou moes word om die betrokke funksie te ondersteun. Die addisionele ruimte wat in so 'n geval deur parameters beslaan word, kan dus moontlik weer bespaar word omdat minder kode benodig word om die voorkomste van die abstrakte toestel te manipuleer.

Die addisionele tyd wat benodig word om inligting aangaande toestelle te onttrek, is minimaal, veral waar die rekenaarapparatuur geskikte adresseringsmodusse voorsien. Die tyd wat benodig word om die inligting te gebruik, kan egter in sommige gevalle 'n probleem veroorsaak : omdat die operasies van 'n abstrakte toestel enige voorkoms van die toestel moet kan hanteer, mag dit vir so 'n operasie nodig wees om op grond van die waarde van 'n parameter te besluit watter een van 'n aantal alternatiewe gedeeltes kode gebruik moet word, of die waarde van so 'n parameter kan binne 'n ingewikkelde uitdrukking gebruik word. Hierdie tyd is op sigself nie groot nie, maar waar dit baie gebruik word (byvoorbeeld in lusse) mag die gesamentlike tyd groter word as wat wenslik is. Waar tyd kritiek is, kan sorg deur die programmeerder wanneer hy sulke operasies kodeer en/of 'n goeie optimeerder, die probleem oplos. Waar dit nie voldoende is nie, sal tot 'n mindere mate van abstraksie gebruik gemaak moet word.

Ook die tyd en ruimte wat vir attribute benodig word, is nie te groot nie, omdat dit tot gevolg het dat minder ruimte vir kode benodig word - as attribute nie gebruik word nie sou elk van die betrokke velde afsonderlik hanteer moet word - en die addisionele tyd wat benodig word - as gevolg van die wysers wat gevolg moet word om die betrokke veld te bereik - is minimaal as geskikte adresseringsmodusse op die betrokke rekenaar beskikbaar is. Die krag wat attribute voorsien, regverdig die oorhoofse koste wat daaraan verbonde is.

Bylaag A.

A Die programmeertaal PT.
A.1 Inleiding.
A.2 Die ontwerp van PT.
A.3 Notasie van die sintaksspesifikasie.
A.4 Die PT-spesifikasie.
A.4.1 Atomiese eenhede van die taal.
A.4.2 Tipografie.
A.4.3 PT-sintaks.
A.5 Strekking van identifiseerders.
A.6 Voorafgedefinieerde bewerkings.

Inhoud.

1 Oorsig.
2 Inleiding.
3 Objekgeoriënteerde programmering.
4 Abstrakte datatipes.
5 Abstrakte datatipes in programmeertale.
6 Die wiskunde van abstrakte datatipes.
7 Data-abstraksie in programmeertale : kort gevallestudies.
8 Abstrakte toestelle in 'n programmeertaal.
9 Rekordgebaseerde toestelle.
10 Hoër-vlak toestelle : 'n voorbeeld.
11 Implementering van die vertaler.
12 Doeltreffende rekenaargebruik.
A Die programmeertaal PT.
B Verwysings en bibliografie.

-oOo-

UNIVERSITY OF JOHANNESBURG

Bylaag A - Die programmeertaal PT.

A.1 Inleiding

PT is ontwerp met die doel om die wyse waarop 'n programmeertaal abstrakte toestelle kan ondersteun, te illustreer. Hierdie bylaag bevat 'n beskrywing van die huidige weergawe van PT.

A.2 Die ontwerp van PT.

Aangesien PT ontwerp is om abstrakte toestelle te ondersteun, is dit die enigste gebied waar dit noemenswaardig van bestaande programmeertale verskil. Gedeeltes van die programmeertaal wat nie 'n invloed op abstrakte toestelle het nie, lyk soos die ooreenkomstige gedeeltes van ander tale in die Algol-familie.

UNIVERSITY OF JOHANNESBURG

A.3 Notasie van die sintaksspesifikasie.

'n Uitgebreide en aangepaste vorm van Backus-Naur-vorm (BNF) word gebruik om die sintaks van PT te definieer. Die notasie is aangepas sodat dit makliker deur 'n rekenaar verwerk kan word - iets wat gebruik is tydens die konstruksie van 'n PT-vertaler.

Die veranderlike simbole van die notasie bestaan uit 'n string alfanumeriese karakters. Terminaalsimbole bestaan uit 'n string karakters wat deur aanhalingstekens begrens word. Die simbole kan langs mekaar geskryf word om aan te dui in watter volgorde terminaalsimbole in 'n program mag voorkom. 'n Aantal simbole mag saamgegroepeer word deur dit in krulhakies te plaas; saamgroepering is nuttig vir die bewerkings wat nou beskryf sal word. 'n Vraagteken wat op 'n simbool of groep simbole volg, dui op 'n

opsionele konstrukt, dit wil sê, 'n konstrukt wat nul of een keer gebruik mag word. 'n Ster (asterisk, *) wat op 'n simbool of groep simbole volg, dui aan dat die konstrukt nul of meer keer herhaal mag word. Soortgelyk word 'n plus gebruik om aan te dui dat konstrukte een of meer kere herhaal mag word. Alternatiewe word deur 'n vertikale streep (|) geskei. 'n Veranderlike simbool word in terme van 'n uitdrukking (wat uit veranderlike simbole, terminaalsimbole asook enige van bogenoemde bewerkingstekens mag bestaan) gedefinieer met behulp van die gelyk-aanteken - die veranderlike wat gedefinieer word, word links van die gelyk-aan-teken geplaas, terwyl die definiërende uitdrukking regs daarvan geplaas word. Die definisie van 'n veranderlike word met 'n kommapunt afgesluit.

A.4 Die PT-spesifikasie.

Hier volg 'n volledige sintaksspesifikasie van PT. Die semantiek word ook bespreek, alhoewel nie volledig nie - die gedeeltes wat elders in die verhandeling volledig bespreek is, word nie altyd hier herhaal nie. Waar Pascal 'n soortgelyke fasiliteit as PT voorsien, en die PT-semantiek word nie gespesifiseer nie, kan aanvaar word dat dit soos die Pascal-semantiek is - dit word gedoen om ruimte te bespaar.

A.4.1 Atomiese eenhede van die taal.

Die atomiese eenhede waaruit programme in die taal gebou kan word is die terminaalsimbole (die gereserveerde woorde en spesiale simbole), identifiseerders en konstantes.

Die reëls wat gebruik kan word om konstantes (*Konst*, in die spesifikasie) en identifiseerders (*Id*, in die spesifikasie) te

konstrueer, is die volgende :

```
Id = Letter { Letter | Syfer } *;

Konst = Syfer + |
        Syfer + "." Syfer * { "E" Syfer + } ? |
        "’" Karakter |
        Aanhalingsteken Karakter * Aanhalingsteken ;

Syfer  =  "0" | "1" | "2" | "3" | "4" |
          "5" | "6" | "7" | "8" | "9" ;

Letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
         "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
         "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "_" |
         "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
         "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
         "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;

Aanhalingsteken = """" ;
```

Aanhalingsteken is die dubbelaanhalingsteken ("). *Karakter* is enige van die karakters wat in die karakterstel van die rekenaar voorkom, met die uitsondering van kodes wat gebruik word om die einde van ’n lyn (CR en LF in ASCII) en die einde van ’n lêer aan te dui.

Daar word nie tussen klein- en hoof*letters* onderskei nie, sodat "abc" en "ABC" dieselfde identifiseerder is. Dieselfde geld vir gereserveerde woorde - dit kan met klein- of hoofletters (of ’n kombinasie van die twee) geskryf word.

Daar word nie beperkings op die lengte van identifiseerders of

konstantes gelê nie, alhoewel net die eerste 255 karakters van so 'n eenheid deur die vertaler gebruik word.

'n Tipe word met elke konstante geassosieer - konstantes wat soos die onderskeie alternatiewe in die *Konst*-definisie lyk, is (in die volgorde waarin die alternatiewe in die *Konst*-definisie voorkom) van tipe *Heel*, *Reeel*, *Kar* of *String(n)*, waar n die aantal karakters in die string is.

Spasies word nie in 'n atomiese eenheid (identifiseerder of konstante) toegelaat nie, behalwe waar dit 'n spesiale betekenis het (dit wil sê, in karakterstring- en karakterkonstantes).

A.4.2 Tipografie.

PT-programme het 'n vry formaat. Enige aantal skeidingsimbole (spasies en lyneindekarakters) mag tussen die atomiese eehede van 'n program voorkom. Waar verwarring mag ontstaan wanneer daar nie 'n skeidingsimbool tussen twee atomiese eenhede sou voorkom nie, is die gebruik van 'n skeidingsimbool verpligtend - dit is byvoorbeeld die geval waar die getal 10 op die gereserveerde woord *Tot* volg, omdat *Tot10* 'n geldige identifiseerder is.

Kommentaar mag enige plek in 'n program voorkom en word net soos enige ander skeidingsimbool hanteer. Kommentaar word ingelei met die persentasie- (%) simbool. Kommentaar word afgesluit deur die volgende persentasiesimbool of lyneindekarakter - watter een ook al eerste kom.

UNIVERSITY OF JOHANNESBURG

A.4.3 PT-sintaks.

- Name.

Naam = Id { "." Id }*;

Die punt (.) word gebruik om identifiseerders te kwalifiseer - as die identifiseerder "i" in module "x" verklaar is, en "i" ook in module "y", dan sal "x.i" na die "i" van "x" verwys, terwyl "y.i" na die "i" van "y" sal verwys. Wanneer enige identifiseerder, wat binne 'n module met 'n naam verklaar is, buite die module gebruik word, moet die identifiseerder deur die modulenaam gekwalifiseer word. Wanneer 'n identifiseerder binne 'n module (met 'n naam) verklaar is, en die module weer binne 'n nader module (met 'n naam) voorkom, sal dit nodig wees om die identifiseerder deur albei die modulename te kwalifiseer, wanneer dit buite hierdie modules gebruik word - die naam van die buitenste module word eerstens gegee, gevolg deur 'n punt, gevolg deur die naam van die ander module, wat gevolg word deur die identifiseerder. Waar meer as twee modules genes is, kan hierdie kwalifisering so veel as nodig herhaal word. Identifiseerders mag natuurlik net buite 'n module gebruik word, as dit eksplisiet so verklaar word - sien die beskrywing van modules, en spesifiek die beskrywing van die *Def*-lys.

Velde van 'n rekord, word ook gekwalifiseer deur die naam van die rekord - dit word presies gedoen soos dit in Pascal gedoen word.

- Tipes.

```
KonstNaam = Naam;

RekordTipeNaam = Naam;

VeldTipeNaam = RekordTipeNaam "." Id;

Konstante = Konst | KonstNaam |
            "#" RekordTipeNaam | "#" VeldTipeNaam;
```

'n *Konstante* kan een van die konstantes wees wat as 'n atomiese eenheid herken word; dit kan ook 'n (moontlik gekwalifiseerde) identifiseerder wees, wat met so 'n konstante geassosieer is (sien *Verklarings* later), in welke geval dit die waarde en tipe van daardie konstante sal hê; 'n *Konstante* kan ook 'n (moontlik gekwalifiseerde) identifiseerder wees wat in die gebied van 'n geënumereerde tipe (dit wil sê, 'n tipe waar die waardes in die gebied van die tipe, 'n lys van identifiseerders is) verklaar is. Waar die nommerteken (#) voor die naam van 'n rekordtipe voorkom, word 'n heeltallige konstante aanvaar - die waarde van hierdie konstante is die aantal velde wat in die rekord voorkom ('n veld wat self 'n rekord is, tel net as een veld). Die nommerteken saam met die naam van die veld, is ook 'n heeltallige konstante wat die posisie van die veld binne die onmiddellik omvattende rekord aandui.

```
AttrId = Id;

VeldId = Id;

VeldLys = "(" VeldId { "," VeldId } * ")" |
          "SOOS" "BASISTIPE";
```

AttrLys = { AttrId VeldLys }*;

'n *AttrLys* volg altyd op die naam van 'n rekordtipe, wat die basistipe van 'n gebruikersgedefinieerde tipe (sien later) is. Die name van die attribute (*AttrId*) moet attribute wees wat deur die gebruikersgedefinieerde tipe gedefinieer is. Die name van die velde (*VeldId*) is name van velde wat binne die genoemde rekordtipe voorkom – hierdie name word nie gekwalifiseer nie. Wanneer so 'n *AttrLys* binne 'n gebruikersgedefinieerde tipe voorkom, waarvan die basistipe 'n rekord is en die *AttrLys* na 'n rekord van hierdie basistipe voorkom en, verder, die tipe wat gebruik word, 'n identiese attribuut toelaat as die tipe waarbinne hierdie *AttrLys* voorkom, dan kan die "SOOS BASISTIPE"-gedeelte gebruik word om aan te dui dat die velde van die basistipe wat met hierdie attribuut geassosieer is, ook met die gelyknamige attribuut van die tipe wat nou gebruik word, geassosieer moet word.

UNIVERSITY OF JOHANNESBURG

VeldBeskr = Id ":" TipeSpes;

Bewerking = Operator | Id;

Die *Id*, wat 'n *Bewerking* mag wees, moet 'n roetine met 'n DEFTIPE-parameter (sien later) wees.

TipeNaam = Naam;

'n *TipeNaam* is 'n (moontlik gekwalifiseerde) identifiseerder wat met 'n tipe geassosieer is (sien *Verklarings* later).

```
VasteTipe = "(" Id { "," Id }* ")" |
            "HEEL" |
            "REEEL" |
            "KAR" |
            "LOGIES" |
            "REKORD" VeldBeskr+ "EINDE";

TipeSpes = VasteTipe |
           "STRING" "(" Konstante ")" |
           "SKIK" "[" Konstante ":" Konstante "]"
                  "VAN" TipeSpes |
           TipeNaam { "(" Konstante { "," Konstante } * ")" } ?
                    { "VAN" TipeSpes AttrLys } ?;

AbstrTipe = VasteTipe |
            "STRING" "(" Konstante ? ")" |
            "DEFTIPE" |
            "BASISTIPE" |
            "TIPEMET" "(" Bewerking { "," Bewerking }* ")" |
            "SKIK" "[" { Konstante ":" Konstante } ? "]"
                   { "VAN" AbstrTipe } |
            TipeNaam { "(" Konstante { "," Konstante } * ")" } ?
                   { "VAN" TipeSpes AttrLys } ?;
```

Veranderlikes mag van die tipes wat in *TipeSpes* genoem word, verklaar word. Die tipes wat in *AbstrTipe* genoem word, word gebruik as die tipes van parameters wat aan 'n roetine gestuur kan word - hierdie tipes is nie altyd volledig gespesifiseer nie omdat parameters wat deel van die tipe vorm (byvoorbeeld die lengte van 'n string) weggelaat mag word.

Die *Konstante*, wat die lengte van 'n STRING aandui, moet (in *TipeSpes* en *AbstrTipe*) van tipe HEEL wees. Die konstantes wat

UNIVERSITY OF JOHANNESBURG

die grense van 'n SKIKking spesifiseer, moet albei van tipe HEEL, albei van tipe KAR of albei van dieselfde geënumereerde tipe wees.

Die tipes DEFTIPE en BASISTIPE kan net binne die definisie van 'n gebruikersgedefinieerde tipe gebruik word. DEFTIPE laat 'n bewerking toe om die interne voorstelling van die tipe wat gebruik word, te gebruik. BASISTIPE mag net gebruik word as die tipe wat gedefinieer word, 'n basistipe (sien later) toelaat - in daardie geval is BASISTIPE dieselfde tipe as die basistipe van die tipe wat gedefinieer word.

- Parameters en argumente.

```
FormParms = "(" { ParmVkl { "," Parmvkl }* } ? ")";
```

```
ParmVkl = Id ":" AbstrTipe "IN" ? "UIT" ?;
```

UNIVERSITY OF JOHANNESBURG

Elke parameter van 'n bewerking word met 'n tipe geassosieer. Verder word die gebruikswyse van die parameter aangedui as IN, UIT of IN UIT. As die gebruikswyse nie eksplisiet gespesifiseer word nie, word IN UIT aanvaar. IN sê dat die waarde van die parameter net gebruik mag word, maar nie gewysig nie; UIT dat die waarde net gewysig mag word - en nie gebruik nie; terwyl IN UIT beide gebruik en wysiging van die parameter toelaat.

```
Arg = Verand | Uitdr | "FOP";
```

```
Args = "(" { Arg { "," Arg }* }? ")";
```

Die argumente (of werklike parameters) wat aan 'n bewerking gestuur word, moet elk dieselfde tipe as die ooreenkomstige formele parameter hê. As die formele parameter IN is, moet 'n

uitdrukking as die werklike parameter gestuur word; as die formele parameter IN UIT is, moet 'n veranderlike as die werklike parameter gestuur word; as die formele parameter UIT is, kan 'n veranderlike of FOP as die werklike parameter gestuur word - as FOP gestuur word, word veranderings wat deur die bewerking aan die formele parameter aangebring word, geïgnoreer.

- Uitdrukkings.

```
FunksieNaam = Naam;

Faktor = Konst |
         KonstNaam |
         Verand |
         FunksieNaam Args |
         "(" Uitdr ")";

Geteken =  "_" Geteken | Faktor;

MagsUitdr = Geteken { "**" MagsUitdr } ?;

VermOperator = "*" | "/" | "MOD" | "DEEL" | "\";

Term = MagsUitdr { VermOperator MagsUitdr }*;

SomOperator = "+" | "-";

Som = Term { SomOperator Term } *;

Relasie = ">" | "<" | "<>" | "<=" | ">=" | "=";

Vergelyking = Som { Relasie Som } ?;
```

UNIVERSITY OF JOHANNESBURG

Inverse = "NIE" Inverse | Vergelyking;

EnOperator = "&" | "EN";

EnUitdr = Inverse { EnOperator Inverse }*;

OfOperator = "OF" | "|";

OfUitdr = EnUitdr { OfOperator EnUitdr }*;

Uitdr = OfUitdr { "->" Filter }*;

Die bewerkings wat in hierdie spesifikasie genoem is, mag net gebruik word as dit gedefinieer is vir die tipe van die operande waarmee dit gebruik word. Die bewerkings is vooraf vir heeltallige en reële operande gedefinieer, en het dieselfde effek as wat die ooreenkomstige bewerkings in Pascal. Die onderkoppel (_) word in PT gebruik om die unêre minus aan te dui. Die unêre plus word nie ondersteun nie.

UNIVERSITY OF JOHANNESBURG

As 'n uitdrukking sy aanvanklike resultaat deur een of meer filters stuur, is die waarde van die uiteindelike resultaat die waarde wat deur die laaste (dit wil sê, die heel regterkantste) filter gelewer word, en die tipe van die uitdrukking is die tipe van die resultaat wat deur die laaste filter gelewer word.

Voorwaarde = Uitdr;

'n Voorwaarde is 'n uitdrukking wat 'n resultaat van tipe LOGIES lewer.

Operator = VermOperator | SomOperator | Relasie | "NIE" |
EnOperator | OfOperator | ":=" | "_";

```
Verand = Naam { { "[" Uitdr "]" }+ { "." Naam }? }*;
```

Die indekse wat langs (of in) die naam van 'n veranderlike tussen vierkantige hakies toegelaat word, identifiseer 'n spesifieke element van 'n skikking.

- Sinne.

```
ToewysingSin = Verand ":=" Uitdr;
```

Die presiese gedrag van die toewysingsin hang van die definisie van die toewysingoperator (:=) vir die betrokke tipe af. Die operator behoort egter die effek te hê dat dit 'n kopie van die waarde van die uitdrukking maak en dan veroorsaak dat hierdie kopie gebruik word wanneer die veranderlike aan die linkerkant van die toewysingoperator weer gebruik word.

```
ProsedureNaam = Naam;
```

```
SubroetineRoep = ProsedureNaam Args;
```

Die *SubroetineRoep* veroorsaak dat die genoemde prosedure geroep word

```
AsSin = "AS" Voorwaarde "DAN" Sinne
        { "ASANDERS" Voorwaarde "DAN" Sinne }*
        { "ANDERS" Sinne } ?
        "ASEINDE";
```

In die geval van die *AsSin* sal die program deur die AS- en ASANDERS-sinne vloei (en die sinne wat op die gepaardgaande DAN volg, oorslaan) totdat dit 'n voorwaarde bereik wat die resultaat

WAAR lewer. Die sinne na hierdie DAN sal dan uitgevoer word, waarna beheer na die sin wat op die ASEINDE volg, verplaas sal word. As nie een van die voorwaardes die waarde WAAR lewer nie, sal die sinne wat op die ANDERS (as dit voorkom) volg, uitgevoer word.

```
BeskouSin = "BESKOU" Uitdr
            { "WANNEER" Uitdr { "," Uitdr }*
              "DAN" Sinne }*
            { "ANDER" Sinne } ?
            "BESKOUEINDE";
```

In die geval van die *BeskouSin*, sal die waarde van die uitdrukking bereken word, waarna deur die WANNEER-lyste gesoek sal word (in die volgorde waarin dit in die program voorkom) om die waarde te vind. As die waarde gevind word, sal die sinne wat op die ooreenkomstige DAN volg, uitgevoer word. As die waarde nie voorkom nie, sal die sinne wat op die ANDER (as dit voorkom) volg, uitgevoer word, of nie een van die sinne van die BESKOU-sin sal uitgevoer word nie (as die ANDER weggelaat is). Hierna sal beheer verplaas word na die sin wat op die BESKOUEINDE volg. Die uitdrukkings wat in die WANNEER-lyste voorkom, moet dieselfde tipe as die uitdrukking wat langs die BESKOU voorkom, hê. Hierdie tipe moet ook die gelykheidsrelasie (die "="-operator) voorsien.

```
Herhaal = "HERHAAL" Sinne { "LUSEINDE" | "TOTDAT" Voorwaarde };

Lus = "SOLANK" Voorwaarde Herhaal |
      Herhaal |
      "VIR" Verand ":=" Uitdr { "MET" Uitdr } ?
            "TOT" Uitdr Herhaal;
```

SOLANK-lusse word uitgevoer solank die lusvoorwaarde WAAR is (die toets word voor elke herhaling uitgevoer). VIR-lusse inisialiseer die betrokke veranderlike met die waarde van die eerste uitdrukking. Die waardes van die ander uitdrukkings word ook nou bereken, en nie weer voor die lus verlaat word nie. Daarna begin dit die lus herhaaldelik uitvoer, en toets voor die begin van elke herhaling of die lusbeheerveranderlike reeds groter is as die waarde van die TOT-uitdrukking is - as dit is, word die lus verlaat, andersins word dit weer uitgevoer. Die beginwaarde en TOT-uitdrukking moet van tipe HEEL, KAR of geënumereer wees en albei moet dieselfde van tipe wees. Die MET-uitdrukking moet van tipe HEEL wees, en spesifiseer met hoeveel die lusveranderlike se waarde telkens aangeskuif (of teruggeskuif as die MET-waarde negatief is) moet word, wanneer die sinne in die lus een keer uitgevoer is. Enige lus (insluitende lusse wat met die woord HERHAAL begin) mag beëindig word met die TOTDAT-spesifikasie. Die voorwaarde wat saam met TOTDAT gaan word evalueer elke keer wanneer die sinne in die lus een keer uitgevoer is (maar voordat die waarde van die lusbeheer-veranderlike aangepas word in die geval van VIR-lusse). As hierdie voorwaarde WAAR is, word die lus onmiddellik verlaat.

TerugSin = "TERUG";

Hierdie sin veroorsaak dat die prosedure waarin dit voorkom, verlaat word. As die einde van 'n prosedure bereik word, word die prosedure soortgelyk verlaat. TERUG mag net onmiddellik binne prosedures voorkom (dit mag nie in funksies of filters voorkom nie, al is die funksie of filter in 'n prosedure genes).

WaardeSin = "WAARDE" "IS" Uitdr;

Die *WaardeSin* word gebruik om 'n funksie of filter te verlaat -

dit is die enigste manier waarop 'n funksie of filter verlaat mag word. Die uitdrukking wat langs die IS voorkom, moet dieselfde tipe as die tipe van die funksie of filter wees. Die *WaardeSin* mag net onmiddellik binne 'n funksie of filter voorkom.

```
BerekenSin = "BEREKEN" Uitdr;
```

Die BEREKEN-sin evalueer net sy uitdrukking en doen niks met die resultaat nie.

```
Sin = Toewysingsin |
      SubroetineRoep |
      AsSin |
      BeskouSin |
      Lus |
      TerugSin |
      WaardeSin |
      BerekenSin;
```

UNIVERSITY
OF
JOHANNESBURG

```
Sinne = Sin *;
```

- Verklarings.

```
KonstEkw = "KONST" { Id "=" Konstante }*;

TipeEkw = "TIPE" { Id "=" AbstrTipe }*;

VerandVkl = "VERAND" VerandDef *;

VerandDef = Id ":" TipeSpes;

Verklaring = TipeEkw | KonstEkw | VerandVkl;
```

KonstEkw en *TipeEkw* word gebruik om 'n identifiseerder met, onderskeidelik, 'n konstante en 'n tipe te assosieer. Die identifiseerder kan dan enige plek gebruik word, waar die konstante of volledige tipebeskrywing gebruik sou word - die identifiseerder voorsien maar net 'n verkorte skryfwyse.

- Roetines.

```
RoetineNaam = "PROSEDURE" Id FormParms |
              "FUNKSIE" Id FormParms ":" TipeSpes |
              "FILTER" Id FormParms ":" TipeSpes "->" TipeSpes |
              "OPERATOR" Operator FormParms;
```

```
RoetineVkl = RoetineNaam { Blok | "EKSTERN" Id };
```

```
Blok = Verklaring * "BEGIN" Sinne "EINDE";
```

- Kode-eenhede.

UNIVERSITY
OF
JOHANNESBURG

```
KodeEenheid = TipeDef | Module | RoetineVkl;
```

```
DefItem = Id | Operator;
```

Die identifiseerders wat as 'n *DefItem* gebruik mag word, is, in die geval van modules, enige identifiseerder wat binne die module verklaar word, of, in die geval van 'n gebruikersgedefinieerde tipe, die naam van enige roetine (bewerking) wat binne die TIPEDEF verklaar word.

```
Module = "MODULE" Id ? ModuleLiggaam;
```

```
DefLys = "DEF" DefItem { "," DefItem }*;
```

```
ModuleLiggaam = DefLys { Verklaring | KodeEenheid }* "EINDE";

HeelKonst = Konstante;

AttrSpes = Id
           { "TIPES" "(" AbstrTipe { "," AbstrTipe }* ")" } ?
           { "MINSTENS" "(" HeelKonst ")" } ?
           { "HOOGSTENS" "(" HeelKonst ")" } ?
           "IMPLISIET" ? |
           Id "SOOS" AttrId;

AttrDefLys = "ATTR" AttrSpes { "," AttrSpes }*;

BasisTipeVkl = "VAN" TipeSpes AttrDefLys ?;

TipeParm = Id ":" AbstrTipe;

TipeParms = "(" TipeParm { "," TipeParm }* ")";

TipeDef = "TIPEDEF" Id TipeParms ?
          BasisTipeVkl ?
          ModuleLiggaam;
```

TIPEDEF word gebruik om gebruikersgedefinieerde tipes (of toestelle) te definieer. TIPEDEF en MODULE word volledig elders in die verhandeling bespreek.

```
Program = KodeEenheid* "PROGRAM" Id Blok;
```

UNIVERSITY OF JOHANNESBURG

A.5 Strekking van identifiseerders.

Identifiseerders is bekend vanaf die plek waar dit verklaar word tot aan die einde van die blok (PROSEDURE, FUNKSIE, MODULE, TIPEDEF, ensovoorts) waarbinne dit verklaar is. As dit in 'n DEF-lys voorkom is dit ook buite die MODULE of TIPEDEF bekend - tot aan die einde van die blok waarbinne die MODULE of TIPEDEF voorkom (tensy dit ook in die blok se DEF-lys voorkom). As die naam van 'n MODULE of TIPEDEF in 'n DEF-lys voorkom, word alles wat in die MODULE of TIPEDEF se DEF-lys voorkom ook na die omvattende blok uitgevoer. Die naam van 'n identifiseerder wat in 'n DEF-lys voorkom, mag eers gebruik word, wanneer dit verklaar is.

Veranderlikes begin bestaan sodra die blok waarbinne dit verklaar is, binnegegaan word en word geskrap wanneer die blok verlaat word.

UNIVERSITY OF JOHANNESBURG

A.6 Voorafgedefinieerde bewerkings.

PT voorsien 'n aantal voorafgedefinieerde bewerkings. Hierdie bewerkings word gebruik om veranderlikes van die "primitiewe" tipes wat deur PT voorsien word, te manipuleer en om in- en uitvoer te doen.

PT voorsien die prosedures *Lees* en *Skryf* om onderskeidelik 'n karakterstring vanaf die terminaal te lees en 'n karakterstring na die terminaal te skryf.

Die filters *NaStr* en *VanStr* word voorsien om heel- en reële getalle na karakterstringe en vanaf karakterstringe om te skakel. Die heelgetal 1 kan soos volg na die terminaal geskryf word :

```
Skryf(1 -> NaStr)
```

Die funksie *Len* word voorsien om die aantal karakters in 'n karakterstring te bepaal - *Len("abcde")* sal 5 lewer.

VeldNr kan gebruik word om die (relatiewe) nommer van 'n veld binne 'n rekord te bepaal. Dit kan ook op 'n attribuut toegepas word om die nommer van die oorspronklike veld wat met die attribuut geassosieer word, te bepaal.

PT voorsien die funksies *BoGrens* en *OnderGrens* om die bo- en ondergrense van 'n skikking te bepaal. Dit lewer 'n resultaat van die tipe van die indeks van die betrokke skikking.

Bylaag B – Verwysings en bibliografie.

[Aho78] Aho, AV en Ullman, JD, "Principles of Compiler Design", Addison-Wesley (1978)

[Bar80] Barnes, JGP, "An Overview of Ada", *Software Practice and Experience*, 10 (1980), 851-887, Herdruk in [Hor83] 322-354

[Bir70] Birkhoff, G en Lipson, JD, "Heterogenous Algebras", *Journal of Combinatorial Theory*, 8 (1970), 115-133

[Bri73] Brinch-Hansen, P, "Operating System Principles", Prentice-Hall (1973)

[Fre82] Freedman, RS, "Programming concepts with the Ada Language", Petrocelli Books (1982)

[Gol83] Goldberg, A en Robson, D, "Smalltalk-80 : The Language and its Implementation", Addison-Wesley (1983)

[Gri77] Gries, D en Gehani, N, "Some Ideas on Data Types in High-Level Languages", *Communications of the ACM*, 20, 6 (Junie 1977), 414-384

[Gut77] Guttag, JV, "Abstract Data Types and the Development of Data Structures", *Communications of the ACM*, 20, 6 (Junie 1977), 396-404

[Gut78] Guttag, JV en Horning, JJ, "The algebraic Specification of Abstract Data Types", *Acta Informatica*, 10 (1978), 27-52

UNIVERSITY OF JOHANNESBURG

[Hoa74] Hoare, CAR, "Monitors : an Operating System Structuring Concept", *Communications of the ACM*, 17, 10 (Oktober 1974), 549-557

[Hor76] Horowitz, E en Sahni, S, "Fundamentals of Data Structures", Computer Science Press (1976)

[Hor83] Horowitz, E, "Programming Languages : A Grand Tour", Springer-Verlag (1983)

[Hor84] Horowitz, E, "Fundamentals of Programming Languages", Springer-Verlag (1984)

[Ich82] Ichbiah, J et al, "Ada Programming Language Reference Manual", *U.S. Department of Defence MIL-STD-1815* (April 1982), Herdruk in [Hor83] 417-658

[Jen75] Jensen, K en Wirth, N, "Pascal User Manual and Report", Springer-Verlag (1975)

[Jen79] Jensen, RW en Tonies, CC, "Software Engineering", Prentice-Hall (1979)

[Led77] Ledgard, HF en Taylor, RW, "Two Views of Data Abstraction", *Communications of the ACM*, 20, 6 (Junie 1977), 382-384

[Lis77] Liskov, B, Snyder, A, Atkinson, R en Schaffert, C, "Abstraction Mechanisms in CLU", *Communications of the ACM*, 20, 8 (Augustus 1977), 564-576, Herdruk in [Hor83] 226-238

UNIVERSITY OF JOHANNESBURG

[Lis80] Liskov, B, "Modular Program Construction using Abstraction", *Abstract Software Specifications, Lecture Notes in Computer Science*, 86, D. Bjørner (red), Springer-Verlag (1980), 354-389

[Mot78] "MC6850, MC68A50, MC68B50 Asynchronous Communications Interface Adapter (ACIA)", Data Sheet, Motorola Semiconductors (1978)

[Mot81] "MC6843 Floppy Disk Controller (FDC)", Data Sheet, Motorola Semiconductors (1981)

[Mye73] Myers, GJ, "Composite Design : The Design and Modular Programs", Tecnical Report TR 002406, IBM (Januarie 1973)

[Par72] Parnas, DL, "A Technique for Software Module Specification with Examples", *Communications of the ACM*, 15, 5 (Mei 1972), 330-336

[Pop77] Popek, GJ, Horning, JJ, Lampson BW, Mitchell, JG en London, RL, "Notes on the Design of Euclid", *ACM Sigplan Notices*, 12, 3 (1977), 11-19, Herdruk in [Hor83] 252-259

[Pou85] Pournelle, J, "Computing at Chaos Manor", *Byte*, 10, 7 (Julie 1985), 309-

[Row84] Rowe, LA, "Programming Language Issues for the 1980's", *ACM Sigplan Notices*, 19, 8 (Augustus 1984), 51-61

[Sim84] Simons, GF, "Data Abstraction", *Byte*, 9, 11 (Oktober 1984), 130-

[Wes80] "FD 179X-02 Floppy Disk Formatter / Controller Family", Data Sheet, Western Digital Corporation (Mei 1980)

[Wie85] Wiener, RS en Sincovec, RF, "Two Approaches to Implementing Generic Data Structures in Modula-2", *ACM Sigplan Notices*, 20, 6 (Junie 1985), 56-64

[Wir71] Wirth, N, "The Programming Language PASCAL", *Acta Informatica*, 1 (1971), 35-63

[Wir77a] Wirth, N, "Modula : a Language for Modular Multiprogramming", *Acta Informatica*, 7 (1977), 3-35

[Wir77b] Wirth, N, "The use of Modula", *Acta Informatica*, 7 (1977), 37-65

[Wir77c] Wirth, N, "Design and implementation of Modula", *Acta Informatica*, 7 (1977), 67-84

[Wir82] Wirth, N, "Programming in Modula-2", Springer-Verlag (1982)

UNIVERSITY OF JOHANNESBURG