CWI Syllabi

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The CWI is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

MC SYLLABUS 30

COLLOQUIUM PROGRAMMEEROMGEVINGEN

J. HEERING & P. KLINT (red.)

MATHEMATISCH CENTRUM AMSTERDAM 1983

1982 CR. Categories: D.2.6, D.2.1, D.2.2, D.3.4, D.4.9, H.2.1
ISBN 90 6196 258 7

INHOUD

VOORWOORD

In de herfst van 1982 is op het Mathematisch Centrum een colloquium over programmeeromgevingen georganiseerd. De teksten van de in het kader van dit colloquium gegeven lezingen zijn in deze syllabus samengebracht.

Een programmeeromgeving is een samenhangende verzameling hulpmiddelen ter ondersteuning van de programmeerarbeid. Doel ervan is de complexiteit van te ontwikkelen programmatuur in de hand te houden en de kwaliteit van het resultaat te verhogen. In de hierna volgende *Inleiding* wordt de betekenis die in dit verband aan het begrip *complexiteit* gegeven moet worden kort uiteengezet.

In de verschillende bijdragen aan dit colloquium worden programmeeromgevingen en het proces van programmatuurontwikkeling vanuit verschillende gezichtspunten belicht. *Wolfgang Hesse* geeft een overzicht van bestaande methodes en gereedschappen voor programmatuurontwikkeling. Hij gebruikt daarbij abstractieniveau, mate van formalisering en mate van automatisering als criteria om de diverse methodes te rubriceren.

De volgende drie bijdragen hebben alle betrekking op programmeeromgevingen die het programmeren in één programmeertaal ondersteunen. *Jan Komorowski* laat zien hoe INTERLISP, een krachtige programmeeromgeving gebaseerd op LISP, de gebruiker behulpzaam is bij het ontwikkelen van programma's. De flexibiliteit en uitbreidbaarheid van INTERLISP illustreert hij aan de hand van een programmeeromgeving voor de taal PROLOG, die hij met beperkte inspanning binnen INTERLISP geïmplementeerd heeft. *Leo Geurts* gaat in op het ontwerp van een programmeeromgeving voor personal computing. Hij baseert zich daarbij op de beginnerstaal B, die op het Mathematisch Centrum ontwikkeld en geïmplementeerd is. Naast een kort overzicht van B, geeft hij een aantal uitgangspunten voor het ontwerp van een B-omgeving. *Hans Buwalda* bespreekt in zijn bijdrage de taal/programmeeromgeving FORTH die op het ogenblik op personal computers populair is.

De volgende twee bijdragen houden zich bezig met de meer algemene principes die aan programmeeromgevingen ten grondslag liggen. *Jan Heering* schetst de contouren van een programmeeromgeving waarin taaldefinities een centrale rol spelen. In deze opzet kan de voortdurende proliferatie van programmeer- en applicatietalen binnen een programmeeromgeving gecontroleerd worden, door de programmeeromgeving met behulp van taaldefinities *uitbreidbaar* te maken. *Paul Klint* gaat in op het principe van partiële evaluatie, dat gebruikt kan worden bij de implementatie van een dergelijke uitbreidbare programmeeromgeving.

Sommige softwaresystemen zijn 24 uur per dag in gebruik. Wat te doen als een dergelijk systeem gewijzigd moet worden? *Raymond Boute* laat zien hoe abstracte datatypes kunnen helpen om het probleem van de dynamische modificatie op te lossen.

Een programmeeromgeving is ook op te vatten als een informatiesysteem, waarin allerlei gegevens over programmatuur in de ontwikkelingsfase beheerd wordt. *Erik Sandewall* beschrijft een methode om operaties in informatiesystemen te specificeren en te implementeren.

Uit deze opsomming zal blijken hoe gevarieerd het onderwerp *programmeeromgevingen* is. Vele aspecten ervan kunnen in deze syllabus niet aan de orde komen. Toch hopen we dat de lezer zich een beeld zal kunnen vormen van het onderwerp en dat zijn of haar interesse gewekt wordt.

INLEIDING

Het centrale punt waar alles in de software engineering om draait is *complexiteitsbeheersing*. Wat wordt in dit verband bedoeld met complexiteit? Duidelijk niet die vorm van complexiteit die de tijdsduur of het geheugengebruik van een berekening geeft als functie van de grootte van de invoer. Nee, het gaat hier om een *structurele* of *descriptieve* complexiteit [1,2], waarvoor men een maat zou kunnen zoeken in de lengte van de programmatekst of in het aantal pagina's documentatie dat nodig is om de programmatuur te beschrijven.

In het algemeen houdt het feit, dat een programma in een gegeven taal L een bepaalde structurele complexiteit heeft, niet in, dat er geen functioneel equivalent of zelfs machtiger L-programma kan bestaan dat een lagere complexiteit bezit, al zal deze niet willekeurig klein kunnen zijn. De ondergrens is de *intrinsieke* structurele complexiteit van de te verrichten taak (relatief ten opzichte van L). Deze kan niet los gezien worden van de voor het verrichten van die taak beschikbare tijd, want zoals iedere programmeur uit ondervinding weet worden programma's langer en ingewikkelder naarmate hun efficiëntie wordt opgevoerd. Structurele complexiteit en tijdcomplexiteit kunnen tot op zekere hoogte tegen elkaar uitgewisseld worden. Een extreem voorbeeld daarvan is te vinden in [2]. Daarnaast heeft de structurele complexiteit van een programma een belangrijk dynamisch aspect in die zin, dat het uiteindelijk gaat om het specificeren en begrijpen van een *proces*.

De intrinsieke structurele complexiteit van een taak T is een ongrijpbare grootheid. Het is onbewijsbaar, dat een gegeven realisatie van T minimale structurele complexiteit bezit. Wel kan men soms laten zien dat een realisatie suboptimaal is door een minder complexe realisatie aan te geven.

Eliminatie van overbodige complexiteit is in zijn algemeenheid geen mechaniseerbaar proces. Door een ontwerpmethodologie toe te passen kan men weliswaar de complexiteit van een realisatie verminderen (zelfs in zo sterke mate, dat een ondoenlijke realisatie doenlijk kan worden), maar dit is geen panacee. Elke voldoend omvangrijke taak vereist voor zijn realisatie *specifieke inzichten* en wel des te meer, naarmate men zich een realisatie met lagere structurele complexiteit (steeds gemeten ten opzichte van een vast gekozen taal L) ten doel stelt. Een realisatie met minimale structurele complexiteit vereist maximaal inzicht.

Er zijn goede redenen om te geloven, dat het inzicht in de meeste te realiseren taken zeer onvolledig is. De structurele complexiteit van bijvoorbeeld bedrijfssystemen is verre van minimaal. De voor bedrijfssystemen geldende wetmatigheden worden onvoldoende doorzien en voor informatiesystemen is de situatie al niet veel beter getuige Erik Sandewall's opmerking in zijn bijdrage op p. 125:

"*In spite of the proliferation of information management systems, there is a striking lack of systematization or formal understanding of what this kind of software really does. In*

fact, the lack of formal understanding is probably one reason for the proliferation: various information management systems perform very similar tasks and it is reasonable to believe that there could be a design which is simpler and more powerful at the same time."

Soms heeft men een bepaald toepassingsgebied grotendeels in kaart gebracht, de inherente wetmatigheden grotendeels onderkend. In die gevallen kan men het verkregen inzicht in de vorm van een *programmagenerator* gieten. Voorbeelden zijn parsergenerators en door architectuurbeschrijvingen gestuurde codegenerator-generators. Door taaldefinities gestuurde programmeeromgevingen vallen eveneens in deze categorie.

Er zijn taken die, ondanks het feit dat ze in principe een effectieve specificatie toelaten, toch niet realiseerbaar zijn omdat hun intrinsieke structurele complexiteit te hoog is, zelfs als men de rekentijd aan geen enkele limiet bindt. Dit zijn de tegenhangers van de taken die in eindige tijd verricht kunnen worden maar toch ondoenlijk zijn omdat hun intrinsieke tijdcomplexiteit te hoog is, hoe hoge structurele complexiteit men ook toelaat. Het is moeilijk een realistisch voorbeeld te geven van zo'n niet realiseerbare taak, omdat de intrinsieke structurele complexiteit van een taak eigenlijk nooit bekend is. In de praktijk is het niet de intrinsieke structurele complexiteit, maar veeleer de door onvolledig inzicht veroorzaakte *extra* complexiteit die de grens bepaalt tussen wat realiseerbaar is en wat niet. Het bestaan van deze grens werd eerst recht duidelijk in de tweede helft van de zestiger jaren toen men moest toegeven, dat men regelmatig boven zijn macht greep bij de ontwikkeling van grote programma's. John von Neumann had er 20 jaar daarvoor al behartenswaardige opmerkingen over gemaakt:

"*We have emphasized how the complication is limited in artificial automata, that is, the complication that can be handled without extreme difficulties and for which automata can still be expected to function reliably. Two reasons that put a limit on complication in this sense have already been given. ... There is, however a third important limiting factor, and we should now turn our attention to it. This factor is of an intellectual, and not physical, character. [It is] the limitation which is due to the lack of a logical theory of automata. We are very far from possessing a theory of automata which deserves that name, that is, a properly mathematical-logical theory.*"

" ... *high complexity plays an important role in any theoretical effort relating to automata, and this concept, in spite of its prima facie quantitative character, may in fact stand for something qualitative - for a matter of principle.*" [3]

Het zwaartepunt van de structurele complexiteit van computersystemen ligt niet bij de hardware, maar bij de software. (Om misverstanden te voorkomen: de complexiteit van de hardware wordt gemeten op registerniveau inclusief eventuele microcode. De complexiteit van het materiële substraat blijft buiten beschouwing.) Om in te zien dat een doorsnee bedrijfssysteem vele malen complexer is dan de hardware van de machine waarop het draait, hoeft men slechts de lengte van de systeemprogrammatuur te vergelijken met de lengte van een beschrijving van de machine in een hardwarebeschrijvingstaal. Als een dergelijke beschrijving niet voorhanden is, kan men ermee volstaan een blik op de microcode te werpen. Deze draagt een aanzienlijk gedeelte van de complexiteit van de machine bij, maar de lengte ervan is slechts een fractie van de lengte van de systeemprogrammatuur. (Om meten met twee maten te voorkomen moet de lengte van de systeemprogrammatuur uitgedrukt worden in 'bits

microcode', maar dat betekent alleen maar dat het werkelijke lengteverschil nog iets groter is dan een oppervlakkige vergelijking suggereert.)

Zal de balans naar de andere kant doorslaan en zal het zwaartepunt van de complexiteit naar de hardware verschuiven? Dat valt voorlopig niet te verwachten. Weliswaar is er een tendens functies te verplaatsen naar microcode en 'silicium' waardoor de complexiteit van de hardware toeneemt, maar niets wijst erop dat deze zogenaamde 'verticale migratie' onevenredig sterk toeneemt in vergelijking met de groei van systeem- en applicatieprogammatuur. Wel betekent het, dat hardware-ontwerpers steeds vaker met ernstige complexiteitsproblemen zullen worden geconfronteerd.

Gezien het voorgaande is het niet verbazingwekkend dat het gebrek aan theoretisch inzicht in het karakter van complexe systemen, zoals dat door Von Neumann als derde begrenzende factor wordt genoemd, zich tot nu toe vooral op softwaregebied heeft gemanifesteerd. In het bijzonder binnen de kunstmatige intelligentie is de complexiteitsproblematiek van het begin af aan nijpend geweest. De belangrijkste vroege inspanningen om tot geïntegreerde programmeeromgevingen te komen zijn dan ook gedaan voor LISP, de voor kunstmatige-intelligentieprogramma's meest gebruikte programmeertaal. Deze ontwikkeling vindt zijn voorlopig hoogtepunt in INTERLISP (zie daarvoor de bijdrage van Jan Komorowski), maar is nog geenszins afgesloten.

Wat zal de toekomst brengen op het gebied van methodes voor complexiteitsbeheersing? *Programmeeromgeving-generators* heb ik al genoemd. Programmeeromgevingen zullen zich los moeten maken van een specifieke taal. Daarnaast zijn de contouren van een andere ontwikkeling zichtbaar: programmeeromgevingen als *expert-systemen*. Een voorproefje daarvan is te vinden in [4]. Tenslotte is er behoefte aan een wiskundig gefundeerde *structurele-complexiteitstheorie*, die quantitatief inzicht verschaft in de vraagstukken waarvoor de software engineering zich gesteld ziet en die de tegenhanger vormt van de in de afgelopen 20 jaar tot ontwikkeling gekomen theorie van de complexiteit van berekeningen.

REFERENTIES

[1] Löfgren, L., "Complexity of descriptions of systems: a foundational study", *International Journal of General Systems*, 3(1977), pp. 197-214.

[2] Chaitin, G.J., "Information-theoretic computational complexity", *IEEE Transactions on Information Theory*, **IT-20**(1974), 1, pp. 10-15.

[3] Beide citaten zijn ontleend aan: Von Neumann, J., "The general and logical theory of automata", *Collected Works*, Vol. V, pp. 288-328.

[4] Waters, R.C., "The programmer's apprentice: knowledge based program editing", *IEEE Transactions on Software Engineering*, **SE-8**(1982), 1, pp. 1-12.

Jan Heering

METHODS AND TOOLS FOR SOFTWARE DEVELOPMENT - A WALK THROUGH THE TECHNOLOGY LANDSCAPE

Wolfgang Hesse
SOFTLAB GmbH*

ABSTRACT

This paper surveys important existing software engineering techniques. Techniques are classified using a three-dimensional scheme with the axes *abstraction* (= distance to the executing machine), *linguistic freedom*, and *automatization*. In the first sections individual techniques for the analysis, definition, design, and validation of software systems are considered. Software engineering environments, i.e. integrated systems supporting major parts of the software production process, are the subject of the last sections.

1. THE SOFTWARE TECHNOLOGY LANDSCAPE - A CLASSIFICATION SCHEME FOR SOFTWARE DEVELOPMENT TECHNIQUES

'HIPO', 'JACKSON', 'Structured Design', 'HDM', 'Data Abstraction', 'Structured Walkthrough', 'Code Inspection' are some key terms arbitrarily taken from the wide variety of software development techniques being offered. Facing the jungle of coexisting and competing techniques, software engineers, managers and users often have difficulty in selecting the proper methods or tools for their specific problem.

Existing survey papers and comparative studies use the phases of the software life cycle as their sole criterion for classifying software development techniques. This approach simplifies things too much as can easily be seen from the fact that it brings such diverse techniques as 'HIPO', 'JACKSON', 'HDM' and 'Data Abstraction' all under the same heading, namely that of design techniques.

Other classification criteria which are at least as important as the phases of the life cycle become apparent from a brief review of the history of software technology.

Taking assembly-like programming on the first computers as our point of departure, at least three essential extensions of programming technology can be distinguished:

First, the ideas of *structured programming* and the introduction of *high level languages* led to a higher *abstraction level.* Programming was no longer limited to the 'low' level associated with the machine but was lifted to 'higher', more problem-oriented, more 'abstract' levels.

Later on, it was realized, that programming is but a part of a larger process comprising many more tasks. This 'software engineering' point of view led to an

* SOFTLAB GmbH, Arabellastrasse 13, D-8000 München 81, BRD.

increase in the *linguistic freedom.* Programs written in a purely formal language were more and more accompanied by preparatory documents, specifications and validations. These are usually written in natural or semi-formal language.

Finally, the growing complexity and variety of software products made their *automatic* (or semi-automatic) production a key problem. Administration, editing and transfer of texts, consistency and completeness checks, simulations, and even methodological guidance are tasks, which can, at least partly, be taken over by a computer. *Software Engineering Environments* (cf. [HUE81]) are systems which combine tools for the solution of these tasks and which support the transition between them.

In the following overview of the software technology field, we shall use

- abstraction degree (or 'distance' to the executing machine),
- linguistic freedom, and
- degree of automatization

as classification criteria. They form the coordinate axes of the so-called 'Software Technology Landscape' (STL). Languages, techniques, and environments for software development will be classified using this three-dimensional scheme.

Of course, such a classification is not possible without introducing major simplifications and sacrificing much detail. This is the price to be paid for compressing such a complex field into a three-dimensional scheme. On the other hand, this approach at least seems to be an improvement over the one-dimensional life-cycle approach.

The rest of this paper is organized in 6 sections. In section 2 and 3 the first dimensions of the STL (abstraction and linguistic freedom) are introduced. In section 4 the STL is used for classifying software design techniques. Validation techniques can also be taken into account after an appropriate extension of the STL (section 5). Finally, in sections 6 and 7 the automatization dimension is introduced and some software engineering environments are discussed.

2. THE FIRST DIMENSION: ABSTRACTION

Following the general program outlined in the introduction, we start construction of the STL by introducing its first axis, the *abstraction axis* (fig.1). This axis ranges from the closest-to-the-machine programming level (the 'machine level') through the 'low' level to the 'high' (or problem-oriented) level.

If we do not restrict this scale to algorithmic levels but also consider non-algorithmic formulations, i.e. expressions which cannot directly be executed by an interpreting machine, we come to the 'very high level' end of the abstraction spectrum.

Programming languages (including 'very high level' languages) may be classified in this scheme by straight lines mirroring their 'range.' As an illustration, three example languages (assembler, FORTRAN, and ALGOL 68) are shown in fig. 1. We omit further examples as well as a discussion of the classification itself in order to leave room for the following considerations which are more central to the subject of this paper.

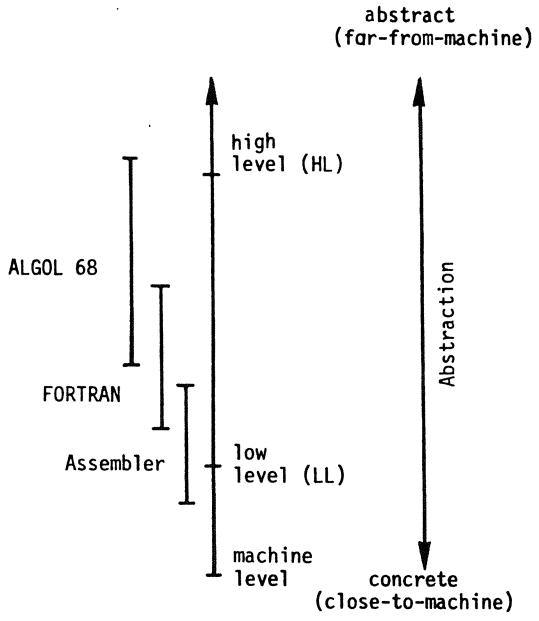


Fig. 1: The abstraction spectrum of programming languages

3. THE SECOND DIMENSION: LINGUISTIC FREEDOM

One of the major results of the 'software engineering' discussion was the realization that a software system not only consists of programs, i.e. statements written in a completely formalized, machine-intelligible language, but that it also comprises preparatory, accompanying and concluding documents written in semi-formal or natural language. These documents which are partly or exclusively intended for human beings are as important as the pure programs themselves.

We conclude that 'linguistic freedom' is a second, independent criterion for the classification of software engineering techniques. Both criteria - abstraction degree and linguistic freedom - form the axes of a two-dimensional scheme, the 'program development plane' (PDP) shown in fig. 2.

Linguistic freedom ranges from 'code' (totally formalized language) to 'ideas', i.e. mental constructs not even written down on paper.

Let us rest for a moment and consider the way a software developer is going through the PDP. His starting point is the upper left hand corner of the diagram: very far from the machine, full of ideas. His goal is the lower right corner: a completely formalized program executable by a concrete machine.

Of course, there are an infinite number of ways from one point to the other. Of these, we shall elucidate two extreme cases:

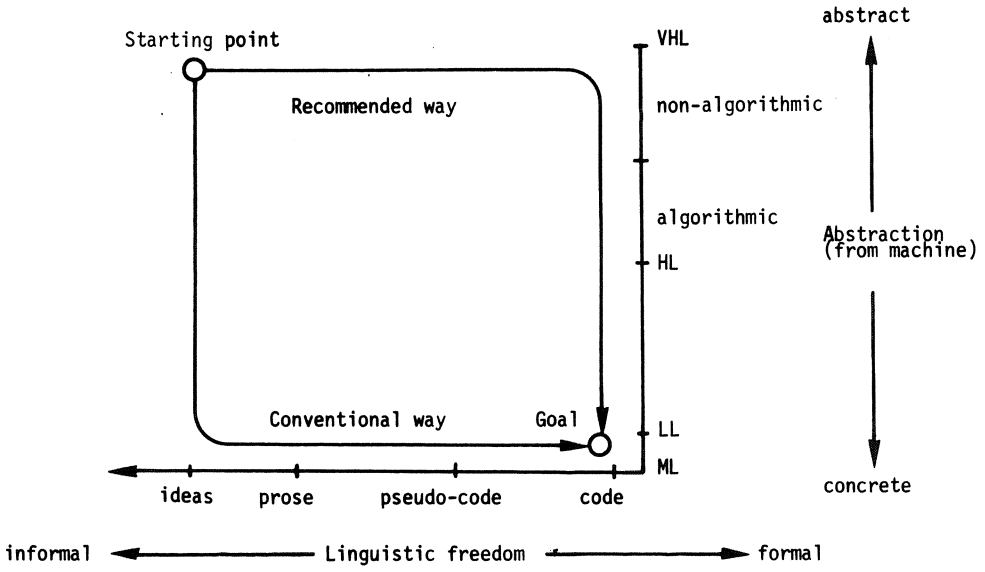


Fig.2: The Program Development Plane

First, the 'conventional way' (cf. fig. 2). This way tries to cling to the axis of linguistic freedom as closely as possible. This means, the lowest level is reached very fast on the 'early programming highway' without any time spent in writing down underlying ideas, let alone in attempts at formalization. But now, at the lower left hand corner of the diagram, the painful and dangerous 'tumbling programmer's ridge trail' starts along the formalization axis, where projects get into loops or even fail completely.

In contrast to the conventional way, the other extreme would be to follow the abstraction axis. This is the 'recommended' or 'ideal' way. Traveling this way, ideas are formalized as much as possible on a level very far from the machine before lower levels are approached taking the formalization as a point of departure. We might also call this way the 'data abstraction express way' because it reflects the basic idea of data abstraction: first describe (in a formal way) *what* to do, then *how* to do it. In terms of SOFTLAB's project model, we call the starting point 'requirements', the first step 'specification', and the second step 'construction' ([HES80], fig. 3).

This process can also be iterated on several modularization levels (system, components, modules). In this case, specifications and constructions resemble the steps of a staircase.

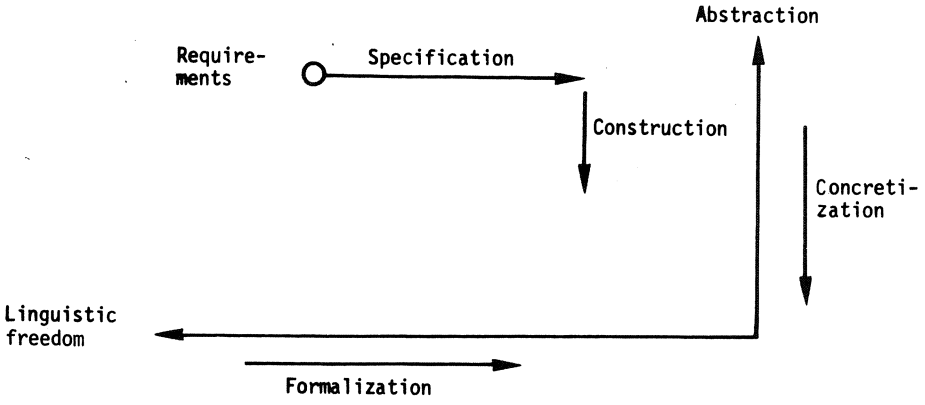


Fig. 3: Design steps represented in the PDP

4. SOFTWARE DEVELOPMENT TECHNIQUES

In fig. 4 an attempt is made to locate some software development techniques in the PDP. The techniques shown are listed and commented on in tab. 5. For a discussion of the classification see [HES81], for details on individual techniques see the references.

5. EXTENSION OF THE SECOND DIMENSION: THE VALIDATION PLANE

The PDP is well-suited for the representation of software design and implementation techniques, but there is no room in the PDP for validation, integration and installation.

This room may, however, be provided by reflecting the PDP with respect to the abstraction axis and thus extending it to the combined 'program development & validation plane' (PDVP, fig. 6). Such a reflection is a consequence of a symmetric conception of the software development process as can be found in B. Boehm's newer life cycle [BOE79] and in the project model [HES80]. In this view, the specification of the function of a software component is dual to its validation, the decomposition of a component during the construction step of the design phase is dual to its composition during the linkage step of the integration phase.

In our larger plane, the formality axis is extended so as to provide room for examination techniques ranging from proof (formal) over spot tests (semi-formal) to hope (very informal).

Similarly to the left hand side of the PDVP, we distinguish techniques with different degrees of formality on its right hand side. *Verification* (in the strict sense of mathematically founded proofs) is the most formal technique, *reviews* and *inspection* are rather informal, and in between lies the wide field of *test* and *simulation* techniques.

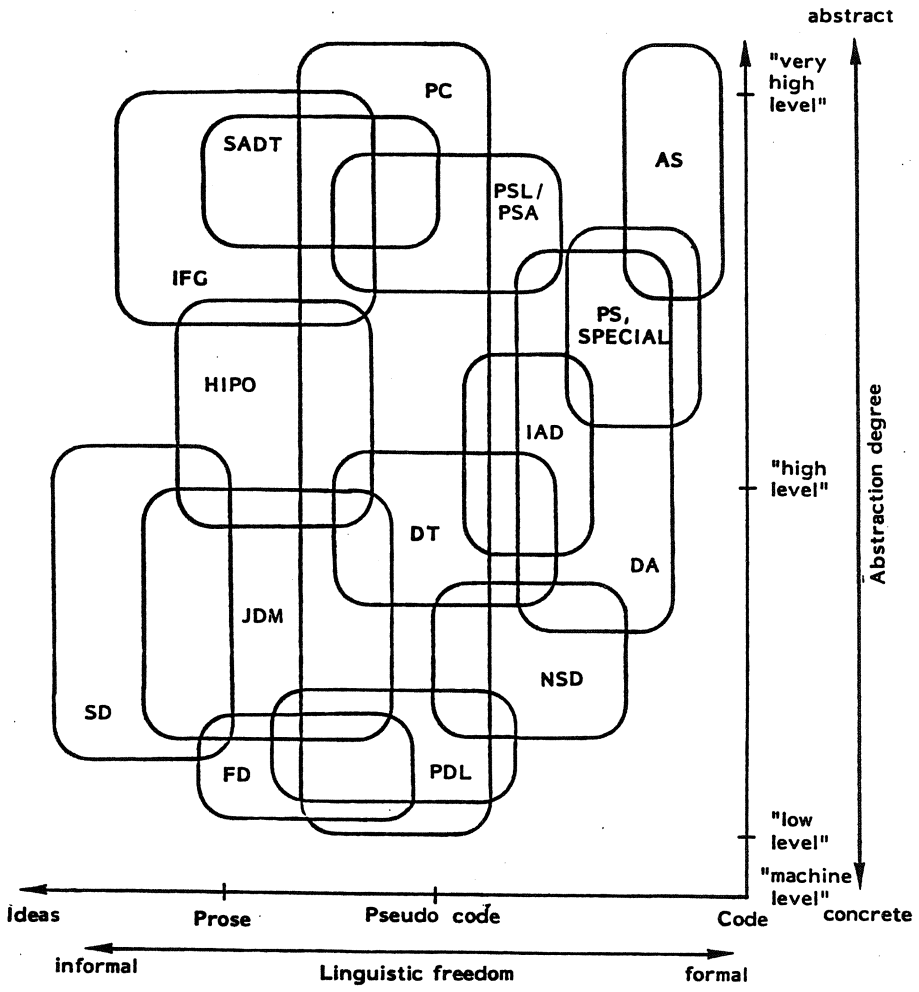


Fig.4: Software development techniques in the PDP (for explanations see tab.5)

Technique Short name	Long name	Origin	Remarks	References
SADT	Structured analysis and design technique	SOFTECH 1976	Diagram representation of activities, data and their interrelationships	/ROS 77/
PSL/ PSA	Problem statement language/ problem statement analyser	Univ. of Mich. 1976	Frame language (PSL) for the description of objects and their relations; analysis tool (PSA)	/T-H 77/
IFG	Infograms	SOFTLAB 1979	Matrix representation of requirements, informations, data and system components and their relations	/FRO 80/
HIPO	Hierarchy-plus-input-process-output	IBM 1974	3-column tables (input/process/output) + overview diagrams	/IBM 74/
JDM	JACKSON design methodology	M. JACKSON 1975	Tree representation of data and control structures, pseudo-code ("schematic logic")	/JAC 76/
SD	Structured design	YORDON/ CONSTANTINE 1975	Methodology including transform analysis, transaction analysis, top down design	/Y-C 75/
PC	Pseudo code		Collective term for all kinds of mixtures between formal ("code") and natural language; example: PDL, see below	
PDL	Program design language	CAINE, FABER, GORDON 1975	Simple pseudo-code with formalized control structures	/C-G 75/
FD	Flow diagrams		Graphical representation of sequential control structures	

Tab. 5: Software development techniques

(Tab. 5, continued)

Technique Short name	Long name	Origin	Remarks	References
NSD	NASSI-SHNEIDERMAN diagrams	NASSI, SHNEID. 1973	Two-dimensional pseudo-code with formalized control structures	/N-S 73/
DT	Decision tables		Table representation of condition/action dependencies	/STR 77/
IAD	Interaction diagrams	SOFTLAB 1977	Finite state automata for dialogue description	/DEN 77/
PS	PARNAS-specification	PARNAS 1972	Specification technique for module functions in terms of their input/output behaviour	/PAR 72/
SPECIAL	Specification and assertion language	SRI 1976	Language framework for PS (see above)	/R-R 76/
DA	Data abstraction	many	Principle: encapsulated data, may only be accessed by well-defined operations	/L-Z 75/ /GUT 77/
AS	Algebraic specification	DA, algebra	"Abstract data types" are treated as heterogeneous algebras	/ADJ 77/

How does the software engineer proceed through the PDVP on the validation side? Normally, he starts in the 'spot test' area, testing his modules in a more or less systematic manner. Tested modules are linked together so as to form larger units ('subsystems'), these are tested again and so on. In a certain sense, this is an abstraction process. Each transition to a larger subsystem requires taking a step back, i.e. disregarding details and adopting a more global viewpoint. The last step is the acceptance test of the whole system on the user's site. This is a 'high level' validation in the sense that acceptance tests principally exercise the user interface.

Of course, top-down testing is also possible. This means starting in the simulation area. Components that are not yet available are simulated by stubs, which are gradually replaced by the components themselves.

The more systematically tests are conducted or the further to the left they can be located in the validation plane, the more safety they provide. Of course, verification is the safest validation method. But correctness proofs are relative. They depend on the correctness of the specification which is used as an unshakable basis. Even if this process is iterated, i.e. if specifications are verified against 'over-specifications', there

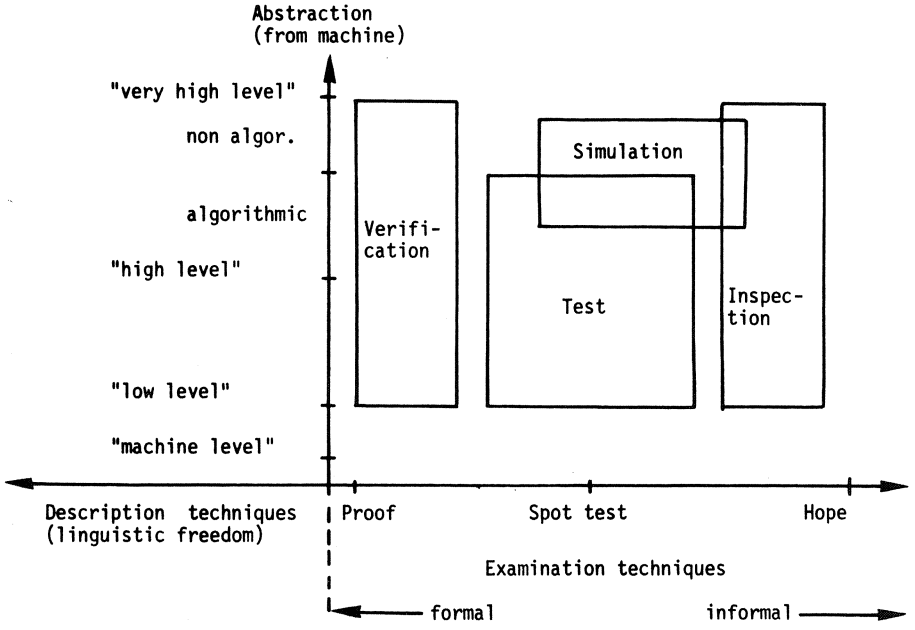


Fig. 6: Extension of the PDP: The validation plane

must be an end of the chain and this is the hope or trust of the software engineer that his product will satisfy the requirements. In this way, the upper right hand corner of the PDVP is reached.

Fig. 7 is an attempt to classify some of the better known validation techniques. Classification is more difficult here than on the design side because the subject is less popular. The classification of test techniques chiefly follows Myers who gives a very helpful overview of the field [MYE79].

The techniques are listed and briefly commented on in tab. 8. For a more detailed discussion, see [HES81] and the references.

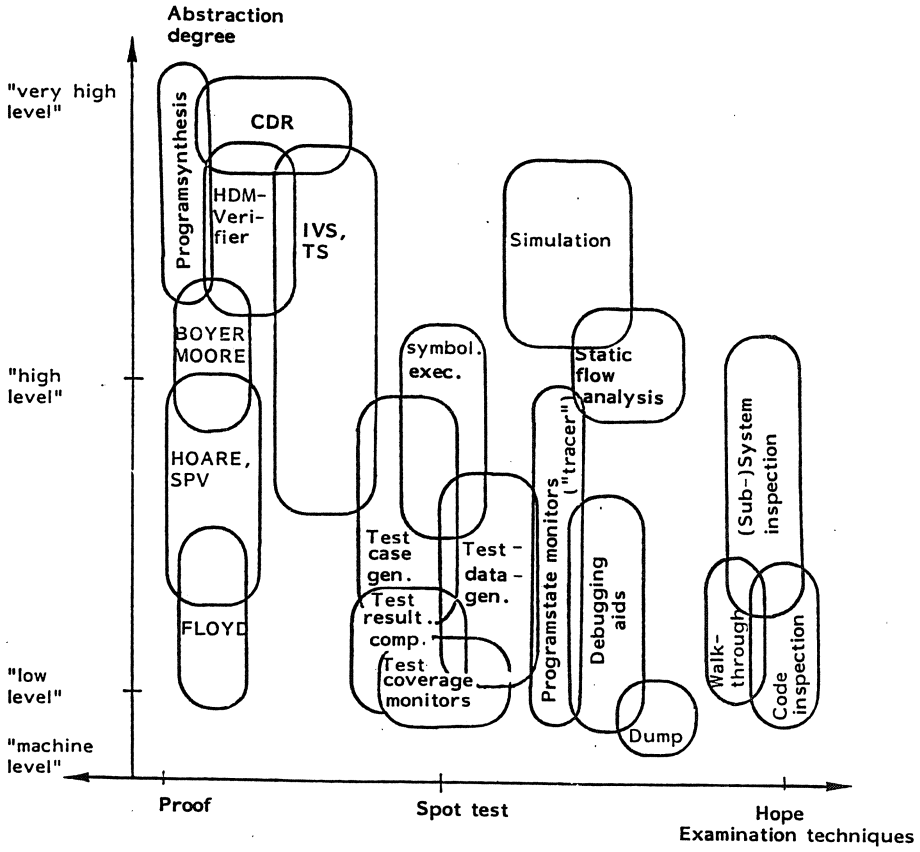


Fig.7.: Validation techniques

Technique	Remarks	References
FLOYD	The classical verification technique for flowchart programs	/FLO 67/
HOARE	Adaption of FLOYD's technique to higher programming languages	/HOA 69/
SPV (Stanford Pascal Verifier)	Verification system for PASCAL programs based on HOARE's technique	/ILL 75/
BOYER/MOORE	LISP-verification system	/B-M 75/
HDM verifier (SRI 1978)	SPECIAL verification system	/LRS 79/
CDR (correctness of data representation)	Correctness proofs for implementation of abstract data types against their specifications	/HOA 72/ /GUT 77/
Program synthesis	Programs are not verified against a specification but generated from it ("synthesized")	/M-W 77/ /EIG 80/ /BIB 80/
IVS (Interactive verification systems)	Systems for parallel development of programs and their correctness proofs from specifications	/DIJ 76/ /R-S 80/
TS (Transformation systems)	Interactive systems for program development by stepwise transformations	/D-B 76/ /BAU 77/
Symbolic execution	Interpreters using variables ("symbolic values") as input instead of alphanumeric values	/MYE 79/
Test case generation	Automated generation of test cases from specifications	/GMH 81/ /MYE 79/
Test data generations	Automated generation of test data (for present test cases)	
Test result comparison	Automated comparison of test results with predicted results or those obtained from previous test runs	
Test coverage monitors	Measurement of execution frequency of selected control structures	/MYE 79/
Program state monitors ("tracer")	Monitors showing the values of selected variables under certain conditions (debugging aid)	
Debugging aids	Tools for the generation of extra output (for debugging purposes) during program execution (e. g.: tracers)	

Tab. 8: Validation techniques

Tab. 8 continued

Dump	Sinking programmer's last rescue	
Static flow analysis	Static checks for possible dynamic errors (undefined values, unreachable code, inconsistent interfaces)	/MYE 79/
Inspection	Expertise of a program/system by a (human) inspection board	
Walkthrough	Dynamic inspection of a program. Inspectors "play computer"	

6. THE THIRD DIMENSION: AUTOMATIZATION DEGREE

In the previous sections, two aspects which are essential to the understanding of Software Engineering Environments have not yet been considered:

- How can the whole software production process (from analysis via definition, design, implementation and integration to installation, operation and maintenance) be supported in a universal and coherent manner?
- How can the software production process (as a whole) be automated?

As a first step, we can define a Software Engineering Environment (SEE) as region of the PDVP (the area which is 'covered' by the SEE). This approach, however, ignores the automatization aspect.

For this reason, we extend our classification scheme once more and introduce as a third axis the degree of automatization. On this axis, we mark automatization facilities of growing power or sophistication.

A possible spectrum ranges from

- text administration

via facilities for

- text processing,
- product administration,
- information processing,
- syntactic analysis,
- semantic analysis, and
- generation of products or subproducts

to facilities for

- decision support and decision making.

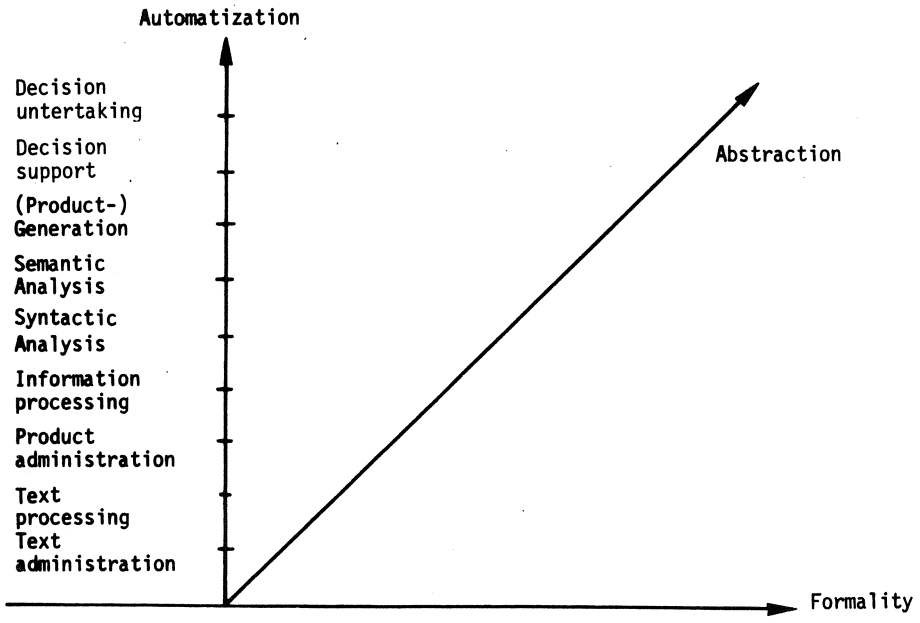


Fig. 9. The main axes of the Software Technology Landscape

The three criteria, viz. abstraction degree, linguistic freedom, and automatization degree, together form the *Software Technology Landscape* (STL, fig. 9).

Fig. 10 shows some typical components of SEE's in the STL. The pyramidal form of this representation reflects the fact, that the more formal the description and/or examination means are, the more powerful or sophisticated the corresponding tools are.

The simplest form of computer support is text administration, i.e. text input, storage and output. This is normally done by a file system.

To the text administration functions a text editor adds text processing functions such as copying of words, records or paragraphs, as well as insert, delete, search, replace and layout functions, etc.

Text administration should not be confused with product administration which is much more. Software products are texts with attributes specifying their origin, history, type, interconnections, version, etc. Products are administered in a project library or - if management information is also included - by a project library [DEN79, D-H80].

For the following automatization levels, we distinguish tools for software design and coding (left hand side of fig. 10) and for software validation (right hand side of fig. 10).

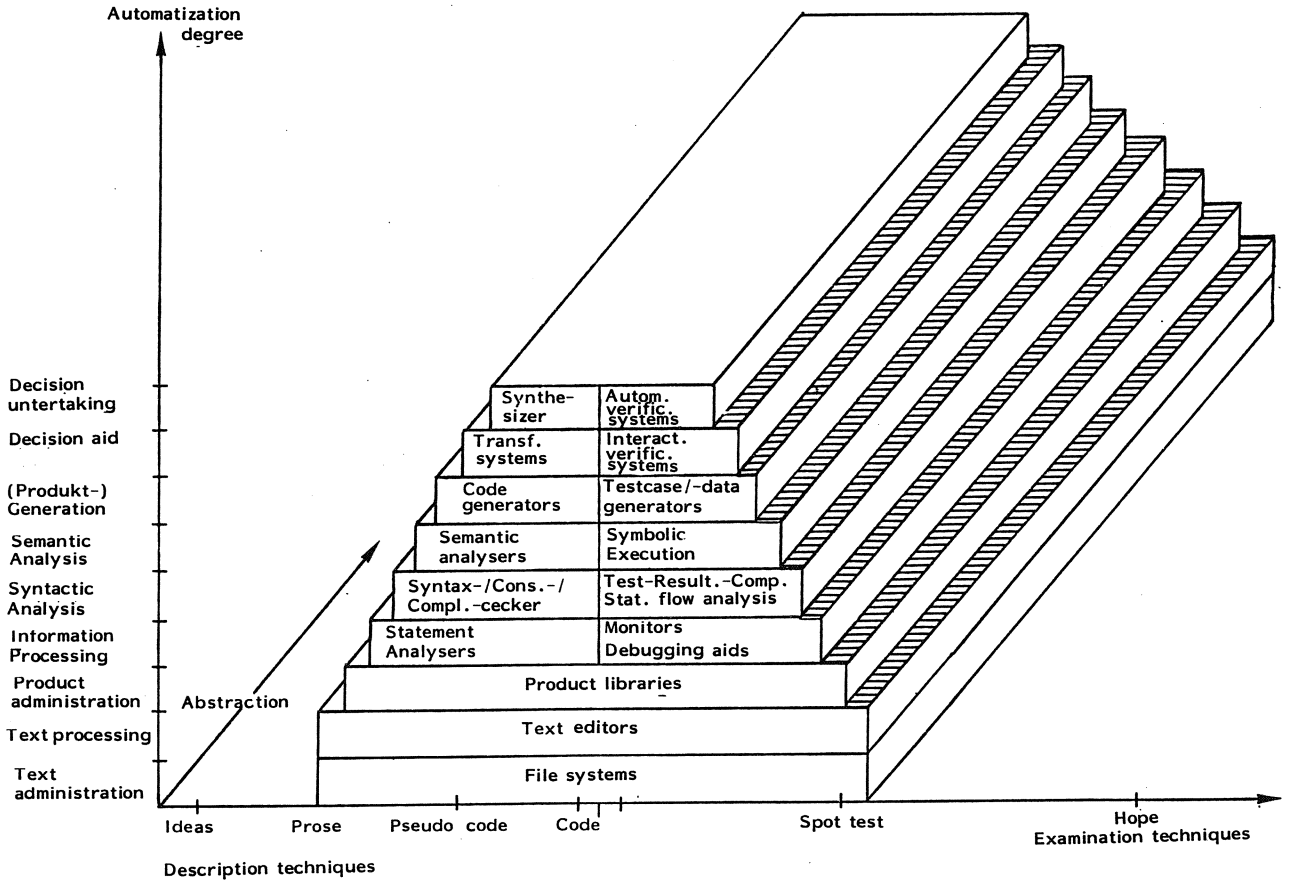


Fig.10 Software development tools in the technology landscape

Information processing means the (possibly partial or incomplete) analysis of given texts and output of analysis results as errors, inconsistencies, cross references, graphical or tabular representations, etc. The 'problem statement analyzer' (PSA) is an example of a corresponding tool [T-H77].

On the validation side, test coverage monitors, program state monitors and similar tools represent this automatization level.

The next three levels on the design side correspond to the classical compiler functions: syntactical analysis, semantical analysis, code generation. On the validation side, test result comparison tools and static flow analyzers, symbolic interpreters, test case and test data generators represent this level.

The top of the pyramid is occupied by tools which support the software engineer in making his decisions or even try to make them for him automatically. To the first category belong transformation systems and interactive verification systems, to the second program synthesizers and automated verification systems.

7. SOFTWARE ENGINEERING ENVIRONMENTS

With the preparations of the previous sections in mind, we are now ready to define a *Software Engineering Environment* (SEE) as a (connected or disconnected) region of the STL.

Normally, an SEE is based on a 'software life cycle' or 'project model' which defines the phases, activities and results of a software project. We will not elaborate on this subject here, but refer to the literature, e.g. [HES80]. Another common component of an SEE, which can only be mentioned here, is a set of management techniques, i.e. means and standards for project planning and control, effort estimation and accounting, quality assurance and evaluation of completed projects.

Considering the field of SEE's, we distinguish two main categories: homogeneous and heterogeneous SEE's.

A *homogeneous* SEE is grouped around one central technique, language, method or tool. Normally, this central technique was also the historical origin of the whole SEE. Examples of homogeneous SEE's (cf. tab. 11) are CDL2 (center: Koster's Compiler Description Language) and APSE (center: the ADA language). On the other hand, a *heterogeneous* SEE is a combination of several components of equal or similar importance. These are used for different activities of software development or even optionally for the same activity.

It is still too early to judge the relative merits of these two approaches. A short discussion of the subject is found in [HES81]. The universities seem to prefer the homogeneous approach while industry seems to prefer the heterogeneous approach.

Tab. 11 is a summary of some better known SEE's. Most of them have been presented at the Lahnstein GMD workshop [HUE81]. For more elaborate summaries see [H-M81] and [HES81].

Figures 12 and 13 show as an example the STL representation of one specific SEE, the 'Software Engineering Technology' (S/E/TEC) of SOFTLAB. In terms of construction diagrams, fig. 12 is the plan view and fig. 13 the vertical section of S/E/TEC in the STL. 'EDDA' stands for 'Design Dialects for Data Abstraction', 'TUS' for 'Test Support System.' The other S/E/TEC components have been explained in section 4. The whole system has been presented and discussed in [DHN81].

SEE identification short	long	Author(s) Institution	Support of Production AN SE MI SI BW DF KE SS IN	Proj.man. PL QS SZ BS	Doc. Ed PB	Remarks special aims, applications
CDL2	Compiler Description Language	KOSTER et al.	CDL2			Compiler construction, Systems programming
COSY	Concurent Systems	LAUER et al.	COSY			Concurrent systems, Operation system development
CIP	Computer-aided, Intuition-guided Programming	BAUER et al.	CIP-L			Program development by transformations
PDS	Programming Development System	CHEATHAM et al.	EL1		DB	Program development by transformations
APSE	ADA Programming Supp. Environment	U. S. DoD	ADA			Standardization / manufactured SW components
GANDALF	Software Development Environment for ADA	HABERMANN, PERRY	SCG, JPC	PM		Comfortable project library for ADA
HDM	Hierarchical Development Methodology	SRI	SPECIAL, HSL, ILPL			PARNAS specifications
SDS	Software Development System	TRW	RSL, PDS		DB	large military projects
CADES	Computer Aided Design Eval. System	ICL	SDL, S3		IDMS	Operation system development
DREAM	Design, Realization Evaluation and Modeling System	Univ. of Col Boulder	DDN		DB	Concurrent systems
UNIX/ PWB	Programmer's Workbench	BELL Labs.	C, TD		SCCS, MRCS	Tool box
AIDES	Automated Interactive Design and Eval. System	HUGHES Aircraft	SCG (SD)	DQMS		Aircraft construction (graphics)
ARGUS	Advanced Software Eng. Workbench	BOEING	STRADA, DYNA			Aircraft construction
SWB	Software Workbech	TOSHIBA	I, II, III, IV	P	I/II	Software-"factory" for real-time applications
SDEM/ SDSS	Software Devel. Eng. Meth. / Support System	FUJITSU	MDL/MDA		PB	Standardization of the SW-production
S/E/TEC	Software Eng. Technology	SOFTLAB	IFG, IAD,ET, EDDA, STG,PC, TUS, IMAS	PF-TEC	PB, PET	Universal, integrated SW-production environment

Tab. 11 : Software Engineering Environments

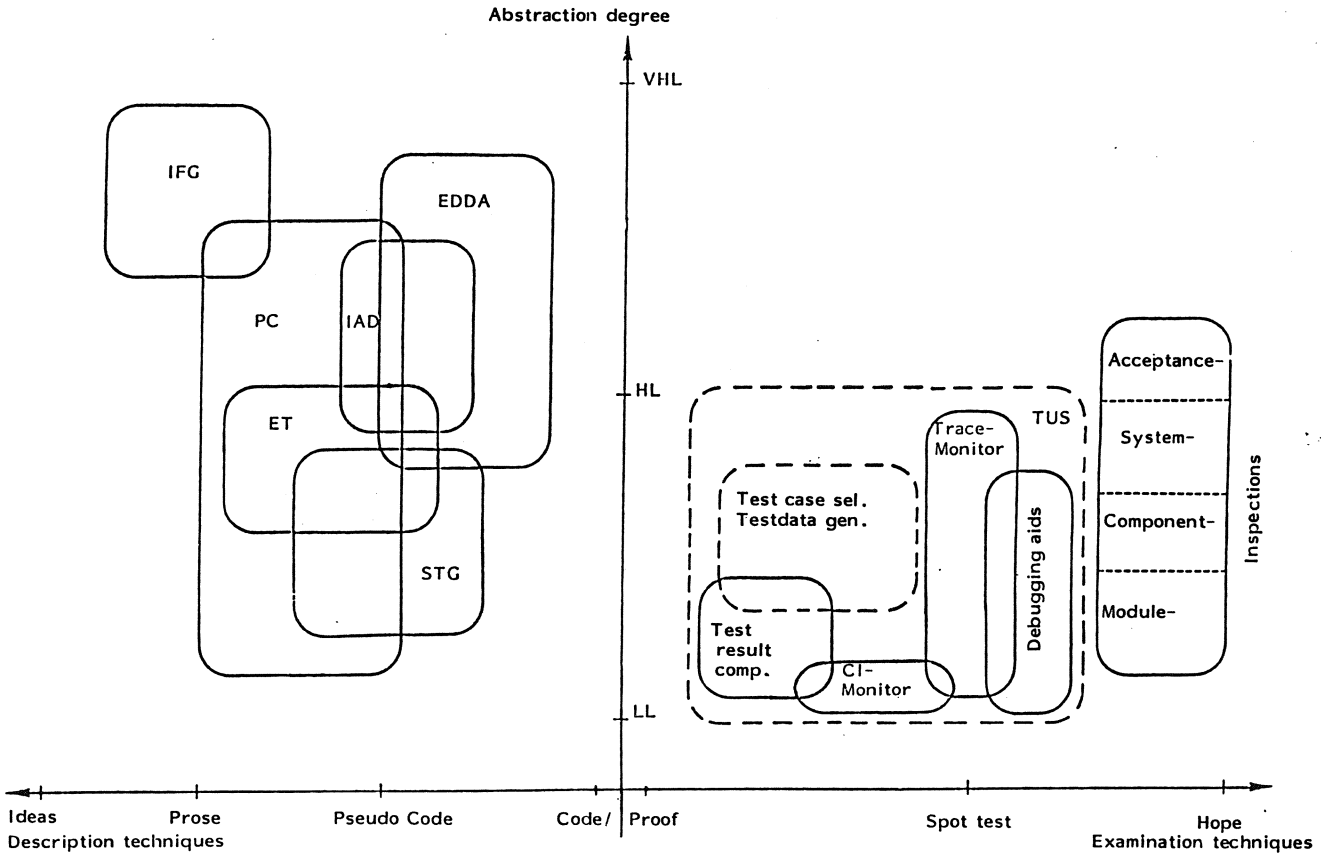


Fig. 12: The S/E/TEC Production techniques in the STL (plan view)

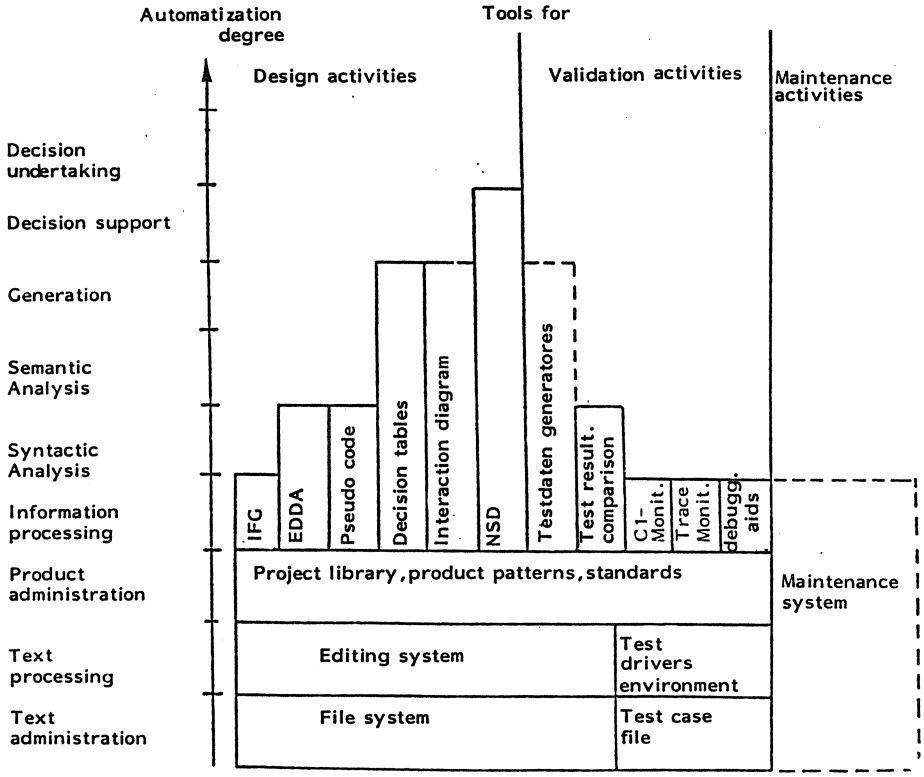


Fig.13: The S/E/TEC production techniques in the STL (vertical section)

8. FINAL REMARKS

In the previous sections, the successive development of the 'Software Technology Landscape' from its origins up to its present form has been elucidated. The definition and use of higher level programming languages, the development and use of techniques for pre- and post-programming activities and the creation and application of software engineering environments were milestones in that development.

To give the technology landscape a structure, to locate several techniques and tools in it, to reveal dependencies and interconnections and to give a summary of the software technology field, were some of the aims of this paper.

The realization that there are more or less arduous ways through the landscape and finding them are among the major methodological results of the last years.

Designers of an up-to-date software engineering environment oriented towards the needs of its users have to keep these results in mind. The SEE has to offer its users maximal comfort and travel support for their journey through the technology landscape. The creation and further development of SEE's obeying these criteria will probably dominate the software technology field for years to come.

ACKNOWLEDGEMENT

This work was partiallly funded by the German Ministry of Research and Technology (grant no. 0815054 A). I thank my colleague W. Ross for a critical review of the English translation.

BIBLIOGRAPHY

[ADJ77] Goguen, J.A., Thatcher, J.W., Wagner, E.G., & Wright, J.B., "Initial algebra semantics and continuous algebras," *JACM*, **24**(1977), 1, pp. 68-95.

[B-T79] Babb, R.G., & Tripp, L.L., "An approach to defining areas within the field of software engineering," *ACM SIGSOFT SE Notes*, **4**(1979), 4, pp. 9-17.

[BAU77] Bauer, F.L., Partsch, H., Pepper, R., & Wössner, H., "Techniques for program development," **in** *Software Engineering Techniques*, Infotech State of the Art Report **34**, 1977, pp. 27-50.

[BAU78] Bauer, F.L., Broy, M., Gnatz, R., Hesse, W., & Krieg-Brückner, B., "A wide spectrum language for program development," *Third Intern. Symp. on Programming*, Paris, 1978.

[BIB80] Bibel, W., "Syntax-directed, semantics-supported program synthesis," *Artificial Intelligence*, 14(1980), pp. 243-261.

[BOE79] Boehm, B.W., "Guidelines for verifying and validating software requirements and design specifications," *EURO IFIP 79*, North-Holland, 1979, pp. 711-719.

[B-M75] Boyer, R.S., & Moore, J.S., "Proving theorems about LISP functions," *JACM*, **22**(1975), 1, pp. 129-144.

[C-G75] Caine, S.H., & Gordon, E.K., "PDL - a tool for software design," *Proc. Nat. Comp. Conf. AFIPS*, 1975, pp. 271-276.

[D-B76] Darlington, J., & Burstall, R.M., "A system which automatically improves programs," Acta Informatica, **6**(1976), pp. 41-60.

[DEN77] Denert, E., "Specification and design of dialogue systems with state diagrams," *Proc. Int. Comp. Symp. 1977*, North-Holland, 1977, pp. 417-424.

[DEN79] Denert, E., "The project library - a tool for software development," *Proc. 4th ICSE München*, IEEE Cat. No. CH1479-5/79, 1979, pp. 153-163.

[D-H80] Denert, E., & Hesse, W., "Projektmodell und Projektbibliothek: Grundlagen für zuverlässiger Software-Entwicklung und Dokumentation," *Informatik-Spektrum*, **3**(1980), 4, pp. 215-228.

[DHN81] Denert, E., Hesse, W., & Neumaier, H., "S/E/TEC - An environment for the production of reliable software," *Third ECI Conference on Trends in Information Processing Systems*, München, 1981, Lecture Notes in Computer Science **123**, Springer, 1981, pp. 65-81.

[DIJ76] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, 1976.

[EIG80] Eigemeier, H., Knabe, C., Raulefs, P., & Tramer, K., "An expert system for automatic coding of abstract data type specifications," GI - 10. Jahrestagung, Informatik-Fachberichte **33**, Springer, 1980, pp. 431-441.

[END78] Endres, A., "Methoden der Programm- und Systemkonstruktion; ein Statusbericht," **in** GI - 8. Jahrestagung, Berlin, Informatik-Fachberichte **16**, Springer, 1978, pp. 29-73 and *Informatik-Spektrum*, **3**(1980), 3, pp. 156-171.

[FLO67] Floyd, R.W., "Assigning meaning to programs", **in** Proc. Symp. on Applied Mathematics, Vol. 19, American Mathematical Society, 1967, pp. 19-32.

[F-W76] Freemann, P., & Wasserman, A., (Eds.), *Tutorial on Software Design Techniques*, IEEE Computer Society Cat. No. 76CH1145-2C, 1976.

[FRO80] Frölich, R., "Das Infogramm: Eine Technik zur Analyse, Definition und zum Entwurf van DV-Systemen," SOFTLAB GmbH, München, Internal Report, 1980.

[GMH81] Gannon, J., McMullin, P., & Hamlet, R., "Data-abstraction implementation, specification, and testing," *TOPLAS*, **3**(1981), 3, pp. 211-223.

[GRI76] Gries, D., "An illustration of current ideas on the derivation of correctness proofs and correct programs," *IEEE Trans. on Software Engineering*, **SE-2**(1976), 4, pp. 238-244.

[GUT77] Guttag, J., "Abstract data types and the development of data structures," *CACM*, **20**(1977), 6, pp. 396-404.

[HES80] Hesse, W., "Das Projektmodell - eine Grundlage für die ingenieurmässige Software-Entwicklung," GI - 10. Jahrestagung, Informatik-Fachberichte **33**, Springer, 1980, pp. 107-122.

[HES81] Hesse, W., "Methoden und Werkzeuge der Software-Entwicklung - Ein Marsch durch die Technologie-Landschaft, Arbeitstagung *Werkzeuge der Programmiertechnik*, Informatik-Fachberichte **43**, Springer, 1981, pp. 113-153, and *Informatik-Spektrum*, **4**(1981), 4, pp. 229-245.

[HOA69] Hoare, C.A.R., "An axiomatic basis for computer programming," *CACM*, **12**(1969), 10, pp. 576-583.

[HOA72] Hoare, C.A.R., "Proof of correctness of data representations", *Acta Informatica*, **1**(1972), 4, pp. 271-281.

[H-M81] Hausen, H.L., & Müllerburg, M., "Software-Produktions-Umgebungen: Entwicklungsstand und Trends," Arbeitstagung Werkzeuge der Programmiertechnik, Informatik-Fachberichte **43**, Springer, 1981, pp. 1-27.

[HUE81] Hünke, H., (Ed.), *Symposium on Software Engineering Environments*, North-Holland, 1981.

[IBM74] HIPO - A Design Aid and Documentation Technique, IBM Report GC 20-1851-0, 1974.

[ILL75] Igarashi, S., London, R.L., & Luckham, D.C., "Automatic program verification I: A logical basis and its implementation," *Acta Informatica*, **4**(1975), 2, pp. 145-182.

[JAC76] Jackson, M.A., *Constructive Methods of Program Design*, Lecture Notes in Computer Science **44**, Springer, 1976.

[LRS79] Levitt, K.N., Robinson, L., & Silverberg, B.A., *The HDM Handbook*, Vols. I, II, III, SRI International, Menlo Park CA, 1979.

[L-S78] Ludewig, J., & Streng, W., "Ueberblick und Vergleich verschiedener Mittel für die Spezifikation und den Entwurf von Software", Bericht KfK 2506, Kernforschungszentrum Karlsruhe, 1978.

[L-Z75] Liskov, B., & Zilles, S., "Specification techniques for data abstraction," *IEEE Trans. on Software Engineering*, **SE-1**(1975), 1, pp. 7-18.

[M-W77] Manna, Z., & Waldinger, R., "The automatic synthesis of recursive programs," *Proc. Symp. on Artificial Intelligence and Programming Languages*, 1977, pp. 29-36.

[MYE79] Myers, G.J., *The art of software testing*, John Willey & Sons, 1979.

[N-S73] Nassi, I., & Schneidermann, B., "Flowchart techniques for structured programming," *SIGPLAN Notices*, **8**(1973), 8, pp. 12-16.

[PAR72] Parnas, D.L., "A technique for software module specification with examples," *CACM*, **15**(1972), 5, pp. 330-336.

[PDV80] "Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage", Teil 1 und 2, Bericht KfK-PDV 186, Kernforschungszentrum Karlsruhe, 1980.

[R-Y78] Ramamoorthy, C.V., & Yeh, R.T., "Tutorial: Software methodology," IEEE Cat. No. EHO 142-0, 1978, pp. 44-164.

[R-S80] Raulefs, P., & Siekmann, J., "Programmverifikation - Darstellung des Forschungsvorhabens," August 1980.

[R-L77] Robinson, L., & Levitt, K.N., "Proof techniques for hierarchically structured programs," *CACM*, **20**(1977), 4, pp. 271-283.

[ROS77] Ross, T.D., "Structured analysis (SA): A language for communicating ideas," *IEEE Trans. on Software Engineering*, **SE-3**(1977), 1.

[R-R76] Roubine, O., & Robinson, L., *SPECIAL Reference Manual*, SRI International, Menlo Park CA, 1976.

[STO80] STONEMAN Document, *Requirements for ADA Programming Support Environments*, U.S. Department of Defense, February 1980.

[STR77] Strunz, H., *Entscheidungstabellentechnik*, Hanser, 1977.

[T-H77] Teichroew, D., & Hershey, E.A., "PSL/PSA: A computer aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. on Software Engineering*, **SE-3**(1977), 1.

[Y-C75] Yourdon, E., & Constantine, L.L., *Structured Design*, Yourdon Inc., New York, 1975, also: Yourdon Press, 1978, and Prentice Hall, 1979.

INTERACTIVE AND INCREMENTAL PROGRAMMING ENVIRONMENTS: EXPERIENCE, FOUNDATIONS AND FUTURE

*Jan Komorowski**
Software Systems Research Center
Linköping University
Linköping
Sweden

ABSTRACT

Which properties of LISP have contributed most to the success of the highly interactive and incremental programming environments that were developed for it? Are there other programming languages for which comparable programming environments can be developed?

In the first part of the talk I shall briefly review the INTERLISP programming system and answer some of these questions. As to the second question, two possible candidates are PASCAL and PROLOG for which successful programming environments were developed in Linköping. The talk will be concluded with a discussion of some possible developments in the area of programming environments.

I am deeply honored by the invitation of the Mathematisch Centrum to give this lecture. I shall try to present the experience we have gained in Linköping in the course of several years of programming in the interactive and incremental milieu of INTERLISP and its offspring [1]. After a sample programming session in INTERLISP, the fundamental notions and basic tools of INTERLISP will be described. Then QLOG - an INTERLISP-based programming environment (PE) for PROLOG - and PATHCAL - an INTERLISP-like PE for PASCAL - will be discussed [2,3]. The paper will be concluded with a glimpse of what future PE's may look like.

1. INTRODUCTION

Today, program development is very often performed on an interactive time-sharing system. A more advanced form of the interactive technique is an incremental one. That is, instead of the conventional (and batch oriented) 'edit-compile-execute' program development cycle, one works in the 'read-evaluate-print' mode. The basic cycle in an incremental system is not the execution of a program but rather that of a statement or declaration. This mechanism, in conjunction with the ever present database of values and declarations, constitutes an incremental system.

An incremental system is the basis of a programming environment. Before proceeding further let us look at a sample session in INTERLISP.

---

* Current address of the author: Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138, USA.

The regular prompt character is an underline. It is preceded by an event number. Input from the user is in bold face. Comments in italics are not part of the log file proper but were added afterwards. The example is that of the well-known factorial.

```
3 _ (DE FACT (N)
      (COND
       ((ONEP N) 1)
       (T (TIMES N (FATC (SUB1 N]

FACT
```

- The *read* step of the 'r-e-p' cycle guarantees a weak form of correctness of the function entered in the sense that only well-formed structures are accepted and stored.
- The effect of the *evaluate* step is in this case an incremental update of a global database of system subroutines and user-defined functions.
- The value of DE(FINE) is the name of the function and is *printed.*

Errare humanum est could be the motto of INTERLISP. It assumes the user is not perfect but will make many, many mistakes. Indeed, the principle of INTERLISP execution is that an error in evaluation does not cause an abort but rather suspends evaluation and allows correction of the error on the spot. Moreover, continuation from the very same point is guaranteed.

Upon encountering an error, control is transferred to an error routine and a breakpoint is created. Usually, the system will try to fix the bug itself and ask the user for a confirmation of the solution presented. If none of the suggested alternatives is accepted or if the error correction is beyond the ken of the system, the initiative is given to the user. The prompt in a breakpoint is a colon preceded, as usual, with an event number. Let us examine an example of computing the value of 5!.

```
4 _ (FACT 5)
u.d.f. ONEP {in FACT} in (ONEP N) (ONEP broken)
```

Predicate ONEP is undefined and the system could not figure out how to correct it. Control is transferred to the user who is informed that an undefined function named ONEP was encountered in the body of FACT in the expression (ONEP N). It is possible to tell the error handler the name of a function to be used instead of the erroneous one:

```
5 : → ZEROP
```

The handler accepts the new function name ZEROP, makes sure the function is defined, calls the editor, and replaces ONEP by ZEROP. FACT is marked as changed and execution continues.

```
FATC {IN FACT} → FACT? Yes
```

This error was easy to correct and after confirmation by the user the same procedure with a call to the editor is performed. The new definition is applied and the result is:

```
120
```

The definition of FACT is now:

```
6 _ PP FACT

(FACT
 (LAMBDA (N)
  (COND ((ZEROP N) 1)
   (T (TIMES N (FACT (SUB1 N)))))))

FACT
```

We can see from this example that INTERLISP is an integrated system, i.e. all tools, not only the interpreter, are resident and may call each other. The error mechanism is also programmable. The user can ask the system to break on a call to a given function.

```
7 _ (BREAK FACT)
FACT
8 _ (FACT 3)

(FACT broken)
9 : ?=
N = 3
10 : GO

 (FACT broken)
11 : ?=
N = 2
12 : GO

   (FACT broken)
13 : ?=
N = 1
14 : GO

     (FACT broken)
15 : ?=
N = 0
16 : BTV
```

The BTV command produces a function call backtrace:

```
N 0

FACT

N 1

FACT

N 2

FACT

N 3

FACT

**TOP**

17 : GO

FACT = 1
FACT = 1
FACT = 2
FACT = 6

6
```

This sample should give the reader some flavor of the INTERLISP programming style although it is by no means a complete presentation. Now let us analyze what properties of the LISP language have been crucial in building a PE of this type.

2. PROPERTIES OF LISP

LISP [4] is an interpreter-based weakly-typed language with a very simple syntax and allowing incremental procedure redefinition. LISP functions and data are syntactically indistinguishable, and are usually represented as lists. Functions are internally stored in canonical form as parse trees. A garbage collector relieves the user of the burden of memory management; it promotes legibility of programs and makes them less prone to errors. LISP is a functional language (I will not discuss this issue with purists!). It does not have declarations and is dynamically typed. Since the interpreter and the runtime system of any LISP system are written in LISP, the user has access to low-level structures and is in complete control.

These are, in my opinion, the most important properties of LISP that have contributed to the advent of sophisticated LISP PE's.

3. PROPERTIES AND TOOLS OF INTERLISP

Let us now return to INTERLISP and have a closer look at it. A PE contains several subsystems such as an editor, a debug system, a prettyprinter, etc. In INTERLISP these are tightly coupled so as to allow invocation of one subsystem from another. Although it is not a single language system, all the subsystems are programmed in LISP and the runtime system, e.g. the stack, is accessible from LISP. We shall first look at the properties of INTERLISP and then at its tools.

3.1. Properties

Incremental execution

Incremental execution gives the user the opportunity to enter statements in the language and have them executed immediately. A secondary consequence of this is that the system is fairly easy to learn. One can try most facilities in the system and the language without having to write a whole program.

Incremental development

In LISP procedures are developed one at a time. It is thus possible to construct and test modules of a program one at a time and later combine them. One may edit and test single procedures of an existing program. It is also possible to test procedures containing calls to procedures that are yet to be defined. The program may be developed 'top down', 'bottom up', or the critical part first, in accordance with the preference of the programmer.

Continuation after errors

When an error occurs during a test run, a program is not aborted, but interrupted while the current environment is retained. After the interrupt, any action, like editing, calling another procedure, etc., is permitted. If a procedure has been edited during an interrupt the new version is used afterwards. It is particularly important to be able to continue after errors. One single error in a program may require a major recomputation if the computation has to be started all over, unless it is possible to repair the error and continue execution.

3.2. Tools

The above-mentioned properties are so deeply rooted in the LISP language that they are shared by virtually all versions of it. On the other hand, the tools offered by the INTERLISP system and its direct successors are unique.

Model of terminal session

The 'r-e-p' loop is in fact supervised by the *programmer's assistant*. It maintains a model of the session and logs interactions and their results on the so called *history list*. There are tools for redoing and undoing the effects of previous interactions, and searching for and editing previous interactions on the history list.

File Librarian

A resident system must maintain consistency between workspace structures and their copies on file. The file librarian in INTERLISP knows about all program structures in the workspace that are to be saved; it knows which parts of which files must be updated by a save operation; and it will ask the user where to keep newly defined items.

Editor

The INTERLISP editor provides means to replace (old) subexpressions (by new ones), to insert or delete them, to add or remove parentheses around expressions, to search for subexpressions, and several other services. Since the editor manipulates structures, only well-formed formulae can result from edit manipulations. The editor is tightly coupled to the rest of the system and can be called from virtually anywhere. It also maintains its own history list in the same way as the top-level. Thus, the user can freely edit her/his material without any fear of losing it should s/he wish to undo a change. Modified structures are marked so as to enable the file librarian to do its job.

Masterscope

Masterscope is a program analyzer which inspects user's programs and stores the results in a database. The user can consult this database by asking questions such as: 'Which functions call FOO?', 'Which functions have free variable X?', 'Edit every function that calls FOO', etc. It knows about changes made by the user and reanalyzes the parts that were changed.

DWIM

DWIM means 'Do What I Mean'. It is used for correcting simple misspellings in case of errors. If a correction is accepted by the user, DWIM calls the editor and makes the change. The editor in its turn informs the file librarian. Also, some parentheses misplacements can be corrected. Of course, the modifications introduced by DWIM are put on the history list and can be undone later, if required.

Compilation

The efficiency of the code produced by the compiler depends on the type of compilation chosen. The standard compilation compiles functions separately and improves the speed from 2 to 5 times. A more sophisticated compilation - called block compilation - requires declarations defining a self-contained part of a program. The speed improvement is up to 30 times the speed of interpreted code. Interpreted, compiled and block compiled code can be mixed freely.

Let me interrupt the description at this point. What has been shown so far should give some understanding and flavor of INTERLISP tools. INTERLISP is without doubt the most flamboyant programming environment ever developed. The amenity of this PE has been criticized - in part quite justly. Of course, INTERLISP causes 'l'embarras de richesse.' It resembles an extremely well equipped but somewhat crowded workshop - if carefully used it provides an exceptional environment for software experiments; if misused, it leads to badly structured and unreliable programs.

INTERLISP and other advanced LISP PE's are still the best ones in existence. For example, I needed less than three months to develop a PROLOG environment in INTERLISP. Similar work in other languages and environments took several years for teams of programmers. A natural question is then whether a PE similar to INTERLISP could be developed for other languages. In particular we, in Linköping, were interested in languages which enforce a stricter programming discipline. We chose PASCAL and PROLOG for our experiments.

4. THE PATHCAL PROJECT

'The goal of the PATHCAL project was to develop - within the framework of a particular language, i.e. PASCAL - an interactive and incremental PE. The project was intended to produce experience and ideas on how to construct a PE for the ALGOL family of languages. The implementation language of PATHCAL is INTERLISP. Most of the PATHCAL tools were borrowed from INTERLISP and it is a single language system. For example, the commands for searching the PATHCAL history list are PASCAL statements.

In PATHCAL there are, among other things, a structure editor, a debug system, and a prettyprinter. All the subsystems are tightly coupled so as to allow invocation of one subsystem from another. All systems are PASCAL procedures, although some may have privileges not allowed in standard PASCAL. PATHCAL provides incremental execution, continuation after errors and structure editing. It maintains a history list and supports incremental program development' [3].

I have no time to describe the system here. Those interested should read Wilander's paper [3]. Let me only point out that experience with PATHCAL has shown that building an incremental programming system for PASCAL is quite feasible. Most ideas incorporated in the system are adaptations of the corresponding INTERLISP facilities. The idea of an all-inclusive PASCAL system block, where test data and declarations are kept, was a very successful technique. It offered convenient testing of single procedures and a good environment for extensions of the system, both by the user and the system implementor. The PATHCAL system forms the basis for a new project in which an incremental compiler for PASCAL is being developed [5].

5. THE QLOG PROJECT

The efficiency of the INTERLISP PE suggested the QLOG project [6,7]. I wanted to obtain a very high quality PROLOG PE at minimum cost. To achieve this, PROLOG procedures were made first-class LISP citizens, i.e. they were made LISP procedures. This technique is called *functional embedding*. It was expected that by embedding PROLOG in LISP one could transfer most of the LISP PE to PROLOG.

The goal of the QLOG project was to investigate to what extent such an embedding would be possible, how LISP tools would fit the requirements of PROLOG, and what new facilities would have to be provided. A secondary goal was to develop a transportable version of the interpreter which would permit tailoring of the system to a particular LISP environment without any major redesign of the system's kernel. The efficiency of the implementation in terms of computing speed was given minor attention. On the other hand, I also felt that it would be desirable to have a set of PROLOG primitives within LISP. Thus the embedding of PROLOG in LISP was to benefit both languages. LISP would contribute with its very sophisticated and advanced PE, PROLOG with pattern-directed invocation, with an associative data base having richer structure than the simple property lists of LISP, with a specification oriented programming methodology, etc.

5.1. PROLOG in the LISP environment

In implementing one language in another there is a range of possible choices, from an embedded sublanguage to a freestanding one. As it is characterized in [8] 'the conventional implementation of a sublanguage in a host language is freestanding: the

new language processor is written in the host language, but is regarded as completely separate. The control structures of the new language are implemented with the help of those of the host language but are distinct from them. Utilities, such as formatters or flow analyzers and many more, must in general be written anew, since they are designed to work on the syntactic and semantic structures of the *host* language, not those of the new language.'

According to the strategy of 'maximal embedding' one should borrow as much as possible from LISP and never build something new without good reason. I decided on this approach since it gives many major components of the LISP environment almost for free. For example, the procedure call mechanism is borrowed from LISP by transforming QLOG procedures into LISP procedures. In this way we obtain utilities like *break* at minimal cost. I chose INTERLISP as my host LISP system. QLOG, however, is portable across LISP dialects; it inherits the tools available in whatever LISP PE it is embedded in.

The utmost in embedding is obtained by implementing the new capability as a set of subroutines in the host language. This maximizes the number of inherited features but permits no departures from the semantics of the host language. If the subroutines conceal such departures internally, for instance by having backtrack stacks or by treating certain data structures as essentially new data types, then to that extent I would characterize the design as freestanding.

5.2. A sample QLOG session

Variables in QLOG are prefixed with a colon. The heads of clauses are enclosed in brackets and all clauses with the same predicate name are grouped and written within brackets too. All the rest is the same as in INTERLISP.

```
@QLOG
QLOG LINKOPING 22-May-82

Hi, Jan.

5 _ (LOAD 'FAMI)
FILE CREATED 27-Mar-82 11:00:05
FAMICOMS
<JAN>FAMI..2
```

This is just the INTERLISP load.

```
6 _ PP GRANDPARENT
[
([GRANDPARENT :X :Z]
   (GRANDMOTHER :X :Z))

([GRANDPARENT :X :Z]
   (GRANDFATHER :X :Z))
]
GRANDPARENT
```

PP works fine as it has been extended to understand the syntax of QLOG.

```
7 _ PP GRANDFATHER
[
([GRANDFATHER :X :Z]
   (PARENT :Y :Z)
   (FATHER :X :Y))
]
GRANDFATHER

8 _ PP FATHER
[
([FATHER GEORGE HENRY])

([FATHER HENRY JAN])

([FATHER HENRY STAN])

([FATHER VICTOR VIOLETTE])

([FATHER GEORGE STANISLAS])

([FATHER STANISLAS JERZY])
]
FATHER

9 _ (GRANDPARENT :WHO JAN)
```

A call to a QLOG procedure is typed in as if it were a LISP call. The programmer's assistant puts the QLOG interaction on the history list and changes from the external to the internal syntax.

```
WHO <= ANNA
More? Yes
WHO <= HENRIETTE
More? Yes
WHO <= VICTOR
More? Yes
WHO <= GEORGE
More? Yes
FAILURE
```

FAILURE means there are no further solutions.

```
10 _ (GRANDPARENT ANNA :WHO)
```

Note that as usual in PROLOG a parameter of a procedure can be input or output.

```
WHO <= JAN
More? Yes
WHO <= STAN
More? Yes
FAILURE
```

```
11 _ (EDITQ GRANDFATHER)
```

I am going to edit this procedure and deliberately introduce a misspelling:

```
edit
1*PP
(((GRANDFATHER :X :Z)
  (PARENT :Y :Z)
  (FATHER :X :Y)))
1*(R FATHER FATHR)          R = Replace
2*OK
GRANDFATHER
12 _ PP GRANDFATHER
[
([GRANDFATHER :X :Z]
   (PARENT :Y :Z)
   (FATHR :X :Y)))
]
GRANDFATHER
```

```
13 _ REDO GRANDPARENT
```

The programmer's assistant searches the history list for the most recent use of GRANDPARENT and reexecutes it:

```
WHO <= JAN
More? Yes
WHO <= STAN
More? Yes
u.d.f. FATHR {in EVAL} in (FATHR (VAR X) (VAR--))
(FATHR broken)
```

QLOG procedure FATHR is of course undefined. Note that everything happens just as in the case of an undefined LISP procedure. Virtually nothing has changed in the error mechanism and this is obtained for free!

```
14 :→ FATHER
FAILURE
15 _ PP GRANDFATHER
```

As in the case of INTERLISP procedures the correction is automatically made in the definition:

```
[
([GRANDFATHER :X :Z]
   (PARENT :Y :Z)
   (FATHER :X :Y))
]
GRANDFATHER
```

5.3. The implementation strategy

For the designer the following two properties of PROLOG are rather important: PROLOG has strong equality between data and procedures, and PROLOG may be given a procedural interpretation. (The equality of data and procedures has similar consequences for PROLOG as it has for LISP, e.g. a structure editor, READ and PRINT, etc.) As an implementation principle I applied the 'law of maximal embedding' i.e. I implemented special facilities for QLOG only when necessary and borrowed as much from LISP as I could. This minimized the cost of implementation, while simultaneously maximizing the number of inherited language features (like READ and the garbage collector) and system facilities (like the list structure editor and file librarian). The inherited services are far larger than QLOG itself (by an order of magnitude), far better than anything I could afford to build alone from scratch, and far better than the facilities built from scratch in other PROLOG implementations.

Data types were very easy. No special finesse was required to implement PROLOG terms in terms of LISP S-expressions. In this way - virtually for free - QLOG obtained allocators, a garbage collector, READ and PRINT functions, macro facilities, a list structure editor, etc.

The control structure and variable binding mechanism of PROLOG were harder to implement and called for freestanding implementations since they are quite different from those of LISP. The backtrack control structure of PROLOG usually requires retention of procedure activation frames, even after a head has successfully matched the current goal and the corresponding body has been executed. This required a special control stack for QLOG procedures with pointers to the evaluation environments of their arguments. Similarly, PROLOG's variable binding mechanism required a separate stack structure. Also, the sequencing of nested calls is different from that of LISP and required special care.

The procedure call mechanism was borrowed from LISP. Usually, constructs of a new language are not given functional or procedural meaning in a host language. They are rather input parameters to an interpreter built on top of the host language. By functionally embedding them, QLOG procedures become LISP procedures, i.e. they are not simply data objects without meaning in LISP terms. For example, the PROLOG procedure

```
appnd(nil, X, X).

appnd([Kar | X], Y, [Kar | Z]) :-
    appnd(X, Y, Z).
```

is represented by the following QLOG procedure

```
(APPND
 (NLAMBDA L
  (+GOAL+ L '( ((APPND NIL :X :X))
               ((APPND (:KAR.:X) :Y (:KAR.:Z))
                   (APPND :X :Y :Z))) )))
```

INTERLISP procedure APPND is an FEXPR (NLAMBDA) with an arbitrary number of arguments (L). At the moment APPND is called, L is bound to a list of the actual parameters and passed to +GOAL+, i.e. to the QLOG interpreter.

This encapsulation of calls to the QLOG interpreter provides the functional embedding, since every QLOG procedure is first seen by the regular LISP interpreter. Moreover, such a procedure is also considered to be an ordinary object by other bookkeeping utilities. Thus, access to the entire set of function-oriented facilities like break, the file librarian, the history package, etc., is retained.

The INTERLISP top loop accepts QLOG forms (since they are just LISP forms), evaluates them and updates the history list. Only a small extra routine is necessary to save multiple results on the list (instead of the the standard single LISP result).

Breakpoints and procedure traces may be placed on any QLOG procedure using the standard LISP facilities. Breakpoints work perfectly as they are. Tracing works, but prints the LISP arguments and bindings instead of the QLOG items. Another small interface function is necessary to make trace print the right things.

The INTERLISP spelling corrector works fine. A call to an undefined QLOG procedure will be trapped and will either be corrected automatically or start a dialogue with the user.

The regular INTERLISP editor is called on QLOG procedures through a special EDITQ function (of 5 lines) which fetches the definition of a QLOG procedure and notifies the file librarian if necessary.

The INTERLISP file librarian knows about all structures in the workspace that have to be saved in case of changes. It knows which parts of which files must be updated, and it will ask the user where to keep newly defined things. All these facilities work without modification.

Lack of space prevents me from extending this list and I shall limit myself here only to my 'worst cases.' Because the INTERLISP prettyprinter does not understand the QLOG procedure structure, a dedicated prettyprinter of 36 lines was written. Furthermore, although LISP backtraces (i.e. the functions that display the current contents of the stack with or without variables) work, they are not very useful since they display the interpreter functions and LISP variables rather than the QLOG items. There was a clear need for specialized backtraces which would display the AND/OR tree structure of the PROLOG computation. Also, there are new functions for introducing new QLOG definitions; these notify the file librarian, too.

One might ask, why not use the embedded approach also for the variable binding mechanism? In principle this would be possible, using the INTERLISP spaghetti stack. It might seem that in that case one would inherit INTERLISP's commands for stack display too. I admit I was tempted to try this solution and I implemented it. But portability was sacrificed, the implementation became clumsy, and context switching costs became unreasonable.

New Facilities

The resulting PE provided a unique set of tools for PROLOG program development. Since PROLOG computation has a different structure than LISP computation, there was a clear need for language specific tools, for example commands for displaying the AND/OR program and computation trees. Another useful facility is the *spy* package [9].

Extensions and improvements.

Instead of a special representation for lists, QLOG uses the regular LISP dot notation. In this notation functional symbols may have a variable arity.

There is a provision for writing QLOG procedures with variable predicate names. At the moment of the call, however, these must be instantiated to a predicate name, otherwise an error occurs.

The semantics of QLOG is better defined than the semantics of other PROLOGs, because it is not total. A call to an undefined procedure does not result in a pattern match failure and thus in an uncontrollable continuation of the program, but causes a break and leaves the decision to the user. For DECsystem-10 PROLOG it is promised that there are no 'incomprehensible error messages' [10]. Indeed, there are no error messages at all. Execution does not even stop if an undefined (very likely misspelled) procedure is called. I can hardly agree, however, that 'this totally defined semantics ensures that programming errors do not result in bizarre program behavior.' Best of all, QLOG error handling can be programmed to behave like the error handling of other implementations, provided the user works in the so-called 'closed world.'

5.4. Conclusions from the QLOG project

LISP allowed me to implement PROLOG procedures as LISP procedures, while still retaining my own variable scoping and control mechanisms. Very likely no other language could do the same. This maximized the number of inherited features.

QLOG inherited most of the major components of the LISP PE with little or no additional code, and in this cheap way became a high quality PE. Interfacing the existing INTERLISP facilities to the new language required 30 to 50 times less code than the LISP facilities require themselves. These facilities are rather unique and not found in any other PROLOG system. Also, some new tools were developed specially for PROLOG. Not many of them are found in other PROLOG programming system.

LISP itself has been complemented with pattern directed invocation of functions, an associative data base with a richer structure than property lists, etc. All this took less than 30 pages of prettyprinted LISP code. The user of QLOG can use LISP without ever noticing that PROLOG is around, or he can choose to run only PROLOG with all the advantages of a modern PE, or he can freely mix PROLOG and LISP. The first option is possible because the LISP interpreter has not been changed and thus the only cost of having QLOG in the workspace is that of some extra storage, not of execution speed. The second option uses LISP as an implementation language (like assembler or FORTRAN in other implementations) and never steps down to it. However, should the need arise (e.g. when interfacing an existing built-in LISP function to QLOG, making experimental changes in the QLOG interpreter, or implementing embedded languages in PROLOG), there are no obstacles to do so. The third option is provided for those users who are aware of the difference between the languages. However, in contrast to LOGLISP [11], I provided an explicit interface. The LISP function must be called from a

PROLOG program to initiate a LISP computation, and vice versa the QLOG function must be called from a LISP program. It was believed that extra care had to be taken if LISP and PROLOG codes were to be mixed.

6. A GLIMPSE OF THE FUTURE

The experience gained with the PATHCAL and QLOG projects shows that it is possible to build interactive incremental PE's for a variety of languages. However, PE research has by no means come to an end. First of all, there is a need for formal definitions of tools used in PE's. Recently, some work was done in this field [12]. I myself have done a few experiments in defining some of the QLOG tools like *break*, *trace* and *spy*. These definitions are based on the abstract PROLOG machine described in [13,14,15]. Eventually, we would like to be able to develop a mathematical model of *incremental* programming.

A second and more important issue is what type of higher level support a PE should provide. Intuitively, one would like the PE to be able to reason about programs and tell the user about their properties. Such reasoning could be either automatic or semi-automatic. I believe that at this point one has to decide whether the tools are to be constructed for a user who follows the specification school, or for one who does not.

Provided with a specification the programmer following the specification school approach is interested in transforming it into a runnable or a more efficient version. This is of course the case of compilation and we are confronted with the problem of the correctness of the compiler. Since I am a 'PROLOGer' (and to some extent a 'LISPer') I prefer to understand any program transformation as a step in a formal proof. (Consider a program transformation system written in PROLOG; then the result of the transformation is also the result of a proof.)

Although there are some automatic theorem provers, even for small programs the complexity of proofs is so great that neither a computer nor a human being can perform them alone. For a computer the proof may be too difficult, for a human being too laborious in its details. A semi-automatic tool offered as part of a PE seems to be the right solution. The purpose of such a PE would be to relieve the user of the great burden of small steps in long proofs, and to enable him to outline the proof in an interactive way. The user should be able to follow the proof, to inspect it and to obtain a good understanding of what was built up.

I know of at least one system where a lot of attention was paid to the design of the interface between the user and the theorem prover. This system is called NATDED and was developed in Uppsala [16]. It uses natural deduction to synthesize PROLOG programs from an axiomatic specification of data structures and executable objects written in first order predicate logic. The system is written in PROLOG. I think the results of this kind of research merit close attention.

On the other hand, the approach of the specification school does not always apply. For instance, cases where the very first specification is runnable are not that unusual today. In fact a PROLOG program is often a direct formalization of a programmer's intuitions, or a formalization of an imprecise natural language specification. In such cases it is impossible to talk about the classical correctness (total or partial) of a program as there is no formal point of reference. Operations in database systems constitute another example.

Although we cannot consider the correctness of such programs/operations, there are other important checks to be performed in these cases for which one would like to have support.

An (idealized) PROLOG program has a very interesting property: it does not contain semantic mistakes (nor are there any syntactic ones, provided a syntax driven editor and/or input is used). This property is due to the fact that a program written in Horn clauses defines a theory. There are other problems, however. For instance, the program may not terminate; or its semantics may not be the one intended by the programmer; or the programmer may have made some parts of the program redundant; or by removing a clause some other clauses depending on it may have become obsolete, etc. We do not yet have suitable tools for helping us in performing the update operations.

Indeed, a shortcoming of today's logic programming (although it has a much more advanced character than other types of programming) is that it deals with a single theory, while many applications must handle deductions from several alternative theories. Thus in maintaining a relational database (i.e. a collection of Horn clauses), tuples are added, deleted or modified. Addition of a tuple corresponds to assertion of a simple unit clause. Although in this case only a single theory is involved, it changes over time. Moreover, the problem of maintaining integrity constraints amounts to testing the consistency of the proposed addition with the theory embodied by the database.

Several attempts are being made to solve these problems. Some of them are based on non-monotonic logic and dependency networks, but these have never been used in the context of logic programming. Nevertheless, they will probably bring us closer to a solution of the maintenance problem [17]. Other attempts try to exploit the possibility of writing PROLOG programs about PROLOG programs, thus using a metalanguage approach similar to Weyhrauch's [18].

7. CONCLUSIONS

I have briefly discussed programming environments for three different languages. We have learned how to build an interactive incremental programming milieu of the INTERLISP type for a broad spectrum of languages. Nevertheless, many questions remain to be answered in the areas of program transformations, maintenance of databases of programs, program testing, universal syntax directed editors, etc. Research in programming environments is moving towards the development of tools that use the semantics and not just the syntax of programs. A lot of information about the meaning of programs will be extracted prior to execution, for example by reasoning about them. To this end one would like to have better methods for expressing the semantics of programming languages.

REFERENCES

[1] Teitelman, W., *INTERLISP Reference Manual*, XEROX Co., Palo Alto Research Center, 1978.

[2] Pereira, L., et al. *User's Guide to DECsystem-10 PROLOG*, Lisbon, September 1978.

[3] Wilander, J., "An interactive programming system for PASCAL," *BIT*, **20**(1980), pp. 163-174.

[4] McCarthy, J., et al., *LISP 1.5 Programmer's Manual*, The MIT Press, 1966.

[5] Fritzson, P., "Finegrained incremental compilation for PASCAL-like languages," Report LiTH-MAT-R-82-15, Software Systems Research Center, Linköping Institute of Technology, 1982.

[6] Komorowski, H.J., "QLOG interactive environment - the experience from embedding a generalized PROLOG in INTERLISP," Report LiTH-MAT-R-79-19, Software Systems Research Center, Linköping Institute of Technology, 1979.

[7] Komorowski, H.J., "QLOG - the programming environment for PROLOG in LISP," **in** Clark, K., & Tärnlund, S.-A., (eds.), *Logic Programming*, Academic Press, 1982.

[8] Komorowski, H.J., & Goodwin, J., "Embedding PROLOG in LISP: an example of LISP craft technique," Report LiTH-MAT-R-1981-2, Software Systems Research Center, Linköping Institute of Technology, 1981.

[9] Byrd, L., "PROLOG debugging facilities," Department of Artificial Intelligence, University of Edinburgh, 1980.

[10] Coelho, H., et al. "How to solve it with PROLOG," Laboratorio Nacional de Engenharia Civil, 1979.

[11] Robinson, J.A., & Sibert, E.E., "LOGLISP: motivation, design and implementation," **in** Clark, K., & Tärnlund, S.-A., (eds.), *Logic Programming*, Academic Press, 1982.

[12] Giacalone, A., et al., "Toward a formally based programming environment," *Proceedings of the ECIICS*, North-Holland, 1982.

[13] Komorowski, H.J., *A Specification of an Abstract PROLOG Machine and its Application to Partial Evaluation,* Ph.D. Thesis, No. 69, University of Linköping, 1981.

[14] Komorowski, H.J., "Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of PROLOG," *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1982.

[15] Komorowski, H.J., "An abstract PROLOG machine," *Proceedings of the ECIICS*, North-Holland, 1982.

[16] Eriksson, A,. & Johansson, A.L., "Computer-based synthesis of logic programs," *Proceedings of the 6th ICI*, Torino, 1982.

[17] Bowen, K., & Kowalski, R., "Amalgamating language and metalanguage in logic programming," Report 4/81, School of Computer and Information Science, Syracuse University.

[18] Weyhrauch, R., "Prolegomena to a theory of mechanized formal reasoning," *Artificial Intelligence*, **13**(1980), pp. 133-170.

ONTWERP VAN EEN PROGRAMMEEROMGEVING VOOR EEN PERSONAL COMPUTER

Leo Geurts
Mathematisch Centrum
Amsterdam

SAMENVATTING

B is een programmeertaal voor personal computing. De belangrijkste doelstelling bij het ontwerp van B is de eenvoud voor de gebruiker geweest. Dezelfde doelstelling staat voorop bij het ontwerp van een programmeeromgeving voor B. De eenvoud is ermee gediend als de omgeving niet uit onafhankelijke functionele elementen bestaat, maar één geïntegreerd systeem is dat voor verschillende functies dezelfde communicatiemethode met de gebruiker hanteert.

1. INLEIDING

In 1975 werd een eerste ontwerp geformuleerd van een programmeertaal voor "beginners" [1]. Dit ontwerp, B0, heeft een aantal evolutiestappen doorlopen, waarvan de laatste in 1981 werd geïmplementeerd en als de programmeertaal B (voorlopige naam) werd gepubliceerd [2, 3]. Al in de B0-publicatie werd van de uiteindelijke taal B geëist dat deze "geïntegreerd moet kunnen worden in een [conversationeel] systeem" en werd het voornemen uitgesproken "uiteindelijk niet alleen de B-taal, maar het complete B-systeem te definiëren", dat "onmiddellijk reageert wanneer dat toepasselijk is, in plaats van zijn reactie op te schorten tot het moment van de uiteindelijke analyse als de gebruiker is uitgepraat", dat "één gezicht vertoont aan de gebruiker, in plaats van een verscheidenheid aan gezichten op verschillende niveaus, zoals een editor, een file-systeem, een compiler, elk met zijn eigen conventies en reacties en nauwelijks op de hoogte van elkaars bestaan", en dat "de gebruiker niet in onzekerheid laat over wie er aan de beurt is, maar zonodig om een reactie vraagt".

Het ontwerp van zo'n programmeeromgeving is het onderwerp van deze colloquiumbijdrage.

2. UITGANGSPUNTEN

De B-programmeeromgeving dient te voldoen aan dezelfde criteria die voor de B-taal golden:

1. Doelgroep: de gebruiker is een beginnend, althans niet professioneel programmeur, die wel meer wil dan kant-en-klare programma's gebruiken, maar niet zover gaat dat hij in teamverband software ontwikkelt. Toch zou een breed spectrum van gebruikers zich aangetrokken moeten voelen. Dat valt ook wel te verwachten, omdat het verschil tussen gebruikersgroepen slechts gradueel is. Daarom ook moet de gebruiker niet kinderachtig behandeld worden; het is immers te verwachten dat

velen al spoedig heel handig met taal en systeem zullen leren omgaan en van bijvoorbeeld overdreven uitgebreide foutmeldingen meer last dan gemak zullen hebben. Anderzijds steekt in de geroutineerde professionele programmeur nog wel zoveel van een beginner dat ook hij gebaat is bij vele van de eigenschappen van taal en systeem.

2. Eenvoud: de constructies die de gebruiker moet leren zijn gering in aantal, ze sluiten aan bij het conceptuele niveau waarop de gebruiker over zijn probleem denkt, en vormen tezamen een krachtig geheel. Er zijn geen duistere hoeken waarin zich weinig gebruikte of slecht begrepen constructies schuilhouden.
3. Interactie: de gebruiker wordt interactief (conversationeel) te woord gestaan; op zijn commando's wordt gereageerd zodra ze gegeven zijn.
4. Eenheid: taal en omgeving vormen een eenheid.
5. Service: de gebruiker wordt veel service geboden; hij wordt vriendelijk en begrijpend tegemoet getreden; hij wordt niet voor vervelende verrassingen geplaatst; hij hoeft zich niet te bekommeren om details die hem uit handen genomen kunnen worden; dit alles ook als het lastig implementatiewerk of verlies van efficiëntie tot gevolg heeft: meer en meer zal programmeertijd de bottleneck van een project vormen, niet machinetijd of geheugengebruik.

Dat de geboden service ook iets mag kosten wordt geïllustreerd door de volgende facetten van de *taal* B:

-- De aritmetiek voor rationale getallen is exact.
-- Het type-systeem is orthogonaal.
-- Het type-systeem is zo ontworpen, dat type checking kan gebeuren op grond van informatie ontleend aan de context, zodat declaraties of specificaties overbodig zijn.
-- Het typesysteem maakt ook incrementele type checking mogelijk, zodat tegen fouten kan worden gewaarschuwd zodra ze gemaakt worden [4].
-- B bevat "dure" datastructuren: lists (verzamelingen die, ook na uitbreiding, gesorteerd blijven bewaard) en tables (arrays die geïndiceerd mogen worden met elke waarde van een zeker type; worden ook gesorteerd gehouden). (Bij nadere bestudering blijken deze structuren op efficiënte wijze geïmplementeerd te kunnen worden [5].)
-- B kent enkele "dure" operaties, zoals het doorlopen van alle verdelingen van een tekst in een gewenst aantal delen.
-- Er zijn geen grenzen gesteld aan de lengte van namen, aan de grootte van getallen, aan de diepte van nestingen, e.d.

Voorbeelden van gemak voor de B-programmeur waarvoor niet zoveel betaald behoeft te worden zijn:

-- Een aanroep van een door de gebruiker of door een derde geschreven procedure heeft dezelfde verschijningsvorm als de tot de taal behorende commando's en functies.
-- I.p.v. `BEGIN ... END`, `DO ... OD` e.d. wordt indentatie gebruikt voor het groeperen van commando's.
-- Een van de controlestructuren van B is de refinement, een lokale parameterloze vorm van procedure, die hiërarchisch programmeren tot op een laag niveau gemakkelijk maakt.
-- B kent slechts twee basis-types en drie manieren om nieuwe types op te bouwen. Ze zijn zo gekozen dat wie andere datastructuren wenst, die gemakkelijk met die van B kan simuleren.

-- Evaluatie van een expressie heeft geen zijeffecten.
-- Alle grootheden die de gebruiker in zijn programma's manipuleert zijn "concreet": ze hebben een standaardrepresentatie die de gebruiker kent en die zichtbaar gemaakt wordt door het commando `WRITE ...` ; zulke "magische" grootheden als pointers komen dus niet voor.
-- Net als in het dagelijks leven zijn meervoudige vergelijkingen als `a<b<=c` toegestaan.

Aan enkele eigenschappen van de taal is te zien dat vooral eenvoudig gebruik beoogd wordt, in tegenstelling tot bijvoorbeeld systeemprogrammering of ontwikkeling van grote programmapakketten:
-- Procedures kunnen geen lokale procedures bevatten. (Recursie is wel mogelijk).
-- Modules vergelijkbaar met de class van Simula kunnen niet gemaakt worden.
-- Unions kunnen niet gedefinieerd worden.

Toch blijkt het geboden gemak ook voor professionele programmeurs aantrekkelijk.

3. DE TAAL B IN VOGELVLUCHT

Om de geest van de taal B, waarmee de te ontwerpen programmeeromgeving op goede voet moet komen te staan, duidelijker te maken, laten we hier de meeste constructies ervan de revue passeren.

3.1. Types en operaties

De twee basis-types van B zijn:
-- getallen; rationale getallen worden precies gerepresenteerd, hoe groot teller en noemer ook worden. Naast de gebruikelijke operaties zijn er nog `*/` en `/*` voor resp. teller en noemer van een exact getal: `*/1.25 = 5`, omdat `1.25 = 5/4`.
-- teksten, met o.a. concatenatie: `'rad'^'ar' = 'radar'`, en twee soorten afkappen: `'radar'|2 = 'ra'` en `'radar'@3 = 'dar'`, eventueel gecombineerd:

```
'radar'@2|3 = 'ada'.
```

Er zijn drie manieren om nieuwe types samen te stellen:
-- compound, een record met een vast aantal velden, zonder veldnamen:

```
PUT 'origin', 0, 0 IN p
PUT a, b, c IN b, c, a
```

Veranderen van een veld:

```
PUT p IN t, x, y
PUT 'oorsprong', x, y IN p
```

-- list, een verzameling van elementen van gelijk type, die elk meer dan eens kunnen voorkomen. Ze worden gesorteerd bijgehouden, hetgeen mogelijk is doordat voor elk type een ordening gedefinieerd is:

```
WRITE {('a', 1, 0); ('b', 0, 0); ('a', 0, 2)}
```

geeft:

```
{('a', 0, 2); ('a', 1, 0); ('b', 0, 0)}
```

`{2..10}` en `{'a'..'i'}` zijn ook lists, elk met 9 elementen.

Operaties:

```
INSERT el IN list
REMOVE el FROM list
```

-- table, een array waarin de keys (indices) waarden van gelijk type zijn (ze hoeven niet aaneensluitend te zijn) en de opgeslagen waarden ook van gelijk type zijn. Ze worden op key gesorteerd gehouden, zodat na

```
PUT {['Jan']: 223687; ['Aad']: 776312} IN tel
```

het commando `WRITE tel` geeft:

```
{['Aad']: 776312; ['Jan']: 223687}
```

Operaties:

```
keys tel = {'Aad'; 'Jan'}
PUT 222222 IN tel['Alarm']
```

Voor alle types bestaan voorts de volgende operaties:

`PUT expr IN target`	assignment
`WRITE expr`	schrijf op scherm
`DELETE target`	ruim op
`READ target EG expr`	input (verwacht hetzelfde type als `expr`)

Bovendien zijn de volgende operaties gedefinieerd voor elk van de drie dynamische "seriële" datastructuren: text, list en table (van tables niet de keys maar de opgeslagen waarden).

aantal elementen	`#'radar' = 5`
aantal voorkomens	`3#{1; 3; 3; 4} = 2`
kleinste element	`min {[1]: 'z'; [2]: 'a'} = 'a'`
grootste element	`max 'radar' = 'r'`
selectie	`4 th'of {1; 2; 3; 5; 8} = 5`

3.2. Tests

Tests worden opgebouwd met <, <=, =, >=, > en <> (ongelijk). Op de seriële structuren kan de test `in` gebruikt worden:

```
IF expr in series: ...
```

Tests kunnen gecombineerd worden met `AND`, `OR` en `NOT`. Voorts kunnen de kwantoren `SOME`, `EACH` en `NO` gebruikt worden:

```
IF SOME d IN {2..n-1} HAS n mod d = 0:
    WRITE '`d` divides `n`'
```

Tenslotte kunnen met `PARSING` in plaats van `IN` alle mogelijke verdelingen van een tekst in een gewenst aantal delen worden doorlopen:

```
WHILE SOME p, q, r PARSING t HAS q=',': PUT p^r IN t
```

verwijdert alle komma's uit `t`.

3.3. Controle-structuren

Voor selectie zijn er twee mogelijkheden: IF als er precies één alternatief is, SELECT voor één of meer alternatieven:

```
IF p<0: PUT -p IN p
SELECT:
    a<0: INSERT a IN neg
    a>0: INSERT a IN pos
    ELSE: PUT nzero+1 IN nzero
```

Bij SELECT moet de test van minstens één van de alternatieven slagen. De eerste van die alternatieven wordt uitgevoerd.

Voor loops zijn er ook twee structuren:

```
WHILE a>1: PUT a/2 IN a
```

en voor het doorlopen van de seriële datastructuren:

```
FOR i IN {1..10}: PUT s+i IN s
FOR char IN 'abracadabra': ...
FOR nr IN tel: ...
```

B kent drie soorten units (procedures):

-- De HOW'TO-unit, het door de gebruiker gedefinieerde commando:

```
HOW'TO PREFIX head TO tail:
    PUT head^tail IN tail
```

Als nu `t = 'dar'`, dan heeft `PREFIX 'ra' TO t` tot gevolg dat `t = 'radar'`.

-- De YIELD, de door de gebruiker gedefinieerde functie:

```
YIELD inv word:
    PUT '' IN i
    FOR c IN word: PREFIX c TO i
    RETURN i
```

Na deze functiedefinitie geldt bijvoorbeeld `inv 'dar' = 'rad'`.

-- Tenslotte de door de gebruiker gedefinieerde test:

```
TEST palindromic word:
    REPORT word = inv word
```

Nu slaagt bijvoorbeeld de test `palindromic 'radar'`.

De parameteroverdracht is *by name* voor HOW'TOs, en *by value* voor YIELDs en TESTs. Een variabele in een unit is lokaal, tenzij het een parameter van de unit is. Als de gebruiker toch globale variabelen in een unit wil introduceren zonder ze als parameters mee te geven, kan hij deze in de eerste regel van de unit noemen na het keyword SHARE:

```
TEST ok text:
    SHARE alphabet
    REPORT EACH c IN text HAS c in alphabet
```

Deze test gaat na of de parameter `text` geheel bestaat uit karakters die in de globale variabele `alphabet` voorkomen. Deze unit toont nog een interessant aspect van de taal: de test werkt zowel wanneer `alphabet` een text is, als wanneer alphabet een

list van texts van lengte 1 is of een table met zulke texts als opgeslagen waarden. Sterker nog: `text` hoeft geen text te zijn. Het is goed zolang `text` en `alphabet` seriële datastructuren zijn met elementen van gelijk type. Dit is te danken aan het feit dat in de unit alleen commando's en functies nodig zijn die op alle drie de seriële datastructuren gedefinieerd zijn.

Met behulp van units kunnen vele datastructuren die in B niet voorkomen gemakkelijk worden nagebootst. Zo simuleert het volgende "pakket" stacks in tables:

```
HOW'TO EMPTY s: PUT {} IN s
HOW'TO PUSH v ON s: PUT v IN s[#s+1]
YIELD top s: RETURN s[#s]
HOW'TO POP s: DELETE s[#s]
TEST empty s: REPORT s={}
```

Naast de drie soorten units kent B ook drie soorten refinements: parameterloze units, lokaal ten opzichte van een unit. In de volgende unit, die van een zin met eventuele leestekens en hoofdletters bepaalt of het een palindroom is, wordt elk van de drie soorten refinements gebruikt.

```
TEST pal sent:
    STRIP
    REPORT sent = inv sent
STRIP:
    PUT sent, '' IN s, sent
    FOR c IN s: PUT sent^repr IN sent
repr:
    SELECT:
        lower'case: RETURN c
        upper'case: RETURN rank th'of {'a'..'z'}
        ELSE: RETURN ''
lower'case: REPORT c in {'a'..'z'}
upper'case: REPORT c in {'A'..'Z'}
rank: RETURN #{'A'..c}
```

Het commando `QUIT` zorgt ervoor dat de refinement of de unit waarin hij voorkomt (voortijdig) wordt verlaten, zoals in deze unit die alle waarden in een serie afdrukt die kleiner zijn dan `m`:

```
HOW'TO PRINT series LESS'THAN m:
    FOR i IN series:
        SELECT:
            i<m: WRITE i
            ELSE: QUIT
```

4. UITGANGSPUNTEN VOOR HET ONTWERP VAN DE PROGRAMMEER-OMGEVING

In het bovenstaande is duidelijk geworden op welke manier de uitgangspunten van het B-project belichaamd zijn in de taal B. Uit deze uitgangspunten is niet onmiddellijk een gedetailleerd ontwerp voor de omgeving af te leiden, maar ze kunnen wel eerst iets nader worden toegespitst.

Omdat het uiteindelijke doel van het B-projekt een computer is die voor de

gebruiker B als moedertaal lijkt te hebben, dienen taal en omgeving een ondeelbare eenheid te vormen. (Bij eerdere implementaties die ingebed zijn in een ander systeem zal van deze inbedding zo weinig mogelijk voor de gebruiker merkbaar moeten zijn.) De omgeving is *geheel* gewijd aan het programmeren in B. De onderdelen van de omgeving, zoals de editor, kunnen hierdoor de gebruiker des te beter de helpende hand bieden.

De systeemfuncties dienen zo gelijk mogelijke gebruikers-interfaces te hebben. Het is niet praktisch te verlangen dat het mogelijk zal zijn aan elke uiting van de gebruiker een betekenis toe te kennen die onafhankelijk is van de onmiddellijke voorgeschiedenis. In die zin zullen er verschillende modes te onderscheiden zijn. Het is echter dringend gewenst dat als in de ene mode een zekere functie op zekere wijze aangeroepen kan worden, dat dan in andere modes, waar dat door de gebruiker redelijkerwijze verwacht zou worden, deze functie evenzeer beschikbaar is en *op gelijke wijze* aangeroepen kan worden. Een voorbeeld moge dit illustreren. Wanneer de gebruiker input intypt voor een `READ`-commando in een lopend programma, dan bevindt het systeem zich in een andere mode dan wanneer hij een commando geeft. Het is toch gewenst dat hij de tekst die hij intypt in beide modes op *gelijke* wijze kan editen alvorens op *gelijke* wijze kenbaar te maken dat hij klaar is met typen en editen. Analoge verschijnselen doen zich voor in de taal. Zo zijn voor lists operaties als `FOR el IN l` beschikbaar. Het ligt voor de hand dat de gebruiker, als hij een andere seriële datastructuur, zoals een text, onder handen heeft, verwacht ook deze te kunnen doorlopen. Welnu, die operatie is dan ook beschikbaar, en wordt op dezelfde manier genoteerd. In het dagelijks leven wordt dit principe ook wel gevolgd. Zo is het nuttig dat in auto's van verschillende merken het rempedaal steeds links van het gaspedaal is aangebracht.

In het ideale geval zou de taal B ook de commandotaal van de B-omgeving zijn. Als het voor sommige functies in de omgeving niet praktisch is ze als commando's in de taal toe te laten, dan moet de aanroep van zulke functies zo'n radikaal ander formaat hebben, dat de gebruiker niet in de verleiding komt ze in een B-programma te gebruiken. Zo'n functie zou bijvoorbeeld aangeroepen kunnen worden met een speciale toets.

Evenals in de taal, worden maar weinig constructies in de omgeving toegelaten, veel minder dan bijvoorbeeld de honderden verschillende commando's van UNIX. De macht van de omgeving moet voortspruiten uit de samenhang van de bouwstenen, niet uit het aantal toeters en bellen. Om die reden is in de taal slechts een handjevol datastructuren opgenomen, die, samen met de andere constructies, de gebruiker toch goede mogelijkheden bieden om zelf in andere wensen te voorzien. De samenhang tussen de bouwstenen van de omgeving zal mede hierin tot uitdrukking moeten komen, dat de gebruiker een eenvoudig intuïtief beeld van het gehele systeem geschetst kan worden dat hem in de meeste gevallen voldoende houvast biedt om het raadplegen van een manual overbodig te maken. Zo'n beeld zou aanknopingspunten moeten hebben met situaties uit het dagelijks leven, en op de gebruiker liefst een overzichtelijkere indruk maken dan een Kafka-eske bureaucratie.

Nu de taal zulke goede mogelijkheden biedt statische checks uit te voeren, dient het ontwerp van de omgeving deze mogelijkheden intact te laten. Dit criterium zal een grote rol moeten spelen bij de beslissing of voor een zekere omgevingsfunctie een commando gecreëerd wordt dat in de taal zelf kan worden opgenomen, of dat dit commando een radikaal ander formaat gegeven moet worden.

5. DE OMGEVING IN EERSTE BENADERING

Een belangrijke eigenschap van de B-omgeving wordt door de taal op een presenteerblaadje aangeboden: globale variabelen (d.w.z. niet lokaal in een unit) kunnen dienst doen als permanente files. Dat maakt invoering van allerhande speciale functies voor het manipuleren van files overbodig. Een voor de gebruiker nog belangrijker voordeel is, dat hij gegevens niet één voor één uit een file hoeft te lezen om er dan de datastructuur uit op te bouwen die door een vorig programma juist is afgebroken.

Als eerste benadering van een eenvoudig model van de omgeving kan het volgende beeld dienst doen. De gebruiker gebruikt het toetsenbord van zijn computer om "documenten" te typen. Dat sluit nauw aan bij het vertrouwde beeld van een schrijfmachine. Nu verschijnen de letters niet op een stuk papier, maar op het scherm van zijn computer, of, preciezer gezegd, het beeldscherm fungeert als een venster waardoor hij (een gedeelte van) zijn document kan zien. Bovendien heeft deze wonder-schrijfmachine heel wat meer edit-mogelijkheden dan een simpele correctietoets, maar daarover later. Als een document af is, kan het "terzijde" worden gelegd, om plaats te maken voor een nieuw te prepareren document. Het terzijde gelegde document is dan niet langer zichtbaar, maar ligt verderop op de "schrijftafel" van de gebruiker, en kan later weer zichtbaar gemaakt en geëdit worden. Een bijzonder document dat de gebruiker kan editen is de "session record". Dit fungeert als een lijst commando's voor de "dienaar" van de gebruiker, die in de computer verscholen zit en ook documenten kan maken en wijzigen. De dienaar wordt in B toegesproken en doet zijn best de commando's in de session record zo spoedig mogelijk uit te voeren.

Als de gebruiker in de session record van zijn nieuwe, juist uit de doos gehaalde computer typt:

```
PUT 'abacadabra' IN a
```

dan zal de dienaar er in een oogwenk voor zorgen dat er een document op de schrijftafel van de gebruiker belandt dat `"a"` heet en de tekst `'abacadabra'` bevat. (We herkennen hier een globale B-variabele, die immers ook file is.) De gebruiker kan document `"a"` ook wijzigen zonder tussenkomst van zijn dienaar, door het voor het venster te halen en te editen. Hij kan dan bijvoorbeeld een `'r'` plaatsen tussen de eerste `'b'` en de erop volgende `'a'`. Als hij nu als volgende commando aan de session record toevoegt

```
WRITE a
```

dan zal zijn dienaar deze opdracht uitvoeren door de inhoud van document `"a"` toe te voegen aan een ander speciaal document: de "output record". Als de gebruiker niets speciaals onderneemt, zal dit document ook door zijn venster waarneembaar zijn. (Het scherm kan dus uitzicht op meerdere documenten tegelijkertijd verschaffen.)

Als de gebruiker bij het typen van een commando een fout maakt, kan hij die door editen verbeteren. Als hij klaar is met het editen van één of meer commando's, dan kan hij dat te kennen geven door een speciale toets, gemerkt "Enter" bijvoorbeeld, in te drukken. Wat nu als hij zich achteraf realiseert dat hij het toch niet zo bedoeld had? Het is alleen voor door de wol geverfde computergebruikers aanvaardbaar dat de schade die door onbedoelde commando's is aangericht, onherstelbaar is. We bieden onze gebruiker dan ook de mogelijkheid de resultaten van zijn commando's ongedaan te maken, en wel door de session record te editen. Zodra die naar wens is, geeft de gebruiker Enter, en zijn dienaar gaat aan het werk om de situatie op de schrijftafel weer in overeenstemming te brengen met de tekst in de session record. Met hetzelfde mechanisme kan de gebruiker series commando's herhalen. Natuurlijk

zullen er grenzen zijn aan de mate waarin de gebruiker kan terugreizen in de tijd, maar in een handige implementatie kan hij een heel eind komen. Een principiëler probleem is: wat te doen als de gebruiker terug wil voorbij eigen edits, zoals het tussenvoegen van de `'r'` in `'abacadabra'`? Zulke handelingen, die zonder tussenkomst van de dienaar zijn verricht, zullen niet in de session record voorkomen. Een praktische strategie zou zijn deze edits evenzeer ongedaan te maken, maar dat vergt nadere studie, omdat dat misschien lang niet altijd is wat de gebruiker wil.

Een derde speciaal document is de "input record". Als de dienaar bij het uitvoeren van een unit een READ-command tegenkomt, verwacht hij de input in de input record. Die input kan er uitzien als:

```
65
```

of als

```
5*13
```

maar ook als

```
a*b
```

en zelfs als

```
PUT 0, 1 IN a, b
WHILE b<1000: PUT b, a+b IN a, b
RETURN b
```

Algemener: een expressie of de body van een YIELD-unit. Net als bij een YIELD-unit heeft evaluatie van input geen zijeffecten, zodat in het laatste voorbeeld de inhoud van eventuele globale variabelen `a` en `b` niet blijvend is aangetast.

Tot zover heeft de gebruiker dus een schrijftafel met permanente variabelen, units en een drietal speciale documenten: session record, output record en input record. Een eenvoudige generalisatie geeft de mogelijkheid meerdere drietallen te hebben, die elk een proces vertegenwoordigen. De dienaar zal het wat drukker krijgen, en we zullen een handige scheduler voor het verdelen van zijn aandacht moeten verzinnen. Als we aan elk proces nog een vierde speciaal document toevoegen, een status record, waarin te zien is of het proces loopt of niet, dan kan de gebruiker daarin aangeven of hij een proces tijdelijk wil stoppen. Net als bij het editen van vorige commando's in de session record, geldt hier de regel: wat in de record staat is geldig, de situatie moet eraan worden aangepast. Een soortgelijke regel is: als de dienaar een wijziging aanbrengt in een document dat op het scherm zichtbaar is, dan moet die verandering ook op het scherm worden aangebracht. Het scherm geeft dus altijd de actuele situatie aan. Het venster geeft uitzicht op de documenten zelf, niet op copieën ervan.

Als de gebruiker ook in staat gesteld wordt zelf de speciale documenten aan processen toe te wijzen, dan kan hij processen in serie schakelen door de output record van het ene proces als de input record van het volgende proces aan te wijzen. Ook coroutines kunnen zo gemodelleerd worden. Het aanwijzen van de output record van een proces als session record van een ander proces maakt het in beginsel ook mogelijk een door een ander proces berekend programma te laten uitvoeren. Als een proces alle records van andere processen kan inspecteren en veranderen, dan stelt dat de gebruiker zelfs in staat zijn eigen scheduler voor zijn dienaar te schrijven. Of deze en andere mogelijkheden niet wat te veel van het goede zijn voor de eenvoudige gebruiker waarop het B-project mikt, valt te bezien. In ieder geval zullen zulke mogelijkheden een nuttige plaats kunnen vinden in Extended B, een uitgebreide versie

van de taal ten behoeve van systeemprogrammeerwerk.

Een volgende generalisatie is die naar meerdere schrijftafels. De gebruiker kan dan bijvoorbeeld één schrijftafel reserveren voor units en permanente variabelen voor het voeren van zijn boekhouding, een andere voor het bijhouden van zijn agenda en een derde voor getaltheorie. Doordat namen lokaal zijn ten opzichte van een schrijftafel, wordt de gebruiker zo ontslagen van de plicht zich ervan te verzekeren dat elke nieuw te introduceren naam niet al in een ander verband voorkomt. Het is handig als er een centrale schrijftafel is met units en permanente variabelen die op alle schrijftafels gebruikt kunnen worden. Als de gebruiker incidenteel een unit of permanente variabele van een der andere schrijftafels wil gebruiken, dan zal hij deze expliciet moeten importeren. Of daaraan een B-commando kan worden gewijd, ook te gebruiken in units, valt nog te bezien.

Het beeld van de verschillende schrijftafels met elk hun verschillende processen roept de gedachte op of de componenten van een schrijftafel misschien opgenomen zouden moeten zijn in een B-datastructuur, met als consequentie dat deze dan met bestaande B-commando's gemanipuleerd zou kunnen worden. Dit zou het ideaal van een ééntalige omgeving aanmerkelijk dichterbij brengen. Toch is het ook hier de vraag of dit niet teveel van het goede is. Bovendien geeft dit de mogelijkheid dat een unit de tekst van een andere unit verandert zonder dat de statische type checking zijn werk nuttig kan doen. De mogelijkheid is in ieder geval kandidaat voor Extended B.

6. DE EDITOR

De editor is dat deel van de omgeving dat de gebruiker in staat stelt documenten op zijn scherm aan te maken, terzijde te leggen, op het scherm te brengen, op verschillende manieren op het scherm te arrangeren en te veranderen. Een fundamentele keus die bij het ontwerp van een editor gemaakt moet worden is die tussen:

a. er is een mode waarin elk getypt karakter op het scherm wordt toegevoegd, en er is een andere mode waarin karakters geïnterpreteerd worden als opdrachten aan de editor (bijvoorbeeld: switch de mode);
b. elk karakter dat getypt wordt verschijnt op het scherm, ofwel tussenvoegend ofwel overschrijvend, en opdrachten aan de editor kunnen alleen met speciale toetsen gegeven worden.

Voor de B-editor kiezen we een restrictieve, syntax-gerichte vorm van alternatief b. In een te editen document zijn steeds aanwezig één "focus" en nul of meer "holes". Een hole is een lege plek in het document, de focus is een stuk van het document waarop de aandacht gericht is. Focus en holes vallen steeds samen met een kleinere of grotere syntactische eenheid uit de boom die met het document correspondeert. Met behulp van speciale toetsen kan de gebruiker de focus bewegen, vergroten totdat hij samenvalt met de eerste omvattende syntactische eenheid, of juist verkleinen tot de eerste kleinere syntactische eenheid binnen de huidige focus. Met een speciale toets kan het stuk document dat de focus beslaat verwijderd worden. Het verwijderde stuk blijft bewaard op een verwijder-stack, en op de lege plaats ontstaat een hole. Als de focus een hole beslaat, dan wordt elk getypt karakter voor die hole tussengevoegd. Karakters getypt als de focus niet samenvalt met een hole zijn illegitiem, en worden bijvoorbeeld niet geëchood.

Hoe de editor met focus en holes en met zijn kennis van taal en omgeving de gebruiker terzijde kan staan, moge uit een enkel voorbeeld blijken. Als de gebruiker bij het intypen van een B-unit aan het begin van een regel een F typt, zal de editor dit laten verschijnen:

```
F■OR □ IN□ :
    □
□
```

(Hier is □ hier een typografische aanduiding van een hole en ■ van een hole waarmee de focus samenvalt.) Als de gebruiker inderdaad bedoelde een FOR-commando te beginnen, kan hij dit aangeven met een speciale toets, bijvoorbeeld "Next". Dat ruimt de hole op waarmee de focus samenvalt en brengt de focus naar de hole tussen FOR en IN. Als hij daar klaar is, en bijvoorbeeld char heeft getypt, geeft hij weer Next, en hij ziet:

```
FOR char IN ■ :
    □
□
```

Als nu vervolgens de hole na IN naar behoren is gevuld, en de gebruiker typt nu W in de hole op de volgende regel, dan ziet de gebruiker:

```
FOR char IN text:
    W■HILE □ :
        □
    □
□
```

Als de gebruiker niet WHILE bedoeld had, maar WRITE, dan typt hij onverstoord verder en geeft dus een R:

```
FOR char IN text:
    WR■ITE □
    □
□
```

Nadat de gebruiker de hole na WRITE heeft gevuld en Next gegeven heeft, komt de focus in de hole op de volgende regel. Als er nu geen commando meer moet volgen op het geïndenteerde niveau, dan geeft de gebruiker weer Next, en hij krijgt:

```
FOR char IN text:
    WRITE char
■
```

Nu bedenkt de gebruiker zich: in plaats van die twee regels had hij wel in één keer WRITE text kunnen schrijven. Met behulp van de toets "Previous" brengt hij de cursor terug naar de vorige syntactische eenheid van hetzelfde niveau:

```
FOR char IN text:
    WRITE char
□
```

(De onderlijning is hier een typografische aanduiding van de plaats van de focus. Op het scherm zal iets anders te zien zijn, bijvoorbeeld een vlek van een andere kleur dan de rest van het scherm.) De toets "Delete" verwijdert nu de tekst waarmee de focus samenvalt, en een hole blijft ter plaatse achter:

```
■
```

We laten de gebruiker nu verder alleen met zijn probleem, en laten intussen nog wat

andere features van de editor de revue passeren. (Een uitgebreidere behandeling is te vinden in [6].)

-- Als de gebruiker van een IF-keyword SELECT maakt, dan wijzigt de editor automatisch de structuur ter plaatse. Hetzelfde geldt bij verandering van unit naar refinement.
-- Als het keyword-skelet van een unit wordt veranderd, helpt de editor met het veranderen van alle aanroepen.
-- Als de gebruiker het giswerk van de editor negeert en gewoon typt wat hij in zijn tekst wil krijgen, dan werkt dat ook.
-- Stukken tekst die met Delete verwijderd zijn, blijven bewaard op een stack en kunnen van daaruit weer in het document gecopieerd worden.
-- Als de gebruiker een ander soort document onder handen heeft dan een unit, bijvoorbeeld een datastructuur, dan gebruikt de editor zijn syntactische kennis van de betreffende soort structuur. Het is niet onmiddellijk duidelijk of voor gewone teksten nog een aparte, niet syntax-gerichte editor nodig zal zijn.
-- Een belangrijke functie van de (en van elke) omgeving is het zoeken naar plaatsen binnen een gegeven gebied die aan een bepaald signalement voldoen. (In UNIX worden daarvoor, naast de mogelijkheden die daarvoor in de editors gegeven worden, *grep* en afgeleide functies gebruikt.) Het is niet duidelijk of deze functie in de B-editor thuishoort dan wel in de B-taal. Een aantrekkelijk model voor zo'n zoekactie is het volgende. Als het te doorzoeken gebied bekend is, bijvoorbeeld alle units op de huidige schrijftafel of alleen een bepaalde datastructuur, dan verschijnt een soort landkaart op het scherm die, in het algemeen op sterk gereduceerde schaal, dat hele gebied in kaart brengt. (In het geval van de hele schrijftafel past misschien niet eens van elke unit de kop op het scherm, maar dan kan wel volstaan worden met één karakter per unit.) Op deze kaart lichten vervolgens de plekken op waar iets gevonden is dat aan het signalement beantwoordt. De gebruiker kan nu inzoomen op een gekozen lichtpunt, de lokale context inspecteren, er iets wijzigen, weer uitzoomen, de operatie "ga naar het volgende lichtpunt" aanroepen, etc. De hiervoor nodige mogelijkheid van het verkleind weergeven van documenten kan de gebruiker wellicht ook nuttig gebruiken om een gehele schrijftafel, of grote delen ervan, permanent zichtbaar, hoewel niet leesbaar, te houden. (Vergelijk Smalltalk [7].)
-- Uit de ervaringen die tot nu toe met de huidige implementatie van de taal en zijn embryonale omgeving zijn opgedaan is niet met zekerheid vast te stellen of de gebruiker in het uiteindelijke B-systeem behoefte zal hebben aan speciale tracefaciliteiten. Mocht dat zo zijn, dan zou als volgt kunnen worden aangesloten bij de mogelijkheden die de editor al biedt. De gebruiker kan zijn dienaar gebieden steeds de unit die uitgevoerd wordt op het scherm te vertonen, en de uitvoering stap voor stap te doen. De grootte van zo'n stap wordt aangegeven door een focus, die de gebruiker op dezelfde manier manipuleren kan als wanneer hij de editor gebruikt. Door de focus klein te maken, kan hij de uitvoering van de unit heel nauwkeurig volgen, als hij de focus groter maakt kan hij met grote stappen door een unit heenlopen. Misschien krijgt hij ook de mogelijkheid in de tijd terug te gaan door de focus steeds een stapje terug te plaatsen. Of dat niet een beeld geeft dat te veel interfereert met het teruggaan in de tijd via het veranderen van de session record, moet nader bekeken worden. Misschien ook is het vruchtbaar die twee manieren van omgekeerde berekening tot één nieuwe manier om te smeden.

7. ANDERE FUNCTIES VAN DE OMGEVING

Enkele andere voorzieningen die in de B-omgeving thuishoren, maar die een minder centrale rol vervullen, zijn de volgende.

-- Graphics, weliswaar meer een zaak van de taal, maar het valt te verwachten, dat gevolgen voor de omgeving niet zullen uitblijven.
-- Hulpfaciliteiten, die liefst op verschillende niveaus en in verschillende mate van uitgebreidheid de gebruiker als hij het echt niet meer weet uitleg kunnen geven.
-- Beveiliging, in enige mate nodig, ook al wordt gemikt op een eenpersoons-computer. Beveiliging is nodig tegen opzettelijke inmenging van derden, maar ook tegen grove eigen vergissingen van de gebruiker. In dit kader past ook de wens, dat de gebruiker bij het aanzetten van zijn computer dezelfde situatie aantreft als hij verliet toen hij het apparaat de laatste keer uitzette.

Tenslotte

Een woord van dank is op zijn plaats aan allen die in de dinsdagochtendbesprekingen van de B-groep de ideeën hebben geleverd waaruit het bovenstaande een selectie is.

REFERENTIES

[1] Geurts, L.J.M., & L.G.L.T. Meertens, Designing a beginners' programming language, New Directions in Programming Languages 1975, 1-18, (S.A. Schuman, ed.), Roquencourt, 1976.
[2] Meertens, L.G.L.T., Draft Proposal for the B Programming Language - Semi-Formal Definition, Mathematisch Centrum, 1981.
[3] Geurts, L.J.M., An overview of the B programming language or B without tears, SIGPLAN Notices 17(12), 49-58 (1982).
[4] Meertens, L.G.L.T., Incremental polymorphic type checking in B, Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, ACM, 1983.
[5] Krijnen, T.J.G., & L.G.L.T. Meertens, Making B-trees work for B, Rapport IW 219, Mathematisch Centrum, 1983.
[6] Nienhuis, A.J.C., On the design of an editor for the B programming language, doctoraalscriptie Univ. van Amsterdam (te verschijnen).
[7] Ingalls, D.H.H., The Smalltalk-76 programming system - design and implementation, Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, ACM, 1978, 9-16.

FORTH: MACHINE, TAAL EN OMGEVING

Hans W.J. Buwalda
Forth Team Utrecht*

SAMENVATTING

De programmeertaal FORTH wordt bekeken op zijn mogelijkheden als programmeeromgeving. Eerst wordt iets van de taal zelf gepresenteerd, waarbij de nadruk vooral ligt op conceptuele aspecten en details worden vermeden. Daarna wordt ingegaan op de rol die FORTH kan spelen als machinemodel, als programmeertaal en als programmeeromgeving.

INLEIDING

FORTH is als programmeertaal door Charles H. Moore ongeveer 10 jaar geleden ontwikkeld 'to make myself a more productive programmer', zoals hij zelf schrijft [1,2]. Hij heeft bij het ontwerp van de taal naar de toekomst gekeken, naar wat hij zag als de vierde generatie software. De naam FORTH is dan ook een verkorte versie van 'fourth': vierde generatie taal, bedoeld voor personal computing.

De taal FORTH, die naast talen als BASIC, PASCAL, C en LISP, snel populair aan het worden is voor personal computers, wijkt sterk af van alle andere talen en biedt unieke mogelijkheden als programmeeromgeving.

FORTH biedt een interactieve omgeving aan, waar integratie, modularisatie, overdraagbaarheid en niet in de laatste plaats een speelse gebruikersvriendelijkheid de hoofdrol spelen. FORTH speelt in deze omgeving dubbelrollen als virtuele machine, assembler, systeemprogrammeertaal, bedrijfssysteem, commandotaal, hogere programmeertaal, aanspreektaal voor subsystemen (zoals een editor), etc.

Het creëren van 'nieuwe modes', waar Jan Heering voor pleit [3], is bij FORTH de enig mogelijke manier van programmeren. Een programma schrijven in FORTH betekent de taal uitbreiden in de richting van de applicatie. Echter, de groei loopt geleidelijk; er is meestal geen sprake van een heel nieuw niveau. Aanwezige software blijft direct bereikbaar: het 'leeneffect', waar in dezelfde bijdrage ook over gesproken wordt.

Deze bijdrage valt in twee gedeelten uiteen. Hoofdstuk 1 geeft een inleiding op het programmeren in FORTH. Hoofdstuk 2 beschrijft hoe binnen FORTH een virtuele machine via een programmeertaal de basis vormt voor een programmeeromgeving.

* Forth Team Utrecht, Maliesingel 20, 3581 BE Utrecht, Tel. 030-318896.

1. PROGRAMMEREN IN FORTH

Dit hoofdstuk gaat over het programmeren in FORTH. Het is slechts een verkenning in vogelvlucht. FORTH echt leren kan alleen maar door ermee te spelen. Voor een goede handleiding daarbij zij verwezen naar het boek van Brodie [4].

1.1. De rekenstack en het routine-mechanisme

FORTH is opgebouwd rond een rekenstack. We kunnen getallen op deze stack zetten door ze in te tikken:

```
11 22 33 ok
```

Op de stack staan nu de getallen 11, 22 en 33, met 33 bovenop. De 'ok' is de prompt van FORTH.

Op deze getallen kunnen we operaties gaan toepassen, bijvoorbeeld

```
+ ok
```

die de bovenste twee getallen optelt en alleen het resultaat op de stack achterlaat. Het vermoeden bestaat dat op de stack nu de getallen 11 en 55 staan. We kunnen dat bekijken met

```
. .       55 11 ok
```

(Ik zal steeds enige spaties tussen de input en de output zetten omwille van de duidelijkheid). De '.' operator pakt een getal van de stack en drukt dit getal af. FORTH kent ook enige operatoren die op de stack zelf werken zoals

```
11 22 SWAP . .       11 22 ok
```

SWAP verwisselt de volgorde van de bovenste twee getallen op de stack. Andere voorbeelden van stackmanipulaties zijn DUP (dupliceer het topelement) en DROP (gooi een getal van de stack weg).

Een belangrijk onderdeel van FORTH vormen de routines. Routines zijn voor FORTH net zo belangrijk als S-expressies voor LISP: de hele taal draait erom. Een voorbeeld van een subroutinedefinitie:

```
: LINE 2 * 3 + ;     ok

0 LINE .  3 ok
1 LINE .  5 ok
2 LINE .  7 ok
  ...
```

Routines worden ook wel kortweg *definities* of *woorden* genoemd. De ':' leidt de definitie in, daarna volgt de naam van de definitie. Na de naam komt de body en de ';' sluit de definitie af. We mogen elke naam gebruiken, die in ons hoofd opkomt, zolang er maar geen spatie in zit, bijvoorbeeld:

```
: 1+ 1 + ;

1 1+ .   2 ok
```

FORTH definities belanden in een datastructuur genaamd *dictionary*. De routines en hun namen zijn door linkvelden met elkaar verbonden. Een dergelijke gelinkte lijst

heet een *vocabulary*. In de dictionary kan meer dan één vocabulary voorkomen. De normale gang van zaken is dat elke applicatie zijn eigen vocabulary heeft. De woorden in een vocabulary vormen gelijktijdig een user interface met de applicatie en een subroutine library.

Het variabele-mechanisme in FORTH:

```
5 VARIABLE X  ok
X .        5 ok

10 TO X      ok

X .        10 ok
```

Een variabele is zelf ook een routine. De routine VARIABLE, die dus kennelijk in staat is andere routines aan te maken is geïmplementeerd met het <BUILDS DOES>-mechanisme, waar ik nog op terugkom. De routine TO zet een vlaggetje op 1. Dit vlaggetje wordt door een met VARIABLE aangemaakte routine getest en bevat informatie over wat er moet gebeuren. Als het vlaggetje op 0 staat, dan wordt een waarde opgehaald en op de stack gezet. Staat het op 1, dan wordt juist een waarde van de stack afgehaald en weggeschreven. Uiteindelijk zet de variabele het vlaggetje weer op nul.

Het hierboven voor variabelen beschreven accessmechanisme blijkt heel algemeen bruikbaar te zijn. Bij de implementatie van de blokkenwereld bijvoorbeeld, waar ik verderop een klein deel van ga behandelen, wordt de positie van een 'robot arm' bestuurd met dit accessmechanisme. Ophalen van de waarde levert de huidige positie van de arm op, assignatie heeft een beweging van de arm tot gevolg. Er is een routine genaamd ARM. Het commando 5 TO ARM brengt de arm op posititie 5; zeggen we daarna echter ARM, dan wordt de waarde van de huidige positie van de arm op de stack gezet - in dit geval is dat 5.

1.2. Controlestructuren

FORTH kent geen sprongopdrachten. Alle control flow moet worden gedaan met daartoe geëigende statements. Als voorbeeld het if-statement:

```
: TEST
    IF ." Waar "
    ELSE ." Onwaar "
    ENDIF
; ok

0 TEST     Onwaar ok
1 TEST     Waar ok
```

Het if-statement heeft wel een wat minder gebruikelijk uiterlijk; IF test het getal bovenop de top van de stack en werkt dus postfix, net als de rest van FORTH. In FORTH betekent 0 'onwaar'; elk getal ongelijk aan 0 wordt als 'waar' opgevat.

FORTH kent verschillende loop-statements. Bijvoorbeeld:

```
: SKIP-TO-100       \ druk getallen af tot we 100 tegenkomen
     BEGIN          \ start van de loop
       DUP          \ dupliceer even bovenste getal
       100 <>       \ en kijk of het ongelijk 100 is
     WHILE          \ is dat het geval?
       .            \ druk dan het getal af
     REPEAT         \ en spring terug naar BEGIN
     DROP           \ gooi getal 100 weg
;

100 1 2 3 4 SKIP-TO-100        4 3 2 1 ok
```

Merk op dat we de getallen dupliceren, voor we ze met 100 vergelijken. Dit is omdat de '<>' operatie zijn argumenten van de stack gooit.

Een ander voorbeeld van een loop-statement in FORTH is de DO LOOP constructie. Bijvoorbeeld:

```
: IOTA 0 DO I . LOOP ;

10 IOTA 0 1 2 3 4 5 6 7 8 9
```

DO neemt twee argumenten - in dit geval 10 en 0 - en loopt dan van het bovenste argument als beginwaarde naar het argument daaronder minus 1 als eindwaarde. Deze argumentenafhandeling verraadt dat DO LOOP oorpronkelijk vooral bedoeld was voor de behandeling van byte arrays. FORTH kent ook een variant DO +LOOP genaamd, waarbij +LOOP een stapgrootte van de stack haalt en bij de index optelt.

Het bijzondere van control flow statements zit echter in de implementatie ervan, gemaakt in FORTH zelf. Het keyword IF is in feite de naam van een routine. Deze routine is *immediate*, dat wil zeggen als het FORTH systeem tijdens de definitie van een andere routine de naam IF tegenkomt wordt de routine IF uitgevoerd. Het zelfde geldt voor ELSE, ENDIF en alle woorden die met control flow te maken hebben.

De FORTH stack wordt tijdens compilatie van een routine dan ook voornamelijk gebruikt voor de communicatie tussen control flow woorden. Deze communicatie gaat meestal over waar sprongopdrachten moeten komen (op dit niveau kent FORTH ze wel) en waar ze naar toe gaan. Ook de syntax checking gaat uiteraard via de stack met behulp van unieke controlegetallen. Dit principe van syntaxherkenning is niet anders dan bij de compilers van andere programmeertalen. Ter illustratie een eenvoudige implementatie van IF en ENDIF. COMPILE zet het executie-adres van een routine in de dictionary. In dit voorbeeld is dat 0BRANCH, een routine, die een sprong uitvoert als het getal boven op de stack een nul was. HERE noteert het adres in de dictionary waar we gebleven zijn met compileren en ALLOT reserveert ruimte in die dictionary op die plaats - en verhoogt daarbij de waarde van HERE. De routine '!' - spreek uit 'store' - tenslotte neemt een getal en een adres van de stack en schrijft het getal op dat adres weg.

```
: IF COMPILE 0BRANCH  \ zet sprong opdracht in dictionary
    HERE              \ dictionary pointer (voor ENDIF)
    2 ALLOT           \ reserveer twee bytes in de dictionary
    1                 \ controle getal
; IMMEDIATE
```

```
: ENDIF
    1 =             \ test of er een 1 op de stack staat
    5 ?ERROR        \ zo niet geef foutmelding nummer 5
    HERE            \ huidige dictionary pointer is target
    SWAP !          \ backpatch de 0BRANCH target
; IMMEDIATE
```

Omdat echter niets in FORTH een speciale status heeft, kan een programmeur de taal naar behoefte aanpassen, bijvoorbeeld in de richting van infix-stijl of zelfs natuurlijke taal. En deze aanpassingen zijn vaak dermate eenvoudig en voor de hand liggend, dat ze ook inderdaad worden toegepast. Laten we bij wijze van voorbeeld het ALGOL 68 if-then-else-elif-fi-statement voor FORTH implementeren. De ALGOL 68 keywords noteer ik tussen aanhalingstekens ter onderscheiding van de al bestaande FORTH woorden. Het in dit voorbeeld gebruikte woord [COMPILE] forceert de compilatie van immediate woorden, zoals IF, die anders zouden worden aangeroepen. De nieuw te definiëren woorden krijgen allemaal de immediate status. Ze communiceren met elkaar via de stack.

```
: 'IF'
     100    \ zet 100 op de stack als 'bodem'
; IMMEDIATE

: 'THEN' [COMPILE] IF    ; IMMEDIATE
: 'ELSE' [COMPILE] ELSE  ; IMMEDIATE
: 'ELIF' [COMPILE] ELSE  ; IMMEDIATE

: 'FI'                   \ FI moet wat werk doen
    BEGIN
      DUP 100 <>         \ zijn we nog niet bij de 'bodem' ?
    WHILE
      [COMPILE] ENDIF    \ zo nee, probeer een IF af te sluiten
    REPEAT
    DROP                 \ drop de 'bodem'
; IMMEDIATE
```

Voorbeeld van gebruik:

```
: TEST
     'IF' 1 1 + 3 = 'THEN' ." daar horen we van op "
    'ELIF' 1 1 + 2 = 'THEN' ." dat wisten we al "
    'ELSE' ." niet helemaal lekker "
     'FI'
;

TEST     dat wisten we al ok
```

en zie daar, een beschaafd ogend infix if-statement volgens het ALGOL 68 principe.

Zo heeft de gebruiker in FORTH vrij spel om naar willekeur met allerlei syntaxvormen te experimenteren. Dit kan omdat FORTH een taal is, die 'zichzelf kent'.

1.3. Routines, die routines maken

Een van de lastigste maar ook een van de interessantste constructies in FORTH is de <BUILDS DOES>-constructie. Het idee voor dit concept is onstaan uit de praktijk. Het is in FORTH gebruikelijk voor zelfs het kleinste meer dan één maal voorkomende programmafragment een routine te schrijven. Op deze wijze kunnen er groepen routines onstaan, die slechts op ondergeschikte punten van elkaar verschillen.

Laten we als eenvoudig voorbeeld nemen het afdrukken van een speciaal character, zoals een spatie, een carriage return of een bell signaal. Een voor de hand liggende manier is dit als een lijst subroutines uit te schrijven:

```
: SPACE BL EMIT ;   \ EMIT drukt een character af
: CR    13 EMIT ;
: BELL  7  EMIT ;
```

<BUILDS DOES> kan nu worden gebruikt om een dergelijke klasse van gelijksoortige routines te beschrijven. De afzonderlijke routines kunnen daarbij als leden van deze klasse gegenereerd worden. Deze routines verschillen van elkaar alleen in een parametergebied, dat over aanroepen heen blijft bestaan.

Algemene vorm van een <BUILDS DOES> constructie:

```
: (klasse-naam)
   <BUILDS
     ( definitie parametergebied )

   DOES>
     ( gemeenschappelijke routine body )
;
```

De gemeenschappelijke routinebody krijgt op de stack het adres van het in het <BUILDS-gedeelte gedefinieerde parametergebied binnen. In ons geval:

```
: SPECIAL
   <BUILDS       \ maak een routine aan
      ,          \ ',' bewaart een getal in de dictionary

   DOES>         \ begin gemeenschappelijk executie gedeelte
      @          \ '@' haalt dat getal weer op
      EMIT       \ druk getal af als ASCII character
;
13 SPECIAL CR    \ een routine CR is gemaakt en 13 is bewaard
7  SPECIAL BELL
BL SPECIAL SPACE
&★ SPECIAL STER \ '&★' zet de ASCII-waarde van '★' op de stack

STER SPACE STER SPACE      ★ ★ ok
```

Een tweede voorbeeld:

```
: FANCY
   <BUILDS
      ,                          \ bewaar een character

   DOES>
      @                          \ haal character op
      SWAP                       \ verwissel bovenste stackelementen
      0 DO DUP EMIT LOOP         \ druk character aantal malen af
      DROP                       \ drop het character van de stack
;

&★ FANCY STARS
&− FANCY STRIPES

5 STARS          ★★★★★ ok
3 STRIPES        −−− ok
```

Een ander eenvoudig practisch voorbeeld is de implementatie van het CONSTANT-mechanisme in FORTH:

```
: CONSTANT <BUILDS , DOES> @ ;

1 CONSTANT EEN
2 CONSTANT TWEE

EEN .     1 ok
TWEE .    2 ok
```

In FORTH worden veruit de meeste datastructuren met <BUILDS DOES> geïmplementeerd. Ook het vocabulary-mechanisme is een voorbeeld van een <BUILDS DOES> routine. FORTH vocabularies zijn gelinkte lijsten van routines, die zich in de dictionary bevinden. FORTH kent op ieder moment een *current* en een *context* vocabulary. De adressen van deze vocabularies haalt FORTH uit de systeemvariabelen CURRENT en CONTEXT. Nieuwe routines worden toegevoegd aan de current vocabulary; de context vocabulary wordt gebruikt om woorden in op te zoeken. Als de naam van een vocabulary in een programma voorkomt, wordt die vocabulary vanaf dat punt context vocabulary. Het woord DEFINITIONS maakt van de context vocabulary ook de current vocabulary.

Een (vereenvoudigde) implementatie ziet er ongeveer als volgt uit:

```
: VOCABULARY
    <BUILDS
      IMMEDIATE     \ ook binnen een definitie aan te zetten
      0 ,           \ link naar de eerste header (nog leeg)

    DOES>
      TO CONTEXT    \ maak context vocabulary
;
```

```
: DEFINITIONS
    CONTEXT TO CURRENT
;

VOCABULARY PLOTLIB      \ nieuwe vocabulary PLOTLIB

PLOTLIB DEFINITIONS     \ aan PLOTLIB gaan we routines toevoegen

: MOVE ... ;
: DRAW ... ;
: PLOT ... ;

FORTH DEFINITIONS       \ FORTH is de standaard vocabulary

: BOX                   \ BOX komt dus in FORTH

    PLOTLIB             \ we gaan plotroutines gebruiken
      10 10 MOVE        \ ga naar positie (10,10)
      10 20 DRAW        \ teken lijn van (10,10) naar (10,20)
      20 20 DRAW
      20 10 DRAW
      10 10 DRAW
    FORTH               \ maakt PLOTLIB weer onzichtbaar
;
```

(De vereenvoudigingen in dit voorbeeld hebben vooral betrekking op het hanteren van meer dan één vocabulary).

Wat ik met dit voorbeeld duidelijk wil maken is hoe een op zich erg intern concept als een gelinkte lijst in FORTH in een stap elegant bruikbaar gemaakt kan worden, zowel op taal- als op gebruikersniveau.

Als een laatste voorbeeld een paar krenten uit een implementatie van het blokken-wereld demonstratieprogramma. De X-dimensie loopt in dit voorbeeld van links naar rechts; de Y-dimensie loopt van boven naar onder.

```
0 VARIABLE ^DESCRIPT    \ pointer naar een objectdescriptor

: VELD                  \ velden binnen een descriptor
    <BUILDS
      ,                 \ bewaar offset binnen descriptor

    DOES>
      @ ^DESCRIPT +     \ tel offset bij pointer op
      @                 \ haal waarde veld op
;

0 VELD TEKEN-ROUTINE
2 VELD HOOGTE
4 VELD BREEDTE
6 VELD X
8 VELD Y
```

```
\ Een paar voorbeelden van tekenroutines

: RECHTHOEKIG
    X              Y HOOGTE +  MOVE    \ ga naar links onder

    X BREEDTE +    Y HOOGTE +  DRAW    \ onderzijde
    X BREEDTE +    Y           DRAW    \ rechterzijde
    X              Y           DRAW    \ bovenzijde
    X              Y HOOGTE +  DRAW    \ linkerzijde
;

: DRIEHOEKIG
    X                Y HOOGTE + MOVE   \ links onder beginnen

    X BREEDTE +      Y HOOGTE + DRAW   \ basis
    X BREEDTE 2/ +   Y          DRAW   \ rechter schuine zijde
    X                Y HOOGTE + DRAW   \ linker schuine zijde
;

\ Voorwerp klassedefinitie:
: VOORWERP              \ op de stack: breedte hoogte
    <BUILDS
      [COMPILE] 'X      \ bepaal routine-adres ('X zoekt dit op)
      ,                 \ wordt tekenroutine
      ,                 \ hoogte
      ,                 \ breedte
      LOCATIE , ,       \ bereken een nieuwe locatie en bewaar die
                        \ (wordt hier verder niet behandeld)
    DOES>
      TO ^DESCRIPT      \ parameterveld wordt descriptor
      TEKEN-ROUTINE     \ zet adres tekenroutine op de stack
      EXECUTE           \ executeer dit adres (d.w.z. teken object)
;
\ Een paar instanties van voorwerpen
\ RECHTHOEKIG is de tekenroutine

40 40 VOORWERP VIERKANT RECHTHOEKIG
40 2  VOORWERP DEKSEL RECHTHOEKIG
40 40 VOORWERP PYRAMIDE DRIEHOEKIG
```

Wat we hier dus zien is dat we eerst een begrip 'huidige descriptor' invoeren, met een paar velden voor wat relevante informatie. Deze informatie betreft de omvang en de plaats van het voorwerp en het adres van een routine waarmee het voorwerp - uitgaande van de omvang en de locatie - getekend kan worden. De descriptors worden aangemaakt in het <BUILDS-gedeelte van de routine VOORWERP en komen terecht in de dictionary. Het DOES>-gedeelte maakt van de descriptor aangemaakt in het <BUILDS-gedeelte de huidige descriptor en tekent het voorwerp door de geassocieerde tekenroutine aan te roepen.

1.4. FORTH en assemblertaal

Ook vermeldenswaard is de assemblertaal-interface van FORTH. Veel routines zijn gedefinieerd in FORTH; hun code is opgebouwd uit aanroepen van eerder gedefinieerde routines. FORTH routines mogen echter ook in assemblertaal worden geschreven. Standaard kent FORTH om deze in assemblertaal geschreven routines te kunnen vertalen een eigen assembler. Hierover valt veel te verhalen. Als voorbeeld een FORTH assemblertaal versie voor '1+' (tel 1 op bij het getal bovenop de stack, het higher byte van dit getal wordt aangeduid met S1H en het lower byte met S1L). Deze routine ziet er op een 6502 microprocessor (een 8-bitter, vandaar de twee stappen) ongeveer zo uit:

```
CODE 1+
        S1L  INC,       \ verhoog lower byte
    0=  IF,             \ carry?
        S1H  INC,       \ zo ja, verhoog higher byte
    ENDIF,
             NEXT,      \ terug naar FORTH
ENDCODE
```

Er vallen een paar dingen op. In de eerste plaats blijken laag-niveau routines een syntax te hebben, die analoog is aan die van FORTH. De routine begint met CODE en eindigt met ENDCODE, te vergelijken met ':' en ';' bij FORTH routines. Verder wordt de control flow genoteerd met statement-achtige uitdrukkingen en niet met sprongopdrachten. Er worden dus ook geen labels gebruikt.

In de tweede plaats staat bij een opcode als INC, het argument vóór de opcode, een typisch FORTH postfix-stijltje dus. Dit is vooral omdat FORTH gebruikt kan worden om argumentwaarden te berekenen, die de opcodes (geïmplementeerd als FORTH routines) dan via de stack binnen krijgen.

In de derde plaats eindigen opcodes op een ','. Dit is een conventie om onder alle omstandigheden FORTH en assemblertaal uit elkaar te kunnen houden.

Tenslotte eindigt de routine met NEXT,. Dit is een macro, die de FORTH inner interpreter aanroept. Macro's zijn uiteraard niets anders dan FORTH routines, die tijdens de assembleerfase worden aangeroepen. S1L en S1H zijn andere voorbeelden van dit soort macro's.

1.5. Parameteroverdracht in FORTH

De parameteroverdracht tussen FORTH routines vindt plaats via een stack. Deze stack, die in andere programmeertalen meestal verborgen gehouden wordt, moet door een FORTH programmeur expliciet worden gemanipuleerd. Een voordeel van deze aanpak is de flexibiliteit bij het ontwerp van de communicatie tussen een routine en zijn omgeving. Dit blijkt de modulariteit van applicaties enorm te bevorderen. Een FORTH programmeur is al snel geneigd heel erg veel subroutines te maken, omdat deze snel en goed aan elkaar te passen zijn. Het succes van FORTH als programmeertaal is dan ook nog het best te vergelijken met dat van LEGO, het bekende en veel bekroonde speelgoed. Het gemakkelijk en voor de hand liggend in elkaar passen van de bouwstenen stimuleert de creativiteit. Bovendien ziet de bouwer bij beide systemen meteen resultaat van zijn werk.

De meest opvallende konsekwentie van het FORTH parametermechanisme is de voor de taal zo karakteristieke postfix vorm. Of dit een voor- of een nadeel is hangt af van factoren als smaak en gewoonte. Mijn eigen ervaring is dat het snel went en dat je al gauw niet anders meer wil, omdat het erg aansluit bij je manier van denken (althans bij de mijne).

Een andere eigenschap van FORTH is het ontbreken van controle op het aantal en het type van de input- en outputparameters van een routine. Bij systeemprogrammering is dit meestal een voordeel, omdat op dat niveau de interpretatie van zoiets als types vaak nog in ontwikkeling is (voorbeeld: het schrijven van het run-time systeem voor een programmeertaal, die wel types kent). Bij het schrijven van applicaties levert dit in de praktijk ook niet zoveel problemen op, omdat de afzonderlijke routines over het algemeen klein zijn en gemakkelijk één voor één kunnen worden uitgetest.

1.6. Lexicale conventies in FORTH

Een opvallend punt in FORTH is de lexicale conventie. FORTH programmatekst is opgebouwd uit *woorden*. Twee woorden zijn van elkaar gescheiden door spaties. Elk printable ASCII character behalve de spatie mag als onderdeel van een woord gebruikt worden. Het resultaat hiervan is dat ook typische operatornamen als '+' geen bijzondere symbolen zijn, wat de uitbreidbaarheid van de taal zeer ten goede komt. Ook vergroot het de vrijheid bij het kiezen van namen in een programma. Deze vrijheid levert naar mijn ervaring een niet te onderschatten bijdrage aan de arbeidsvreugde van een programmeur.

Naast de postfixnotatie is de lexicale conventie het meest verantwoordelijk voor het typische uiterlijk van FORTH programma's. Het vergroot de leesbaarheid voor mensen, die FORTH enigzins gewend zijn. Veel andere programmeertalen zien echter een groot verschil tussen alfanumerieke en niet-alfanumerieke characters. Alfanumerieke characters worden gebruikt in 'identifiers' en 'numbers'. De andere characters, zoals ';' en '+' betekenen iets bijzonders en zijn vaak juist scheiders van symbolen die uit alfanumerieke characters bestaan. Programmeurs worden hierdoor geconditioneerd; zij verdelen een programmatekst op het eerste gezicht al in identifiers, numbers en 'de rest'. Een FORTH programma is dan een klap in het gezicht.

2. FORTH ALS OMGEVING

In dit deel wordt bekeken wat voor voordelen FORTH te bieden heeft als basis voor een programmeeromgeving. We kunnen daarbij drie niveau's onderscheiden:

(1) de virtuele FORTH machine,

(2) de programmeertaal FORTH en

(3) de op FORTH gebouwde programmeer- en gebruiksomgeving.

Deze gezichten vormen de onderscheidbare niveau's in een FORTH systeem. Eventueel kan de omgeving waarbinnen FORTH draait nog als niveau 0 daaraan worden toegevoegd. Het belangrijkste voordeel van FORTH is, dat vooral tussen niveau 2, de taal, en niveau 3, de omgeving, geen zichtbare scheiding ligt. Ook niveau 1, de machine, ligt zeer dicht bij de beide andere. Tenslotte is ook de afstand tussen FORTH en de omgeving waarbinnen het draait minimaal. Op elk van de 3 genoemde niveau's in FORTH wordt in dit hoodstuk wat dieper ingegaan.

2.1. FORTH als machine

Een goede manier om tegen FORTH aan te kijken is als virtuele machine, vergelijk P-code of EM1. Het navolgende beschrijft hoe deze machine is opgebouwd rond een threaded code interpreter. Details zijn zorgvuldig weggelaten.

Op het laagste niveau bestaat FORTH uit een aantal in assemblertaal geschreven routines. De adressen van deze routines, op een rij gezet, vormen een soort code; de basis instructieset van de 'FORTH machine'. Het executeren van deze code bestaat uit het na elkaar aanroepen van deze adressen.

FORTH heeft een program counter, de 'instruction pointer'. Elke in assembler geschreven routine eindigt met een sprong naar de interne interpreter, die het volgende routine adres ophaalt, de instruction pointer verhoogt en naar het opgehaalde adres toespringt. Voor deze interpretatieslag kan op een PDP11 of op een 6809 met één enkele instructie worden volstaan, in plaats van een sprong naar de interne interpreter.

Met behulp van de zojuist beschreven virtuele FORTH machinecode kunnen opnieuw routines worden gemaakt. Deze routines worden wel 'high level' routines genoemd, in tegenstelling tot de 'low level' in assemblertaal geschreven routines. High level routines bestaan uit een lijst van adressen, die zowel van low level als van high level routines kunnen zijn.

De vraag die hier gesteld kan worden is hoe de FORTH interpreter het verschil tussen een high level en een low level routine ziet. Hiervoor zijn verschillende mogelijkheden. De eenvoudigste oplossing is dat de routine het zelf weet. Elke high level routine begint met een instructie (meestal een subroutine-aanroep), die er voor zorgt dat de interpretatie van FORTH machinecode begonnen wordt, en dat de waarde van de instruction pointer wordt bewaard. Het laatste woord is een returninstructie voor de FORTH machine.

Een nieuwe implementatie van FORTH bestaat uit het schrijven van een relatief kleine kern in assemblertaal. De rest van FORTH is op deze kern gebaseerd. Er zijn FORTH systemen die, inklusief disksysteem, editor, en assembler, niet groter zijn dan 6k bytes, waarvan minder dan 2k in assemblertaal. Meestal wordt om snelheidsredenen iets meer van het systeem in assemblertaal uitgeschreven, bijvoorbeeld ongeveer 4k. Ook zijn zaken als de editor en de assembler over het algemeen aparte produkten.

De geringe omvang van FORTH maakt de taal met een erop gebaseerde programmeeromgeving uitermate geschikt voor personal computers en zelfs voor zeer kleine 'hand held' computers. Ook zijn er implementaties van FORTH op grotere machines, zowel stand alone als onder vreemde operating systems. In FORTH geschreven programma's zijn meestal zonder problemen overdraagbaar van de ene machine naar de andere. Uiteraard zijn in assemblertaal uitgeschreven onderdelen van een programma niet overdraagbaar tussen machines met verschillende processors. Het verdient daarom aanbeveling van elk FORTH programma in eerste instantie een zuivere high level versie te maken. Hiervan kunnen dan eventueel in een lokale versie tijdkritische 'innermost loops' in assemblertaal worden uitgeschreven. Dit is dan vrijwel altijd een eenvoudige routineklus.

Tot besluit van deze paragraaf nog twee punten, die bij het gebruik van de FORTH machine sterk opvallen.

Werken onder de FORTH machine schermt de programmeur, die dat niet wil, *niet* af van de onder FORTH liggende systeemniveau's. Hij kan bijvoorbeeld eigenschappen

van lokale hardware, die FORTH zelf niet gebruikt, toch volledig uitbuiten. Het is vanwege deze eigenschap en omdat FORTH applicaties kort en snel kunnen zijn, dat de taal in applicaties als spelmachines, procesbesturing en dergelijke eigenlijk geen concurrenten heeft.

De FORTH machine biedt een plezierig target voor compilers. Vooral het feit dat de 'machine' modificeerbaar is en het gemak waarmee die modificaties kunnen worden uitgevoerd zijn daarbij belangrijke voordelen.

2.2. FORTH als programmeertaal

Buiten de 'inner interpreter', de virtuele machine die de basis van FORTH vormt, kent FORTH ook een 'outer interpreter', die op een redelijk recht toe recht aan manier de instructies van de virtuele machine koppelt aan FORTH namen. We zullen verder spreken van *woorden* van de FORTH taal. De woorden bevinden zich, met hun naam en hun code, in een interne datastructuur, de eerder genoemde *dictionary*. Bovenop de daar al aanwezige woorden kunnen nieuwe woorden gemaakt worden, met een routine body die bestaat uit aanroepen van al eerder gedefinieerde woorden.

De outer interpreter werkt nu als volgt:

(1) Vraag een regel van de gebruiker (of haal hem uit een source file).

(2) Zoek de - door spaties van elkaar gescheiden - woorden één voor één in de dictionary op.

(3) Als we

  (a) in de 'executie mode' zitten, executeer het woord.

  (b) in de 'compilatie mode' zitten en het woord is niet 'immediate', zet het executie-adres van het woord in de dictionary.

  (c) in de 'compilatie mode' zitten en het woord is wel 'immediate', executeer het woord als in (a).

  (d) het woord niet kunnen vinden probeer het te interpreteren als een getal; lukt dit, zet dan het getal op de stack of compileer een routine die het getal te zijner tijd op de stack zet, al naar gelang we in executie of compilatie mode zitten.

  (e) het woord niet kunnen vinden en het was ook geen getal, geef dan een foutmelding..

Dit algoritme, dat de hele outer interpreter van FORTH beschrijft, is erg eenvoudig, zodat het in een paar regels geïmplementeerd kan worden. Dit heeft twee voordelen. Ten eerste kan de FORTH interpreter altijd in geheugen aanwezig zijn. FORTH is bij wijze van spreken alomtegenwoordig en kan dus ook altijd als user interface van een applicatie dienen.

Ten tweede heeft een programmeur de mogelijkheid om de standaard outer interpreter te vervangen door een eigen versie. Een vergaand voorbeeld van een dergelijke aanpak is een op FORTH gebaseerde PASCAL compiler.

Het programmeren in FORTH bestaat uit het uitbreiden van de taal in een specifieke richting. Elke nieuwe routine voegt als het ware een woord aan de taal (of zo u wilt een instructie aan de machine) toe. Immediate woorden kunnen worden gebruikt om de control flow structuur van de taal verder uit te breiden.

Jan Heering pleit voor een 'gesloten' programmeertaal, een taal waarmee alle functies van het systeem aan te spreken zijn [3]. Binnen een FORTH systeem zijn de FORTH interpreter/compiler zelf, het operating system, en alle gereedschappen, zoals de assembler, de debug tools en de editor, geschreven in FORTH. Alle facetten van het systeem vormen daarmee een integraal onderdeel van FORTH. Elke systeemfunctie is een FORTH routine, die als onderdeel van andere FORTH routines kan worden gebruikt. FORTH kan daarom worden beschouwd als een gesloten programmeertaal.

Een laatste aspect van FORTH dat nog even aandacht verdient is het 'leeneffect'. Alle constructies, die FORTH of een van de applicaties in zich bergt, zijn afzonderlijk te gebruiken als FORTH woorden. Behalve voordelen bij debugging, geeft dit de programmeur een tool-kit, waar hij/zij naar believen uit kan putten. Dit mechanisme draagt ertoe bij dat, ondanks de groei die een FORTH omgeving in de loop der jaren doormaakt, toch een zekere consistentie gehandhaafd blijft.

2.3. FORTH als omgeving

Een programmeeromgeving presenteert zich aan zijn gebruikers meestal als een verzameling gereedschappen, zoals een editor, een file systeem, een debugger, een compiler, etc. Elk gereedschap heeft zijn eigen aanspreektaal. De editor heeft edit commando's, het operating system een commandotaal, een compiler de hogere programmeertaal en veelal een optie-mechanisme. Uiteraard verschillen deze talen in de verzameling commando's die ze accepteren. Minder logisch echter, maar wel historisch zo gegroeid, is het dat ook de structuur van elk der talen meestal sterk verschilt en dat gemeenschappelijke elementen vaak niet als zodanig herkenbaar zijn. Bovendien zijn de aanspreektalen van veel gereedschappen, zoals editors of debuggers, geen programmeertalen. Ze accepteren alleen een verzameling van basiscommando's, toevoegen van nieuwe faciliteiten is niet of nauwelijks mogelijk.

Een dergelijke wildgroei is in een FORTH omgeving veel onwaarschijnlijker. In FORTH geïmplementeerde applicaties bestaan uit een verzameling FORTH woorden, voor elke 'feature' minimaal één. Men kan als programmeur meestal volstaan met het naar buiten brengen en documenteren van deze woorden. De gebruiker hoeft dan slechts deze nieuwe woorden te leren. De manier waarop de woorden moeten worden gebruikt is de gebruiker al bekend van andere FORTH applicaties.

Soms vereist de aard van de applicatie wat meer werk om FORTH als aanspreektaal bruikbaar te maken, maar bijna altijd is dit voor een programmeur toch eenvoudiger dan zelf een gebruikersinterface te moeten maken.

Laat ik een paar voorbeelden geven. De ontwerper van een tekstopmaker wil met 'punt'-commando's gaan werken, dat wil zeggen de tekst bevat ook regels met opmaakcommando's, die met een punt beginnen. Hij implementeert de opmaakcommando's als FORTH woorden. Verder biedt hij tijdens het opmaken elke regel die met een punt begint aan de FORTH interpreter aan, die de op die regel aanwezige woorden opzoekt en aanroept.

De gebruiker heeft nu een herkenbare interface en is bovendien in staat met behulp van FORTH de opmaker verder uit te breiden. Een toepassing hiervan zou kunnen zijn het in-line aanroepen van een wiskundige typesetter of een tabel-opmaker. Ook denk ik aan mogelijkheden als het aanroepen van plotroutines, invoer vragen van de gebruiker, raadplegen van databestanden, verrichten van berekeningen, etc.

In ieder geval heeft de maker van de opmaker de mogelijkheid een goed leesbare interface voor de gebruiker te maken, wat normaliter bij opmakers nog weleens een probleem pleegt te zijn.

Een regel-editor kan de commando's van zijn gebruiker direct met FORTH interpreteren. Een scherm-editor kan de ingetikte controle-characters als index in een tabel gebruiken, waar zich de adressen van FORTH routines bevinden.

De enigzins gevorderde gebruiker of een systeemprogrammeur heeft dan de mogelijkheid om aan de bestaande editor allerlei uitbreidingen te maken, zoals het op een rijtje zetten van bestaande commando's, maar ook meer op een specifieke omgeving toegesneden acties zoals boekhoudkundige saldoberekeningen, etc. En taalimplementators kunnen taalondersteuningen, zoals edit-time syntaxchecking en prettyprinting, aan een al aanwezige editor koppelen.

Compilers voor andere programmeertalen kunnen zich presenteren als een verzameling FORTH woorden. Voor 'normaal' gebruik kan volstaan worden met het woord dat een programma vertaalt. Andere woorden kunnen opties wijzigen of het compilatieproces besturen (conditionele compilatie, etc.).

Echter ook de diepere lagen van de compiler zelf kunnen als FORTH woorden naar buiten worden gebracht. Voorbeelden van mogelijke acties van dergelijke woorden zijn: herken en vertaal een expressie of een statement, genereer code, etc. Deze verzameling woorden kan worden gebruikt om de compiler in een bepaalde richting uit te breiden door er nieuwe woorden aan toe te voegen. De compiler moet in staat zijn deze nieuwe woorden te herkennen en tijdens het vertaalproces aan te roepen. Voorwaarde is, dat het nieuwe woord voldoet aan de syntax van een symbool van de programmeertaal, die wordt vertaald. Het aangeroepen woord kan dan tijdelijk de compilatie overnemen, gebruik makend van routines uit de compiler om standaard substructuren van de taal te verwerken. Bij dit alles ga ik dan uit van een recursive descent compiler. Mogelijke redenen om dit mechanisme te gebruiken zijn speciale codegeneratie, taal switches - naar assemblertaal, FORTH of zelfs een andere compiler - mogelijk maken of toevoegen van concepten die binnen de taal normaal niet implementeerbaar zijn. Op deze manier kan een algemene taal zoals PASCAL aan een specifiek applicatieterrein worden aangepast. Ook kan een gebruiker nieuwe taalconcepten op hun bruikbaarheid toetsen, door ze aan een al bestaande compiler toe te voegen.

Zoals uit de voorgaande opsomming al blijkt lopen implementatie- en applicatietaal bij FORTH door elkaar. Het is hierbij vooral een belangrijk punt dat de FORTH interpreter zo klein is dat hij altijd aanwezig kan zijn en dus door applicaties ook run-time gebruikt kan worden. Het gevolg hiervan is dat alle FORTH applicaties dezelfde taal spreken - alleen de woordenschat verschilt. Implementators ontlenen aan FORTH hun gebruikersinterface en andere geschikte subroutines, en de latere programmeurs breiden liever de al bestaande applicaties uit in plaats van nieuwe te maken.

3. SAMENVATTING

FORTH is op drie niveau's te beschouwen:

(1) FORTH is een virtuele machine. Deze virtuele machine is eenvoudig te implementeren op bestaande machines. FORTH code bestaat uit een lijst executie-adressen van routines, die door een interne interpreter één voor één worden uitgevoerd. De routines

kunnen zowel in FORTH zelf als in assemblertaal zijn geschreven.

(2) FORTH is een programmeertaal. De interne code kan worden aangemaakt en ook geëxecuteerd door een externe interpreter, die zich voornamelijk bezighoudt met het opzoeken van namen van routines. De bij de naam behorende routine wordt aangeroepen of zijn executie-adres wordt gecompileerd, al naar gelang FORTH zich in de executie- dan wel in de compilatiemode bevindt. FORTH ondersteunt compile-time executie van immediate routines - bijvoorbeeld voor control flow handling - en kent machtige mechanismen om data handling uit te drukken. FORTH is een hoog-niveau, incrementeel werkende, interactieve programmeertaal.

(3) FORTH is een programmeeromgeving. De FORTH taal kan worden uitgebreid naar een programmeeromgeving. Het is daarbij niet nodig om volledig nieuwe niveau's in te voeren. Dit bevordert een uniforme interface voor de gebruiker. Het moedigt bovendien het lenen van al aanwezige routines/concepten aan. In FORTH geschreven applicaties kunnen dan ook vrijwel altijd naderhand worden uitgebreid, omdat ze een integraal onderdeel van de taal vormen. De noodzaak compleet nieuwe versies van bestaande applicaties te maken kan daardoor meestal worden vermeden.

Mijn dank gaat uit naar mijn collega's van Forth Team Utrecht, in het bijzonder Dick Wesseling en Wim van Velthoven, voor hun hulp bij het voorbereiden van deze bijdrage.

LITERATUUR

[1] Moore, C.H., "FORTH: a new way to program a computer", *Astronomy and Astrophysics Supplement*, 1974, 15, pp. 497-511.

[2] Moore, C.H., "The evolution of FORTH, an unusual language", *BYTE*, Augustus 1980, pp. 76-90.

[3] Heering, J., "Taaldefinities als kern voor een programmeeromgeving", deze syllabus, pp. 69-81.

[4] Brodie, L., *Starting FORTH*, Prentice Hall, 1981.

[5] James, J.S., "FORTH for micro computers", *Dr. Dobb's Journal of Computer Calisthenics & Orthodontia*, Mei 1978, en *SIGPLAN Notices*, **13**(1978), 10, pp. 33-39.

BYTE, augustus 1980, en Dr. Dobb's Journal, september 1981, zijn geheel aan FORTH gewijd.

TAALDEFINITIES ALS KERN VOOR EEN PROGRAMMEEROMGEVING

Jan Heering
Mathematisch Centrum
Amsterdam

SAMENVATTING

De programmeur is ermee gediend als de verschillende componenten van een programmeeromgeving zoveel mogelijk dezelfde taal gebruiken. Integratie van bijvoorbeeld programmeertaal en commandotaal is in verregaande mate mogelijk en dit geeft een zeer homogene omgeving. De eis, dat ook nieuwe, door de gebruiker gedefinieerde, (applicatie)talen op uniforme wijze in het systeem moeten passen, leidt tot een programmeeromgeving waarin taaldefinities centraal staan.

1. INLEIDING

1.1. Integratie van modes

Een programmeeromgeving is een verzameling gereedschap voor het maken en bewerken van programma's. Het ligt - zeker in eerste instantie - voor de hand een dergelijke omgeving modulair op te bouwen en elk stuk gereedschap zoveel mogelijk als een zelfstandige eenheid te ontwikkelen. Dit leidt tot omgevingen met vele 'modes' (command mode, edit mode, debugging mode, etc.), waarin elke mode zijn eigen toepassingsgebied en zijn eigen taal heeft.

In de praktijk blijkt echter, dat de afbakening van de verschillende toepassingsgebieden die aan de modularisatie ten grondslag ligt geen steek houdt. Omdat de ontwerpers van afzonderlijke modes echter grotendeels hun eigen gang kunnen gaan, worden vele concepten en constructies, die qua abstracte semantiek nauw verwant of zelfs identiek zijn, in de bijbehorende talen in zodanig verschillende vorm aangeboden, dat het de gebruiker vrijwel onmogelijk wordt gemaakt de overeenkomsten te zien. Zo speelt het begrip *variabele* een vooraanstaande rol in programmeertalen, maar het treedt - in andere vermommingen - ook op in commandotalen (job control languages) en in edit- en uitlijntalen. Dat dit de gebruikersvriendelijkheid negatief beïnvloedt behoeft wel geen betoog.

Als de overlap tussen de verschillende toepassingsgebieden erg groot wordt treedt nog een ander verschijnsel op: de bijbehorende talen gaan met elkaar concurreren. Dit geldt in het bijzonder voor commandotalen en programmeertalen. Op vele installaties wordt al de helft van alle procedures in de commandotaal geschreven. Dit is des te ernstiger, omdat ook de verst ontwikkelde commandotalen geen behoorlijk gedefinieerde semantiek bezitten. Bovendien zijn in commandotalen geschreven procedures volstrekt niet overdraagbaar.

Hoe heeft deze schrikbarende ontwikkeling kunnen plaatsvinden? De belangrijkste oorzaak lijkt wel te zijn, dat de meeste programmeertalen *onvolledig* zijn in die zin, dat

ze essentieel afhankelijk zijn van een krachtige, maar verder grotendeels ongespecificeerde, 'buitenwereld'. Vanzelfsprekend is er geen natuurwet, die decreteert dat programmeeertalen onvolledig moeten zijn en er zijn dan ook uitzonderingen, waarvan LISP wel de belangrijkste is. Niettemin is het een feit, dat programmeertalen over het algemeen onvoldoende uitdrukkingsmacht bezitten om het beheer van permanente gegevens (files) of het creëren van nieuwe programma's te beschrijven. Zo zijn bijvoorbeeld functies, die een file opruimen of een nieuw file directory aanmaken, in de meeste programmeertalen eenvoudig niet gedefinieerd. Als ze wel beschikbaar zijn, dan alleen via de aanroep van een systeemprocedure, waarvan de body niet in de programmeertaal zelf geschreven kan worden en die dus systeem-afhankelijk is. Ook het creëren van nieuwe programma's (procedures) - een functie die essentieel is in het programma-ontwikkelingsproces - kan in de meeste programmeertalen niet beschreven worden. Commandotalen hebben de vereiste uitdrukkingsmacht wel en zijn in die zin *volledig*. Bovendien zijn van oorsprong typische programmeertaalconstructies als **if**- en **for**-statements zo langzamerhand ook in commandotalen doorgedrongen, zodat deze hun achterstand in dit opzicht voor een aanzienlijk deel hebben ingelopen.

Nu de diagnose eenmaal gesteld is, ligt de remedie - althans in principe - voor de hand: de modulaire opbouw dient plaats te maken voor een geïntegreerde. In hoofdstuk 2 zal ik uiteenzetten welke consequenties een dergelijke integratie en, meer in het bijzonder, de integratie van commando- en programmeertaal heeft. Een interessant punt daarbij is, dat een omgeving op basis van een geïntegreerde commando/programmeertaal niet alleen een veel homogenere gebruikersinterface heeft dan een niet geïntegreerde omgeving, maar dat bovendien de taal zelf aan striktere eisen moet voldoen dan een 'gewone' programmeertaal. Dientengevolge heeft de taalontwerper op essentiële punten minder vrijheid. Alhoewel dit op het eerste gezicht een nadeel mag lijken, is het in de praktijk een voordeel, omdat er een groot gebrek bestaat aan deugdelijke criteria op grond waarvan beslist kan worden welke constructies wel en niet in een taal moeten worden opgenomen.

Hiermee is het probleem van - wat men zou kunnen noemen - de *initiële consistentie* van programmeeromgevingen goeddeels opgelost. Toch is dit slechts een gedeelte (en misschien niet eens het belangrijkste gedeelte) van de problematiek.

1.2. Maatregelen om geleidelijke desintegratie tegen te gaan

Een programmeeromgeving is geen statisch geheel, maar evolueert in de tijd. De mate van consistentie van het systeem ontwikkelt zich dus eveneens. In de praktijk betekent dit vrijwel altijd, dat, naarmate de modificaties en uitbreidingen zich opstapelen, het geheel langzaam maar zeker aan consistentie verliest en in de meest letterlijke zin van het woord desintegreert.

Het voortschrijdende verval openbaart zich op twee manieren. Enerzijds wordt het systeem, naarmate het aan doorzichtigheid inboet, steeds minder gebruikersvriendelijk, anderzijds wordt het tegelijkertijd steeds moeilijker te repareren en uit te breiden. Terwijl het groter en groter wordt, raken de onderdelen geleidelijk aan minder goed op elkaar afgestemd. Ook een systeem met een hoge graad van initiële consistentie is voor een dergelijke ontwikkeling niet gevrijwaard, al heeft het een (min of meer constante) voorsprong.

De oorzaak van het verval is, dat modificaties en uitbreidingen onvoldoende zorgvuldig worden aangebracht en te weinig met het reeds bestaande worden geïntegreerd. Dit betekent niet, dat het simpelweg betrachten van groter zorgvuldigheid de oplossing zou

zijn, want het te modificeren of uit te breiden geheel is veelal zo complex, dat optimale reparatie en integratie eenvoudig onmogelijk zijn zonder grotendeels van voren af aan te beginnen.

Ongrijpbaar, maar onmiskenbaar, speelt in dit soort beschouwingen op de achtergrond het uit thermodynamica en informatietheorie afkomstige begrip *entropie* een rol. De gedachte dringt zich op, dat entropie een potentieel belangrijke software engineering grootheid is, al valt er in de context waarin het begrip hier naar voren komt nog geen precieze betekenis aan te geven. Wie hier dieper op in wil gaan, kan ik aanraden de *informatietheoretische complexiteitstheorie* van Chaitin en Kolmogorov te bestuderen (zie [1] en de inleiding tot deze syllabus). Omdat programmeeromgevingen geen gesloten systemen vormen - zij evolueren immers niet uit zichzelf, maar slechts als gevolg van menselijk ingrijpen - is er geen reden om aan te nemen, dat hun geleidelijke desintegratie op grond van een of andere gegeneraliseerde entropiewet onvermijdelijk zou zijn. In de praktijk blijkt echter een onverwacht grote inspanning noodzakelijk om aan entropietoename te ontsnappen. Een interessant overzicht van de wetten, die - bij de huidige stand van management en software engineering - de evolutie van grote programma's beheersen, is te vinden in [2].

Is het mogelijk de techniek van de integratie van modes, die in eerste instantie leidt tot een omgeving met hoge initiële consistentie, zó in een systeem te verwerken, dat ook in een later stadium door de gebruiker toegevoegde modes gemakkelijker geïntegreerd kunnen worden? Dat zou immers betekenen, dat de oorspronkelijke consistentie langer behouden kan blijven waardoor het minder vaak noodzakelijk is van voren af aan te beginnen. In hoofdstuk 3 zal ik uiteenzetten, dat dit uitgangspunt op natuurlijke wijze leidt tot een omgeving, die op (operationele) taaldefinities gebaseerd is. Alle talen en taaltjes in het systeem, zowel de oorspronkelijk aanwezige als de later door de gebruiker toegevoegde, worden op gelijke wijze behandeld. Daarbij moet er zo mogelijk voor gezorgd worden, dat nieuwe talen van reeds bestaande kunnen *lenen*. Enerzijds kan men zo bereiken, dat constructies uit de ene mode ook in de andere beschikbaar zijn, anderzijds helpt het te voorkomen dat constructies met dezelfde abstracte semantiek zich onder allerlei verschillende vermommingen aan de gebruiker presenteren. Het lijkt immers redelijk aan te nemen, dat de gebruiker een constructie, die er al is, niet zo snel zelf opnieuw zal definiëren.

2. INTEGRATIE VAN COMMANDO- EN PROGRAMMEERTAAL

2.1. Het dynamische karakter van commandotalen

Alvorens in te gaan op de eigenschappen van een omgeving, die op een geïntegreerde commando/programmeertaal gebaseerd is, is het nuttig commandotalen eerst eens wat nauwkeuriger te bekijken.

Een commandotaal is de gebruikersinterface van een bedrijfssysteem en stelt als zodanig de gebruiker in staat de functies van het systeem aan te spreken. Zoals reeds in de inleiding uiteengezet, bestrijken commandotalen daarmee een gebied - namelijk dat van file- en procesbeheer - waar programmeertalen grotendeels machteloos zijn. Commandotalen zelf zijn in zichzelf gesloten en niet op hun beurt weer afhankelijk van nóg machtiger talen. Toch is er geen *fundamenteel* verschil tussen beide soorten talen en er is weinig bezwaar tegen commandotalen te beschouwen als krachtige (maar, in hun huidige vorm, nog zeer chaotische) programmeertalen, die op complete files

opereren in plaats van op simpele integers, zoals programmeertalen gewoonlijk doen.

Omdat commandotalen volledig zijn, hebben ze een veel dynamischer karakter dan programmeertalen. Zoals reeds in de vorige paragraaf opgemerkt, is het in de meeste programmeertalen niet mogelijk om het aanmaken van nieuwe programma's (procedures) uit te drukken. Zonder een dergelijk mechanisme kan een systeem echter niet groeien. Wat dat betreft zijn programmeertalen dus geheel afhankelijk van commandotalen. Evenmin staan de meeste programmeertalen toe, dat een programma dynamisch variabelen aanmaakt, met andere woorden, er kunnen tijdens uitvoering van een programma geen nieuwe variabelen gecreëerd worden. Op commandoniveau, waar de rol van variabelen vervuld wordt door files (een file is immers niets anders dan een permanente variabele*), zou een dergelijke beperking tot gevolg hebben, dat een filenaam nooit het resultaat van een berekening kan zijn. Dat is niet acceptabel. Het programmatisch creëren van filenamen is op commandoniveau heel gewoon. In het algemeen geldt, dat naarmate de levensduur van variabelen langer is, er meer waarde aan hun namen gehecht wordt. Het opzoeken van alle filenamen, die aan een bepaald patroon voldoen (bijvoorbeeld met 'int' beginnen), is op commandoniveau zonder probleem uit te drukken, maar patroonherkenning op de namen van programmavariabelen is in bestaande programmeertalen niet mogelijk.

Het dynamische karakter van commandotalen komt ook daarin tot uiting, dat de grootte van files dynamisch kan variëren en niet van te voren hoeft te worden vastgelegd. Dit is geen onverantwoorde luxe, maar noodzaak. De lengte van vele files is immers niet van te voren aan te geven (denk bijvoorbeeld aan een tekst, die met een editor wordt aangemaakt, of aan de symboltable van een compiler). Programmeertalen zijn in dit opzicht meestal veel minder liberaal, zoals elke gebruiker van PASCAL zal kunnen beamen. Als de nood aan de man komt moeten zij een beroep doen op het filesysteem, maar dat heeft als nadeel, dat de communicatie met het filesysteem vanuit programma's moeizaam gaat. Er komen allerlei ingewikkelde conversies en in- en uitvoeroperaties aan te pas, want de types van files en variabelen zijn volstrekt niet op elkaar afgestemd. Dezelfde asymmetrie treedt overigens eigenaardigerwijze ook in commandotalen op. Vele commandotalen kennen namelijk naast files ook nog lokale variabelen, die andere eigenschappen hebben.

Het volgende voorbeeld illustreert hoe deze asymmetrie in een eenvoudig geval tot uitdrukking komt in een van de krachtigste op dit moment bestaande commandotalen, de zg. *shell* van het UNIX bedrijfssysteem [3]. Beschouw het volgende programmaatje in pseudo-ALGOL:

```
permanent x; local y
x := 'abc'
y := x
```

x en y zijn respectievelijk een permanente en een lokale variabele, terwijl 'abc' een tekst is. Na afloop hebben zowel x als y de waarde 'abc' en het enige verschil is, dat x in principe een langere levensduur heeft dan y. Het shell equivalent ziet er wat minder gepolijst uit:

---

* Ik gebruik het woord *file* zowel in de betekenis van *permanente waarde* als van *permanente variabele*. In het algemeen zal uit de context duidelijk zijn welke van beide betekenissen bedoeld wordt.

```
echo abc >x
y='cat x'
```

Ik zal niet proberen precies uit te leggen wat deze twee regels betekenen, maar waar het op neerkomt is, dat (in UNIX terminologie) de stringwaarde 'abc' naar file *x* geschreven wordt, terwijl vervolgens de waarde van *x* aan shell variabele *y* geassigneerd wordt. Beschouw nu hetzelfde programmaatje, maar met de scopes van *x* en *y* verwisseld:

```
local x; permanent y
x := 'abc'
y := x
```

Hoe groot de asymmetrie tussen lokale en permanente variabelen in de shell is blijkt uit het feit, dat het shell equivalent hiervan er totaal anders uitziet dan in het eerste geval:

```
x=abc
echo $x >y
```

Het is mischien niet overbodig op te merken, dat voor dit voorbeeld geen uitzonderlijk geval geselecteerd is. De shell behandelt permanente en lokale variabelen altijd verschillend.

2.2. Schets van een geïntegreerde commando/programmeertaal

Ik zal nu in het kort uiteenzetten hoe de integratie van commando- en programmeertaal in zijn werk gaat. Meer details dan in dit bestek gegeven kunnen worden zijn te vinden in [4] en [5].

In de vorige paragraaf is benadrukt, dat er qua abstracte semantiek geen verschil is tussen files en programmavariabelen. Evenmin is er een wezenlijk onderscheid tussen programma's en procedures. Het ligt voor de hand dit ook in de bijbehorende concrete semantiek tot uitdrukking te brengen, dat wil zeggen

(1) het onderscheid tussen programma's en procedures wat betreft aanroep en parametermechanisme op te heffen en

(2) het onderscheid tussen files en variabelen wat betreft type en naamgeving op te heffen.

Wat betreft het eerste punt - het commandoniveau van het geïntegreerde systeem correspondeert met interactief programmeren op het hoogste procedureniveau. Het equivalent van een conventioneel programma is een procedure, die vanaf het hoogste niveau wordt aangeroepen. Volledige integratie van commando- en programmeertaal impliceert, dat er qua mogelijkheden geen verschil mag zijn tussen het commandoniveau en diepere procedureniveau's. Wat betreft het tweede punt - de eerste stap bij het unificeren van files en variabelen is om alle verschillen tussen filetypes en 'gewone' types af te schaffen. Dat betekent bijvoorbeeld, dat een procedure die een integer resultaat heeft berekend dat zo maar, zonder enige conversie- of uitvoeroperatie, aan een permanente variabele kan assigneren. Of stel, dat iemand on-line een telefoonboek bijhoudt, dan zijn in een conventioneel systeem de volgende stappen vereist als een abonnee een ander nummer krijgt:

(1) Activeer telefoonboek-editor

(2) Breng mutatie aan.

(3) Keer terug naar command mode.

In een geïntegreerd systeem kan het telefoonboek gerepresenteerd worden met behulp van een geschikt type (waarover dadelijk meer). Een goede keuze zou een associatieve tabel zijn, die namen met telefoonnummers associeert. Eén enkel commando volstaat nu om een abonnee een nieuw nummer te geven:

telefoonboek[abonnee] := nieuwnummer

Er komt geen speciale edit mode meer aan te pas.

Welke datatypes moet een geïntegreerde commando/programmeertaal hebben? Zowel 'kleine' types, zoals integers en korte teksten, als 'grote' types, zoals arrays, bomen, lange teksten en directories, moeten zonder moeite hanteerbaar zijn. Gezien deze veelheid van types, lijkt de enige redelijke oplossing de taal uit te rusten met de mogelijkheid *abstracte datatypes* te definiëren. Op basis van een voldoende krachtige verzameling basistypes (waaronder associatieve tabellen en dynamische arrays) moet de gebruiker in staat zijn de meeste types, die hij of zij nodig heeft, zelf te definiëren.

Nu er eenmaal een uniforme verzameling types is, zijn de meeste typeconversies en in- en uitvoeroperaties, die in het conventionele geval nodig zijn om de afstand tussen de types van files en variabelen te overbruggen, niet meer nodig. Dientengevolge hebben in een geïntegreerde omgeving zowel de basistypes als de door de gebruiker toegevoegde types een hoog rendement. (De ontwerpers van UNIX beschouwen het als een van de grote voordelen van hun systeem, dat het geen filetypes kent. Alle files onder UNIX zijn homogene strings van bytes zonder verdere structuur. Gezien het voorgaande moet ernstig betwijfeld worden of dit wel een voordeel is, al is het waar dat filetypes in conventionele systemen onhandelbaar zijn.)

De typeconsistentie van programma's wordt in het conventionele geval gecontroleerd door de compiler. Zelfs de linkage editor, die toch te beschouwen is als een uitgestelde fase van de compiler, laat meestal na te controleren of een procedure-aanroep en de bijbehorende procedure wel verenigbaar zijn wat betreft aantal en type van de parameters. Filetypes maken geen onderdeel uit van het typesysteem en worden niet gecontroleerd. Commandotalen kennen evenmin typechecking.

In een geïntegreerde omgeving daarentegen ligt het voor de hand typeconflicten in een zo vroeg mogelijk stadium te melden zonder willekeurig allerlei uitzonderingen te maken of beperkingen op te leggen. Dit houdt bijvoorbeeld in, dat ook permanente variabelen in het typechecking proces betrokken worden, dat verder na modificatie van een procedure zoveel mogelijk alle statisch niet meer consistente aanroepen gesignaleerd worden en dat na modificatie van een typedefinitie gecontroleerd wordt of de instanties van de oude definitie nog wel statisch consistent zijn met de nieuwe. Bij de implementatie hiervan kan - naar het zich laat aanzien - met vrucht gebruik gemaakt worden van *partiële evaluatie*. Zie daarvoor de bijdrage van Paul Klint in deze syllabus [6]. Ook interessant in dit verband is het B-systeem, dat in de bijdrage van Leo Geurts aan bod komt [7].

Een geïntegreerde commando/programmeertaal moet volledig zijn, dat wil zeggen:

(1) procedure- en typedefinities moeten in de taal zelf aangemaakt en gemanipuleerd kunnen worden en

(2) de permanente omgeving (het equivalent van het filesysteem) moet geheel vanuit de taal beheersbaar zijn.

Beide punten hebben interessante consequenties. Het 'geïntegreerde' equivalent van het aanmaken van een programma is het declareren van een procedure op het commandoniveau. Als de declaratie af is, wordt hij aan een naam gebonden, dat wil zeggen, dat hij geassigneerd wordt aan een procedurevariabele (het equivalent van een programmafile). Omdat, zoals eerder uiteengezet, diepere procedurele niveaus qua mogelijkheden niet verschillen van het commandoniveau, moet een geïntegreerde commando/programmeertaal noodzakelijk *geneste proceduredeclaraties*, lokale zowel als permanente *procedurevariabelen* en *procedureparameters* toelaten. Om dezelfde redenen is het noodzakelijk *typevariabelen* en *typeparameters* toe te laten. De waarde van een procedure- of typevariabele is de declaratie in de oorspronkelijke vorm, waarin hij aan de variabele geassigneerd is. Verborgen voor de gebruiker kan de waarde echter een geoptimaliseerde (gecompileerde) versie van de declaratie bevatten, die bij aanroep van de procedure of typedefinitie gebruikt wordt in plaats van de oorspronkelijke declaratie. Optimalisatie is dus in deze opzet geheel transparant.

Wat betreft punt (2) volsta ik met te vermelden, dat volledige beheersing van de permanente omgeving vereist, dat er een datatype *omgeving* of *bibliotheek* in de taal aanwezig is [5]. Een (procedure)bibliotheek is niet veel anders dan een verzameling (procedure)variabelen. Een bibliotheek kan zelf weer bibliotheekvariabelen bevatten, zodat willekeurige bibliotheekstructuren kunnen worden opgebouwd. Bibliotheken, maar dan opgevat als taaluitbreidingen, komen in het volgende hoofdstuk nogmaals ter sprake.

3. EEN OMGEVING OP BASIS VAN TAALDEFINITIES

Is het mogelijk een omgeving zo te structureren, dat de integratie van nieuwe subsystemen vergemakkelijkt wordt? Dat is het thema van dit hoofdstuk. Het uiteindelijke doel daarbij is, het in de inleiding gesignaleerde desintegratieproces een halt toe te roepen. Dit doel is nog ver verwijderd en de weg er heen - dat is in het verleden wel gebleken [8] - is moeilijk. Het volgende is dan ook niet meer dan een voorzichtige poging een stapje dichterbij te komen.

De in het vorige hoofdstuk geschetste geïntegreerde omgeving heeft weliswaar een veel grotere initiële consistentie dan conventionele omgevingen, maar de gebruiker, die een nieuwe mode met bijbehorende applicatietaal wil toevoegen, krijgt daarvoor van het systeem niet veel steun:

(A) Het feit, dat er in het systeem al minstens één parser aanwezig is (namelijk die voor de geïntegreerde commando/programmeertaal), betekent niet, dat het maken van de parser voor de toe te voegen taal gemakkelijker is.

(B) De beste manier om met het systeem in de commando/programmeertaal te communiceren is via een syntax-directed editor. Het feit, dat zo'n editor in het systeem aanwezig is, betekent echter niet dat het ook gemakkelijker is er een voor de nieuwe mode te maken.

(C) Semantische overlap van de toe te voegen taal met reeds in het systeem aanwezige talen (en in het bijzonder met de commando/programmeertaal) kan niet uitgebuit worden. Het *lenen* van reeds bestaande constructies is in het algemeen onmogelijk.

Enigszins anders geformuleerd: de *implementatie* van de geïntegreerde commando/programmeertaal is onvoldoende algemeen, niet toegankelijk genoeg en daardoor niet rendabel.

Figuur 1 toont de globale opzet van een omgeving, die - althans in principe - niet met deze bezwaren behept is. De nieuwe opzet is gebaseerd op *operationele (interpretationele) taaldefinities* en is te beschouwen als een generalisatie van de geïntegreerde omgeving uit het vorige hoofdstuk. De bezwaren (A) en (B) zijn ondervangen door de parser, evaluator en editor geheel los te maken van de taal, waarvoor ze oorspronkelijk bedoeld waren, en ze in gegeneraliseerde vorm aan de gebruiker ter beschikking te stellen ten behoeve van nieuw toe te voegen modes.

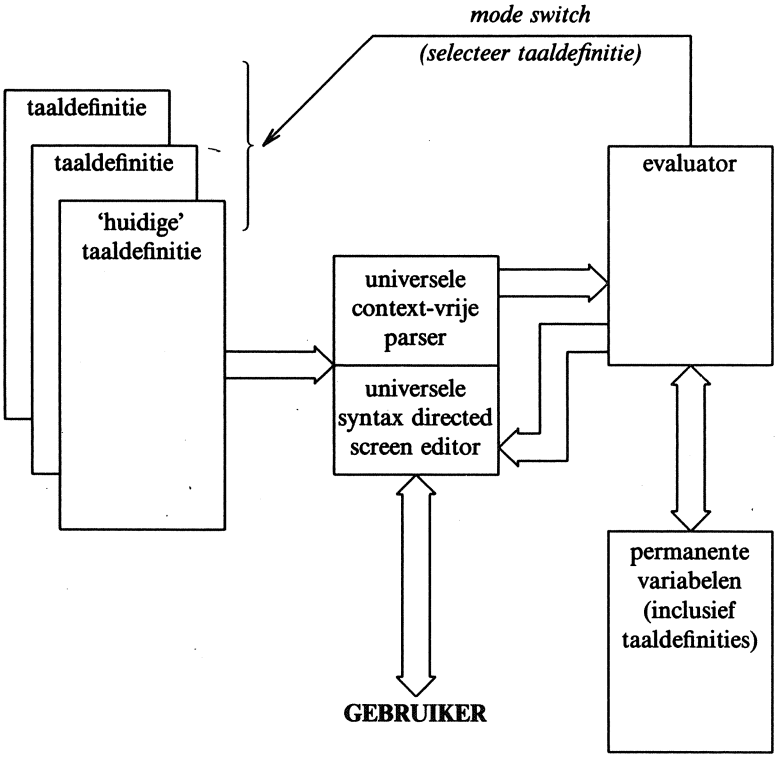


Fig. 1

Er is in het systeem tenminste één *oertaal*, die een krachtiger uitgave is van de geïntegreerde commando/programmeertaal uit het vorige hoofdstuk. Net als zijn voorganger is ook deze taal weer volledig. Dat betekent in dit geval, dat

(1) taaldefinities in de oertaal aangemaakt en gemanipuleerd kunnen worden en

(2) de oervorm van de permanente omgeving (dat is de permanente omgeving van het oorspronkelijke systeem) geheel vanuit de oertaal beheersbaar is.

In het bijzonder bevat de permanente omgeving dus objecten (waarden) van type *taaldefinitie*. Dat zijn de belangrijkste objecten in het systeem. Ze zijn voor alle duidelijkheid links in figuur 1 getekend, ondanks het feit dat daardoor hun relatie met de permanente variabelen rechts onder in de figuur niet goed meer tot uitdrukking komt.

Op elk moment communiceert de gebruiker met het systeem in één van de talen uit het talenbestand (in figuur 1 aangegeven als de 'huidige taaldefinitie'). Deze interactie vindt plaats via een door de taaldefinitie gestuurde screen editor, zodat foutencorrectie en prompting uniform zijn in alle modes. In geval van een mode switch wordt een andere 'huidige taal' geselecteerd. In figuur 1 is de selectie van een andere taal aangegeven met een pijl van de evaluator naar het talenbestand. De mode switch is een van de belangrijkste primitieven van de oertaal.

Een taaldefinitie bestaat uit drie componenten (figuur 2). Ten eerste de *syntaxregels*. Daarvoor kan de bekende BNF-notatie worden gebruikt, maar aangevuld met parameters voor het bij elke syntaxregel behorende *actie-* of *semantiekgedeelte* [9,10]. De parser/transducer geeft als onderdeel van het herkenningsproces tevens de parameters hun waarden en levert ze met het bijbehorende semantiekgedeelte af aan de evaluator. De semantiekgedeelten zijn uitgedrukt in een definitie- of implementatietaal T', waarnaar in de taaldefinitie verwezen wordt en die natuurlijk niet in alle gevallen dezelfde hoeft te zijn. In figuur 2 is die verwijzing aangegeven als een gestippelde pijl van het semantiekgedeelte van T naar de definitie van T'. T' is òf een oertaal òf een taal die tot het talenbestand behoort en waarvoor dus een definitie aanwezig is. In figuur 2 is aangenomen, dat de semantiek van T' zelf weer in een andere taal T'' is gedefinieerd.

Prettyprintaanwijzingen vormen de derde component in een taaldefinitie. Deze stellen de editor in staat teksten in de 'huidige taal' in een goede layout op het scherm te presenteren.

De directe voorlopers van de hier ingevoerde taaldefinities zijn ook al aanwezig in de geïntegreerde commando/programmeertaal uit het vorige hoofdstuk, namelijk in de vorm van bibliotheken en abstracte datatypes. Ik zal deze analogie wat verder uitwerken om aannemelijk te maken, dat de overgang naar de nieuwe opzet minder abrupt is dan hij misschien lijkt.

Een abstract datatype definieert een domein en een bijbehorend stel operaties. Het stelt deze operaties ter beschikking van de buitenwereld, maar houdt de implementatie ervan verborgen. Deze implementatie is te vergelijken met de semantiekcomponent in een operationele taaldefinitie. Wat is echter het equivalent van de syntactische component? Het antwoord is, dat de syntactische component in geval van abstracte types onderontwikkeld is. Terwijl de taalontwerpers voor waarden van een built-in type en de bijbehorende operaties allerlei sprekende notaties invoeren, geven zij de gebruiker tot nog toe niet of nauwelijks de mogelijkheid voor later toegevoegde types hetzelfde te doen. In dat geval beperkt de syntactische vrijheid zich goeddeels tot de 'punt-notatie'. Bijvoorbeeld:

```
type set-of-integers
begin
  export procedure add(n) integer n
        { ... }
  export procedure intersection(t) set-of-integers t
        { ... }
end set-of-integers

set-of-integers even, primes
comment let op de punt-notatie in de volgende drie regels!
primes.add(11)
even.add(4); even.add(6)
even-primes := primes.intersection(even)
```

(De punt-notatie is niet alleen star, maar bovendien *asymmetrisch*. In de laatste regel van het voorbeeld is *primes* het eerste - in de definitie impliciete - argument van *intersection* en *even* het tweede - in de definitie expliciete - argument. Vanuit wiskundig standpunt is er geen onderscheid tussen beide argumenten, in tegenstelling tot wat de notatie doet vermoeden. De diepere reden van de asymmetrie die - zoals uit de vorm van de definitie van *intersection* blijkt - niet alleen maar van syntactische aard is moet gezocht worden in het feit dat *intersection* eigenlijk niet van type *set-of-integers* is, maar van type *set-of-integers* $\times$ *set-of-integers* $\rightarrow$ *set-of-integers*. Dit punt is belangrijk, maar blijft hier verder buiten beschouwing.)

Het ligt voor de hand typedefinities uit te breiden met syntaxregels, die bepalen hoe de waarden van het nieuwe type en de bijbehorende operaties er *van buiten af* uitzien. Bovenstaand voorbeeld kan dan als volgt herschreven worden (vergelijk het LITHE systeem [11]):

```
type set-of-integers
begin
  export rule 'add' integer: n 'to' set-of-integers: s
              { ... }
  export rule 'intersection' 'of' set-of-integers: s 'and' set-of-integers: t
              { ... }
end set-of-integers

set-of-integers even, primes
add 11 to primes
add 4 to even; add 6 to even
even-primes := intersection of primes and even
```

Procedures en functies vormen het oudste taalextensiemechanisme. Een bibliotheek van matrixfuncties is op te vatten als een taal voor matrixmanipulatie en is in die zin te zien als taaldefinitie. Ook in dit geval is er een syntactische (en semantische) discrepantie tussen de built-in constructies van een programmeertaal en de extensies die zich met behulp van het proceduremechanisme laten realiseren. Bovendien wordt het datatype *bibliotheek* in het algemeen niet door de programmeertaal maar door het bedrijfssysteem ondersteund (zie vorige hoofdstuk).

Bezwaar (C) op p. 76 hield in, dat nieuwe talen onvoldoende kunnen profiteren van

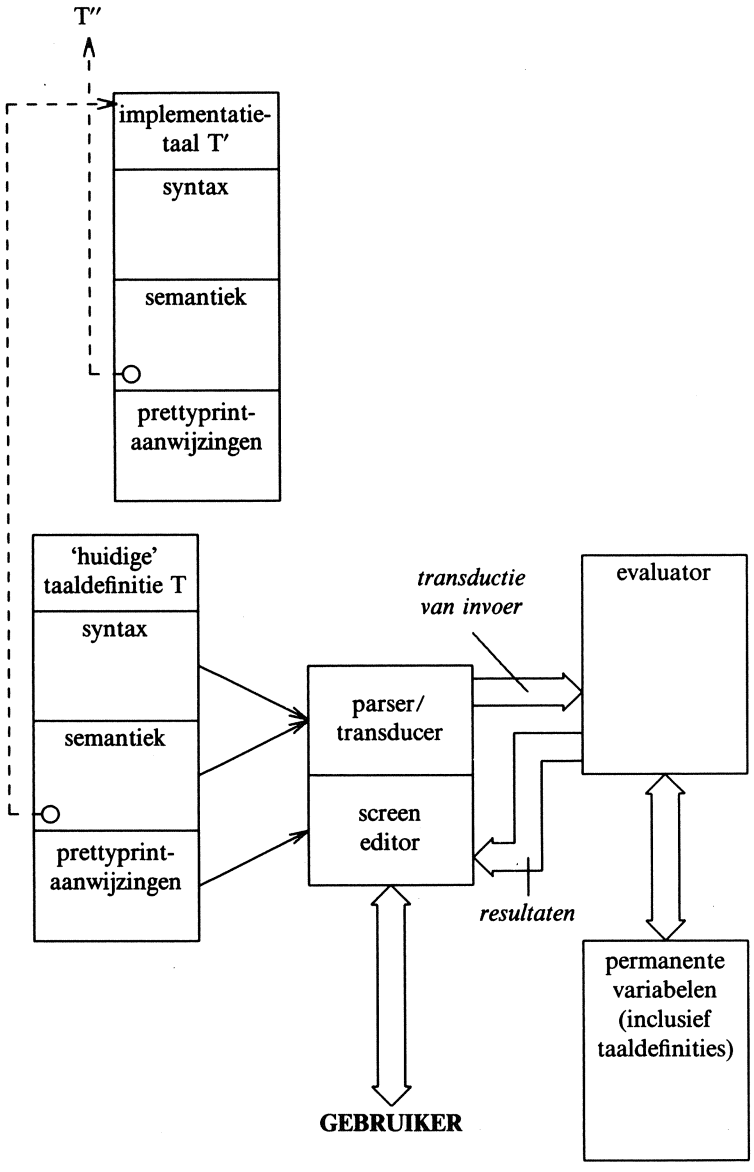

T''
implementatie-
taal T'
syntax
semantiek
prettyprint-
aanwijzingen
'huidige'
taaldefinitie T
syntax
semantiek
prettyprint-
aanwijzingen
transductie
van invoer
evaluator
parser/
transducer
screen
editor
resultaten
permanente
variabelen
(inclusief
taaldefinities)
GEBRUIKER

Fig. 2

in andere talen reeds bestaande constructies met als gevolg dat veel dubbel werk gedaan wordt en dat de verschillende talen (modes) meer dan noodzakelijk van elkaar gaan afwijken. De mogelijkheid tot het 'lenen' van constructies uit bestaande taaldefinities - te vergelijken met het lenen uit procedurebibliotheken en met het *subclass*-mechanisme in SMALLTALK - staat daarom bij het ontwerp van het type *taaldefinitie* centraal.

Het belangrijkste probleem met een omgeving als de hier geschetste is, dat hij moeilijk efficiënt te implementeren is. Ten gevolge van de algemene opzet ontaardt het systeem gemakkelijk in een onbruikbaar traag geheel. Een parser, die een willekeurige context-vrije grammatica plus te ontleden tekst als parameters heeft, is nu eenmaal aanzienlijk trager dan een gespecialiseerde parser die op één grammatica is toegespitst en hetzelfde geldt voor de evaluator en de editor. Gelukkig is hulp te verwachten van twee kanten: *partiële evaluatie* en *VLSI*. Naar verwachting - maar verder onderzoek zal dit moeten uitwijzen - zal het met behulp van partiële evaluatie mogelijk zijn de universele parser, evaluator en editor systematisch om te zetten naar efficiëntere versies, die op een gegeven taaldefinitie zijn toegespitst [6]. Het is in het bijzonder van belang de opstapeling van interpretatieniveau's zoveel mogelijk te voorkomen. Daarnaast biedt VLSI aantrekkelijke nieuwe mogelijkheden. Zo wordt bijvoorbeeld in [12] een in hoge mate parallelle VLSI-versie van het Cocke-Kasami-Younger algoritme beschreven, die een string ter lengte n in een willekeurige context-vrije taal herkent in tijd $2n$ onder gebruikmaking van $n(n+1)/2$ identieke 'cellen'. Een hele vooruitgang gezien het feit, dat de sequentiële versie er cn^3 tijd voor nodig heeft.

REFERENTIES

[1] Chaitin, G.J., "Algorithmic information theory", *IBM Journal of Research and Development*, **21**(1977), 4, pp. 350-359.

[2] Lehman, M.M., "Programs, life cycles, and the laws of software evolution", *Proceedings of the IEEE*, **68**(1980), 9, pp. 1060-1076.

[3] Bourne, S.R., "An introduction to the UNIX shell", **in** *UNIX Programmer's Manual*, Vol. 2A, Bell Telephone Laboratories, Inc., 7th Ed., 1979.

[4] Boute, R.T., "Building a uniform programming environment based on data abstraction", *International Computing Symposium (ICS81)*, 1981, pp. 415-424.

[5] Heering, J., & Klint, P., "Towards monolingual programming environments", Rapport IW 185/81, Mathematisch Centrum, 1981.

[6] Klint, P., "Partiële evaluatie als implementatiemethode voor een programmeeromgeving", deze syllabus, pp. 83-104.

[7] Geurts, L.J.M, "Ontwerp van een programmeeromgeving voor een personal computer", deze syllabus, pp. 39-51.

[8] Standish, Th. A., "Extensibility in programming language design", *SIGPLAN Notices*, **10**(1975), 7, pp. 18-21.

[9] Bell, J.R., "Transformations: the extension facilitiy of PROTEUS", *Proceedings of the Extensible Languages Symposium*, May 13, 1969, *SIGPLAN Notices*, **4**(1969), 8, pp.27-31.

[10] Klint, P., "Formal language definitions can be made practical", **in** De Bakker, J.W., & Van Vliet, H., (Eds.), *Proceedings of the International Symposium on Algorithmic Languages*, North-Holland, 1981, pp. 115-132.

[11] Sandberg, D., "LITHE: A language combining a flexible syntax and classes", *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1982, pp. 142-145.

[12] Chu, K.-H., & Fu, K.-S., "VLSI architectures for high-speed recognition of context-free languages and finite-state languages", *Proceedings of the Ninth Annual Symposium on Computer Architecture*, *SIGARCH Newsletter*, **10**(1982), 3, pp. 43-49.

Partiële Evaluatie als Implementatiemethode voor een Programmeeromgeving

Paul Klint
Mathematisch Centrum
Amsterdam

SAMENVATTING

Partiële evaluatie is een berekeningsprincipe dat het mogelijk maakt om een programma waarvan nog niet alle parameters gespecificeerd zijn om te zetten in een nieuw programma waarin de waarden van reeds bekende parameters zoveel mogelijk verwerkt zijn. Allerlei ad hoc technieken voor optimalisatie en code-generatie in traditionele compilers blijken met behulp van partiële evaluatie gerealiseerd te kunnen worden. Verder blijkt dit principe ook toepasbaar te zijn bij de implementatie van verschillende onderdelen van programmeeromgevingen.

In deze bijdrage wordt, aan de hand van een miniatuurprogrammeertaal, uitvoerig ingegaan op praktische en theoretische aspecten van partiële evaluatie. Voor deze programmeertaal wordt een volledige partiële evaluator behandeld. Daarnaast komen verschillende onderdelen van bestaande en toekomstige programmeeromgevingen ter sprake die met behulp van partiële evaluatie gerealiseerd kunnen worden.

1. INLEIDING

In een programmeeromgeving kan men allerlei talen en subtalen onderscheiden die nodig zijn om de diverse componenten ervan toe te spreken. Voor de hand liggende voorbeelden zijn talen voor job control, tekstediting en -formatting, debugging en uiteraard de 'gewone' programmeertalen. Bij pogingen om deze verschillende talen zoveel mogelijk te integreren, en te komen tot een eentalige programmeeromgeving [HEE81], stuit men al snel op de niet te ontkennen, of te omzeilen, behoefte aan speciale *applicatietalen*. Met andere woorden, zelfs als de programmeeromgeving gebaseerd is op een krachtige, universele basistaal, dan doet toch de behoefte zich voelen om voor sommige toepassingen aparte notaties in te voeren, omdat ze handiger in het gebruik zijn of aansluiten bij de traditionele notatie in het toepassingsgebied, of om de basistaal ten behoeve van de toepassing met nieuwe constructies en concepten uit te breiden. Een voorbeeld van een op syntactische uitbreiding gebaseerde applicatietaal is de commandotaal van een teksteditor: de commando's van een teksteditor moeten kort en bondig zijn maar kunnen verder de stringmanipulatiefunkties uit de basistaal 'lenen' als deze maar krachtig genoeg zijn. Zowel syntactische als semantische uitbreiding van de basistaal zal, bijvoorbeeld, nodig zijn om een query-taal in het systeem te integreren: de query-taal zèlf vereist een syntactische uitbreiding, maar de databankprimitieven die voor de uitvoering hiervan nodig zijn moeten waarschijnlijk als

semantische uitbreiding aan het systeem toegevoegd worden.

Deze gedachtengang leidt, zoals in de voorafgaande bijdrage [HEE82] uiteengezet, tot de conclusie dat een programmeeromgeving gebaseerd moet zijn op *taaldefinities*. Meer konkreet betekent dit dat nieuwe syntaxregels toegevoegd kunnen worden waarvan de semantiek op een operationele, interpretatieve, wijze beschreven is. Als deze semantiekbeschrijvingen gebruik mogen maken van eerdere toevoegingen, dan leidt deze opzet tot een stapeling van interpretatieniveau's. De centrale vraag in deze bijdrage luidt nu:

> *Hoe kunnen deze meertraps, interpretatieve semantiekbeschrijvingen op een efficiënte manier geïmplementeerd worden?*

Het is duidelijk dat hierbij behoefte bestaat aan een algemeen optimalisatiemechanisme, dat

- de rol van compilatie in traditionele systemen kan overnemen,
- incrementeel toegepast kan worden, en
- zich ertoe leent om in de evaluator van een basistaal ingepast te worden.

In de nu volgende paragrafen wordt uiteengezet welke rol het principe van *partiële evaluatie* in dit verband kan spelen. Hiertoe wordt dat principe eerst zelf uitvoerig besproken. Daarna worden toepassingen ervan behandeld in traditionele programmeeromgevingen én in de hierboven al genoemde uitbreidbare programmeeromgevingen.

2. WAT IS PARTIELE EVALUATIE?

Men spreekt van *partiële evaluatie* (ook wel *symbolische interpretatie* of *gemengde berekening* genoemd) als een programma met een verzameling invoerwaarden alvast zoveel mogelijk uitgevoerd wordt terwijl nog maar een deel van de invoerwaarden beschikbaar is. Zo zal, bijvoorbeeld,

if $x < 1$ **then** $y := y + x$ **else** $y := z$ **fi**

vereenvoudigd kunnen worden tot $y := y - 1$ als bekend is dat x de constante waarde -1 heeft. Een ander voorbeeld: uit de procedure

proc *join* (x, y) *return*$(2 * x + y)$

kan voor een vaste waarde van x, zeg 3, een nieuwe procedure afgeleid worden waarin het argument x zoveel mogelijk verwerkt is. Dit levert:

proc $join_3$ (y) *return*$(6 + y)$.

Alle aanroepen van de vorm *join*(3, . . .) kunnen hierna vervangen worden door $join_3(\cdots)$.

Een partiële evaluator probeert op grond van de beschikbare informatie zoveel mogelijk delen van een programma uit te voeren. Als alle waarden die in een expressie voorkomen bekend zijn, kan deze uitgerekend worden en kan het resultaat de oorspronkelijke expressie vervangen. Als alle waarden in de test van een if-statement bekend zijn, kan de test geëvalueerd worden en kan op grond van de uitkomst één van beide takken geselecteerd en de andere verwijderd worden. Er zijn uiteraard veel meer gevallen te noemen. Daarop kom ik later nog terug.

Partiële evaluatie is een vorm van *programmatransformatie* en is bruikbaar voor het optimaliseren, testen en verifiëren van programma's. We zullen in het volgende enkele nogal uiteenlopende toepassingen tegenkomen.

3. PICO: EEN VOORBEELD

In deze paragraaf zal ik ingaan op de vraag hoe een partiële evaluator er nu precies uitziet. Een poging om een partiële evaluator voor een taal als PASCAL of LISP in enige mate van detail te beschrijven zou de omvang van deze bijdrage al snel buiten proportie doen toenemen. Daarom wordt nu eerst de miniatuurtaal 'PICO' ingevoerd die het mogelijk maakt om net niet triviale programmaatjes te schrijven, maar die tegelijkertijd zo eenvoudig is dat een volledige partiële evaluator voor de taal behandeld kan worden.

3.1. Voorbereidingen

De grammatica voor PICO is in figuur 1 gegeven. Enkele begrippen zoals <var> (variabele), <const> (integer-constante) en <op> (operatorsymbool) zijn daarin niet gedefinieerd maar hebben een voor de hand liggende vorm en betekenis. Een legaal PICO-programma ter berekening van x^n is bij wijze van voorbeeld gegeven in figuur 2. Dit voorbeeld is ontleend aan [ERS82] en zal in § 3.3 verder gebruikt worden.

```
<unit>   ::= <const> | <var> .
<stat>   ::= <asg> | <if> | <while> .
<asg>    ::= <var> ':=' <expr> .
<expr>   ::= <unit> | <unit> <op> <unit> .
<if>     ::= if <expr> then <series> else <series> fi .
<while>  ::= while <expr> do <series> od .
<series> ::= <stat> | <stat> ';' <series> .
<pico>   ::= <series> .
```

Figuur 1. De grammatica voor PICO.

```
y := 1;
while n > 0
do  t := n mod 2;
    while t = 0
    do  n := n / 2;
        x := x * x;
        t := n mod 2
    od;
    n := n − 1;
    y := y * x
od
```

Figuur 2. PICO-programma ter berekening van x^n.

Een eerste stap op weg naar een partiële evaluator is het construeren van een conventionele evaluator. We zullen hierbij gebruik maken van vier soorten hulpfunkties:

(1) **Predikaten** voor syntactische herkenning, die vaststellen of een gegeven tekst van een bepaalde syntactische vorm is. Al deze predikaten hebben namen die beginnen met '*is-*' gevolgd door de naam van de syntactische categorie die ze herkennen. Zo zal, bijvoorbeeld, het predikaat *is-asg('x := 3')* waar zijn, maar *is-if ('x := 3')* niet.

(2) **Extractors** die componenten van een bepaalde syntactische vorm uit een tekst halen. Deze extractors hebben namen die beginnen met de naam van de syntactische categorie die ze extraheren (eventueel gevolgd door een getal om ambiguiteiten op te heffen) en eindigen steeds op '*-in*'. Zo levert de extractor

test-in('while x = 0 do y := 3 od')

de waarde *'x = 0'* op en

series1-in('if x = 0 then x := 1 else x := 2 fi')

de waarde *'x := 1'*.

(3) **Constructors** die uit gegeven componenten een nieuwe tekst van een bepaalde syntactische categorie maken. Constructors hebben namen die beginnen met '*mk-*' gevolgd door de naam van de syntactische categorie. Zo levert *mk-asg('x', 'y * 3')* de tekst *'x := y * 3'* op.

(4) De **waardenmanipulatiefunkties** *value-of* en *assign* die respectievelijk de waarde van een variabele opleveren en een nieuwe waarde aan een variabele toekennen. De waarden van variabelen worden bewaard in een hier niet nader gespecificeerde *omgeving*, die men zich het beste kan voorstellen als een lijstje van variabele-namen met hun bijbehorende waarden.

3.2. Een conventionele evaluator

Na deze voorbereidingen laat een evaluator voor PICO zich eenvoudig opschrijven. Dat is in figuur 3 dan ook gebeurd. Dyadische expressies worden geëvalueerd door *eval-dyadic-op* waarvan de tekst hier weggelaten is. Het resultaat van de evaluatie kan waargenomen worden door in de, hierboven al genoemde, *omgeving* de waarden van variabelen te inspecteren. Deze conventionele evaluator voor PICO is tamelijk transparant, bevat weinig verrassingen en vormt derhalve een goed fundament om een partiële evaluator op te bouwen.

3.3. Een partiële evaluator

Het partiële evaluatieproces wordt in eerste instantie gestuurd door informatie over het al dan niet beschikbaar zijn van waarden van variabelen. Het predikaat *available* stelt vast of een waarde beschikbaar is of niet. (Iets preciezer: *available* is alleen waar voor integer constanten en is onwaar voor alle andere waarden, inclusief programmateksten.) Omgekeerd markeert *mark-not-available* een variabele als niet-beschikbaar. De beschikbaarheidseigenschap wordt tijdens partiële evaluatie gepropageerd zodat van ingewikkelder constructies ook vastgesteld kan worden of hun waarde beschikbaar is of niet.

Een partiële evaluator voor PICO is in figuren 4a en 4b weergegeven. De partiële evaluator levert, net als *pico-eval*, een omgeving op als het mogelijk is met de beschikbare waarden van variabelen het programma volledig uit te voeren of een residuprogramma plus resulterende omgeving indien dit niet het geval is. Zo levert een variabele waarvan de waarde niet beschikbaar is zichzelf (d.w.z. de variabele-naam) als waarde op. Een assignment waarvan zowel linker- als rechterkant beschikbaar zijn

```
pico-eval(s):

is-const(s)   → const-in(s);
is-var(s)     → value-of(var-in(s));
is-asg(s)     → assign(var-in(s), pico-eval(expr-in(s)));
is-expr(s)    → eval-dyadic-op(op-in(s),
                        pico-eval(unit1-in(s)),
                        pico-eval(unit2-in(s)));
is-if(s)      → if pico-eval(expr-in(s)) = TRUE
                then pico-eval(series1-in(s))
                else pico-eval(series2-in(s))
                fi;
is-while(s)   → if pico-eval(expr-in(s)) = TRUE
                then pico-eval(series-in(s));
                     pico-eval(s)
                fi;
is-series(s)  → pico-eval(stat-in(s));
                if not empty(series-in(s)) then pico-eval(series-in(s)) fi;
```

Figuur 3. Conventionele evaluator voor PICO.

kan gewoon uitgevoerd worden. Als dit niet het geval is dan wordt een nieuw (eventueel vereenvoudigd) assignment als waarde opgeleverd. Bovendien moet dan soms de variabele aan de linkerkant als niet-beschikbaar gemarkeerd worden en kan soms de assignment tijdens de partiële evaluatie ook nog uitgevoerd worden. De details zijn te vinden in de figuren. Evaluatie van programmateksten die qua vorm in een van de overige syntactische categorieën vallen, levert op vergelijkbare wijze òf een waarde òf een (wellicht vereenvoudigde) programmatekst op. In figuur 5 zijn een aantal voorbeelden gegeven van reducties die op deze manier te bereiken zijn. In deze voorbeelden is aangenomen dat x niet beschikbaar is en dat n beschikbaar is en waarde 5 heeft. Merk op dat naast het aangegeven resultaat de evaluatie van deze statements ook aanleiding geeft tot wijzigingen in de omgeving: deze zijn in de figuur niet weergegeven.

Bestudering van de hier gepresenteerde partiële evaluator roept allerlei vragen op, zoals:

- Is het mogelijk om de **then**- en **else**-tak van een **if**-statement alvast partieel te evalueren als de uitkomst van de test niet beschikbaar is?
- De evaluatiewijze van **while**-statements kan leiden tot zeer omvangrijke residuprogramma's als de test beschikbaar is en het statement zeer vaak doorlopen wordt. Is hier iets aan te doen? Wat te doen als het **while**-statement niet termineert?

Deze en vergelijkbare problemen kunnen in het algemeen opgelost worden ten koste van een toename in complexiteit van de partiële evaluator. Ik kom hier in § 5 nog op terug.

```
partial-pico-eval(s):

is-const(s)   → const-in(s);
is-var(s)     → v := var-in(s);
                c := value-of(v);
                if available(c)
                then c
                else mk-var(v)
                fi;
is-asg(s)     → v := var-in(s);
                lhs := value-of(v);
                rhs := partial-pico-eval(expr-in(s));
                if available(lhs)
                then if available(rhs)
                     then assign(v, rhs)
                     else mark-not-available(v);
                          mk-asg(v, rhs)
                     fi
                else if available(rhs) then assign(v, rhs) fi;
                     mk-asg(lhs, rhs)
                fi;
is-expr(s)    → left := partial-pico-eval(unit1-in(s));
                right := partial-pico-eval(unit2-in(s));
                if available(left) & available(right)
                then eval-dyadic-op(op-in(s), left, right)
                else mk-expr(left, op-in(s), right)
                fi;
```

Figuur 4a. Partiële evaluator voor PICO.

4. TOEPASSING VAN PARTIELE EVALUATIE BIJ DE IMPLEMENTATIE VAN PROGRAMMEERTALEN

Na de voorgaande uiteenzetting zal het duidelijk zijn dat een groot aantal optimalisaties die traditioneel door middel van verschillende *ad hoc* methodes bereikt worden door middel van partiële evaluatie onder één noemer gebracht kunnen worden:

- Propageren van constanten ('constant folding').
- Uitrollen van loops.
- Elimineren van niet bereikbare code. Dit komt in het geval van PICO neer op het verwijderen van takken van **if**-statements of bodies van **while**-statements die nooit uitgevoerd kunnen worden.
- Integratie en specialisatie van procedures. Deze optimalisatie kan, gezien het ontbreken van procedures in de taal, niet aan de hand van PICO geïllustreerd worden. De essentie is dat indien de aanroep van een procedure één of meer parameters bevat waarvan de waarden beschikbaar zijn tijdens de partiële evaluatie deze informatie gebruikt kan worden om de body van de aangeroepen procedure te vereenvoudigen. Deze vereenvoudigde procedure-bodies kunnen

```
is-if (s)       → test := partial-pico-eval(expr-in(s));
                  if available(test)
                  then if test = TRUE
                       then partial-pico-eval(series1-in(s))
                       else partial-pico-eval(series2-in(s))
                       fi
                  else mk-if (test, series1-in(s), series2-in(s))
                  fi;
is-while(s)     → test := partial-pico-eval(expr-in(s));
                  if available(test)
                  then if test = TRUE
                       then body := partial-pico-eval(series-in(s));
                            if available(body)
                            then partial-pico-eval(s)
                            else mk-series(body, partial-pico-eval(s))
                            fi
                       else
                            nil
                       fi
                  else mk-while(test, series-in(s))
                  fi;
is-series(s)    → first := partial-pico-eval(stat-in(s));
                  if available( first)
                  then if empty(series-in(s))
                       then first
                       else partial-pico-eval(series-in(s))
                       fi
                  else if empty(series-in s)
                       then first
                       else mk-series( first, partial-pico-eval(series-in(s)))
                  fi;
```

Figuur 4b. Partiële evaluator voor PICO (vervolg).

òf als nieuwe procedure toegevoegd worden (men spreekt dan van *specialisatie*: de oorspronkelijke aanroep wordt vervangen door een aanroep naar deze nieuwe, gespecialiseerde, procedure) òf ze kunnen de aanroep zelf vervangen (dit wordt *integratie* of *macro-expansie* genoemd).

Enkele andere toepassingen die op het grensvlak liggen van programmeertaal en programmeeromgeving worden in § 7 verder besproken.

5. BEPERKINGEN, PROBLEMEN EN UITBREIDINGEN

Tot nu toe is partiële evaluatie opgevoerd als panacee. Bij implementatie of toepassing van de methode doen zich echter verschillende problemen voor. Laat ik met de implementatieproblematiek beginnen. Het blijkt dat tijdens de reductie van programmafragmenten tussenresultaten ontstaan die, zonder dat er speciale maatregelen getroffen worden, niet verder reduceerbaar zijn. In geval van PICO zijn

Expressie/statement	Resultaat
x	x
n	5
$x := n$	$x := 5$
$x := 2 * n$	$x := 10$
if $n = 2$ **then** $x := 3$ **else** $x := 4$ **fi**	$x := 4$
while $x = 3$ **do** $n := 4$ **od**	**while** $x = 3$ **do** $n := 4$ **od**
while $n > 4$ **do** $x := 7$; $n := n - 1$ **od**	$x := 7$
(exponent uit figuur 2)	$y := 1 * x$; $x := x * x$; $x := x * x$; $y := y * x$

(Aannames in alle voorbeelden:
x niet beschikbaar;
n beschikbaar met waarde 5.)

Figuur 5. Voorbeelden van partieel geëvalueerde PICO statements.

bijvoorbeeld

$y := x$; $y := x$
if $p > 0$ **then** $z := a$ **else** $z := a$ **fi**
$s := 1$; **while** $s >= 0$ **do if** $s <= 0$ **then** . . . **fi**; $s := -1$ **od**

in principe reduceerbaar tot respectievelijk $y := x$, $z := a$ en $s := -1$. In een LISP-context zijn expressies als

(*car* (*cons* x y))
(*plus* 3 x 5)
(*append nil* z)

te reduceren tot respectievelijk x, (*plus* x 8) en z. Dergelijke statements zullen weliswaar nooit door een programmeur opgeschreven worden maar kunnen gemakkelijk als tussenresultaat tijdens het evaluatieproces ontstaan. De partiële evaluator zoals eerder gegeven zal deze reducties echter niet vinden. Een partiële evaluator die dergelijke vereenvoudigingen wél aankan zal gestuurd moeten worden door een verzameling herschrijfregels voor de betrokken programmeertaal en zal gebruik moeten maken van technieken die ook in stellingenbewijsprogramma's toegepast worden (zie bijvoorbeeld [CHA80] voor implementatietechnieken op dit gebied). Dit vormt uiteraard een principiële en implementatie-technische complicatie van de partiële evaluator.

Een ander, veel moeilijker, probleem doet zich voor als de programmeertaal in kwestie neveneffecten toelaat. In dat geval is het, bijvoorbeeld, niet zonder meer duidelijk of een procedure-aanroep met constante argumenten door de waarde van de procedure-aanroep vervangen mag worden. De aanroep zou immers neveneffecten kunnen hebben die bij conventionele evaluatie pas in een later stadium tot stand zouden komen. Er moet dus zorgvuldig voor worden gewaakt dat de betekenis van het oorspronkelijke programma door partiële evaluatie niet gewijzigd wordt. Om dit te bereiken is een uitvoeriger analyse van het programma nodig. Deze is ook om andere redenen gewenst.

Een beperking van de tot nu toe besproken vormen van partiële evaluatie is dat slechts *voorwaartse* (om in termen van dataflow-analyse [HEC77] te spreken) optimalisaties mogelijk zijn, d.w.z. optimalisaties die gebaseerd zijn op het doorlopen van een programmatekst in een volgorde die overeenkomt met de normale evaluatievolgorde. Dit sluit bijvoorbeeld uit dat op grond van het programmafragment

$x := y;$
$y := 3 * z;$

de conclusie getrokken kan worden dat x òf een integer òf een real waarde moet hebben (onder de aanname dat de vermenigvuldigoperatie alleen maar waarden van deze types op kan leveren). Een dergelijke conclusie kan alleen getrokken worden door middel van *achterwaartse* analyse van het programma. Een op deze analyserichting gebaseerde evaluatiemethode is de zogenaamde *luie* [FRI76, HEN76] evaluatie. Een zeer interessante, maar hier verder niet uitgewerkte, mogelijkheid is dan ook om gebruik te maken van een evaluatiemethode die deze twee analyserichtingen in zich verenigt. Hierdoor kan bijvoorbeeld ook het elimineren van gemeenschappelijke expressies bereikt worden met behulp van één, algemeen evaluatieschema.

6. THEORETISCHE ACHTERGROND

We hebben hierboven gezien hoe voor een gegeven taal, in dit geval PICO, zowel een gewone als een partiële evaluator geconstrueerd kunnen worden. Voor een PICO-programma $\pi(x,y)$ met argumenten $x=a$ en $y=b$ geldt nu:

$$pico\text{-}eval(\pi,a,b)=\pi(a,b)$$

d.w.z. executie van de interpreter *pico-eval* met argumenten π, a en b levert hetzelfde resultaat op als executie van π met argumenten a en b. Als alleen de waarde van x bekend is, maar niet die van y (of omgekeerd) dan kan *pico-eval* niets doen, *partial-pico-eval* wel:

$$partial\text{-}pico\text{-}eval(\pi,a,y)=\pi_a(y)$$

waarin π_a het residuprogramma van π voor $x=a$ is. Voor een willekeurige taal T met interpreter *eval* en partiële evaluator *part-eval* en voor een willekeurig T-programma π geldt natuurlijk hetzelfde:

$$eval(\pi,a,b)=\pi(a,b)$$
$$part\text{-}eval(\pi,a,y)=\pi_a(y)$$

Zowel *eval* als *part-eval* zijn zelf weer in een implementatietaal T' geschreven. In het vervolg zal ik, ter vereenvoudiging van de presentatie, twee beperkingen introduceren en wel dat T en T' één en dezelfde taal zijn en dat, als T-programma's gecompileerd worden, T ook als doeltaal (objecttaal) gebruikt wordt. Hierbij moeten twee aantekeningen gemaakt worden:

(1) De taal PICO zoals eerder gedefinieerd zou enigzins uitgebreid moeten worden om als implementatietaal te kunnen dienen: toevoeging van procedures en primitieven voor stringmanipulatie is daarvoor dringend gewenst. Vandaar dat ik nu verder over T i.p.v. PICO zal spreken.

(2) Het laten samenvallen van brontaal en doeltaal is minder vreemd dan op het eerste gezicht lijkt. Veel taalconstructies (bijvoorbeeld **for**-statements) kunnen immers vertaald worden naar simpeler constructies (bijvoorbeeld **goto**-statements) in dezelfde taal. Meer in het algemeen zijn bij systemen voor programmatransformatie bron- en doeltaal identiek. Een algemenere behandeling waarbij bron- en doeltaal niet noodzakelijk gelijk hoeven te zijn is te vinden in [ERS82].

Goed beschouwd spelen twee gereduceerde versies van een programma een rol. Aan de ene kant leidt partiële evaluatie tot reductie van een programma. Aan de andere kant is de objectcode van een programma ook een gereduceerde versie van dat programma. De vraag dringt zich dan ook op:

Is het mogelijk om eval en part-eval zó te combineren dat voor elk programma de bijbehorende objectcode gevonden kan worden?

Ik zal nu laten zien dat compilatie beschreven kan worden als vorm van partiële evaluatie. Alvorens dit te doen moeten de begrippen 'objectcode' en 'compiler' iets aangescherpt worden:

- Voor ieder programma π in T bestaat een programma ob in T, de objectcode voor π, zodat voor alle mogelijk argumenten x geldt: $\pi(x)=ob(x)$. Hieraan is uiteraard op triviale wijze te voldoen. In het algemeen zal men nog als extra eis willen stellen dat de objectcode efficienter is dan het oorspronkelijke programma.
- Er bestaat een compiler *comp* voor T zodat voor alle programma's π en alle argumenten x geldt: $comp(\pi,x)=ob(x)$.

Als we de voorgaande resultaten combineren krijgen we:

$$part\text{-}eval(eval,\pi,x)=eval_{\pi}(x)$$

Bovendien geldt

$$eval_{\pi}(x)=eval(\pi,x)=\pi(x)$$

zodat $eval_{\pi}$ als objectcode voor π kan fungeren, d.w.z.

$$eval_{\pi}(x)=ob(x)$$

of, anders uitgedrukt:

de objectcode is de projectie van de evaluator op het bronprogramma.

Intuïtief is dit resultaat aannemelijk. In het simpelste geval wordt een programma conventioneel geïnterpreteerd. De objectcode van dat programma wordt nu verkregen door deze conventionele interpretatie van het programma op zich weer partieel te evalueren. Dit leidt ertoe dat zoveel mogelijk acties die behoren bij de interpretatie van het programma alvast uitgevoerd worden. Het restprogramma dat hierbij overblijft is de objectcode van het programma.

Deze redenering nog een stap voortzettend krijgen we:

$$part\text{-}eval(part\text{-}eval,eval,(\pi,x))=part\text{-}eval_{eval}(\pi,x)$$

en

$$part\text{-}eval_{eval}(\pi,x) = part\text{-}eval(eval,\pi,x)$$
$$= eval_{\pi}(x)$$
$$= ob(x)$$

zodat $part\text{-}eval_{eval}$ als compiler voor T kan fungeren, d.w.z.

$$part\text{-}eval_{eval}(\pi,x) = comp(\pi,x)$$

of, met andere woorden gezegd:

> *een compiler voor T is de projectie van de partiële evaluator op de conventionele evaluator.*

De intuïtie achter dit resultaat is dat bepaalde algemene aspecten van het proces om de objectcode voor een specifiek programma te vinden door partiële evaluatie geïsoleerd kunnen worden.

Deze resultaten lijken ver af te staan van programmeeromgevingen in praktische zin. Bij nadere beschouwing blijkt die afstand toch niet zo groot te zijn. De essentie is namelijk dat door partiële evaluatie allerlei vormen van interpretatie-overhead geëlimineerd kunnen worden. Dit komt in een programmeeromgeving op verschillende manieren tot uiting zoals uit het vervolg zal blijken. Bovendien geven deze resultaten een antwoord op de, in de inleiding van deze bijdrage gestelde, vraag omtrent de mogelijkheden tot optimalisatie van meertraps, interpretatieve semantiekbeschrijvingen, zoals in § 7.2 verder uiteengezet wordt.

7. TOEPASSING VAN PARTIELE EVALUATIE BIJ DE IMPLEMENTATIE VAN PROGRAMMEEROMGEVINGEN

7.1. Toepassingen in conventionele programmeeromgevingen

Het principe van partiële evaluatie kan op verschillende manieren in een conventionele programmeeromgeving toegepast worden. Dergelijke toepassingen leiden in het algemeen tot homogenisering van de programmeeromgeving zelf en tot vereenvoudiging van de implementatie.

Een eerste toepassing betreft het proces van *linkage-editing* dat in de meeste systemen nodig is om onafhankelijk van elkaar vertaalde modulen zó aan elkaar te knopen dat een executeerbaar programma ontstaat. Hierbij worden nog niet opgeloste externe referenties vanuit het ene module naar variabelen of procedures in een andere module opgeheven. Het is, om te beginnen, eigenlijk al vreemd dat referenties naar grootheden binnen eenzelfde module geheel anders behandeld worden (namelijk door de compiler) dan referenties naar externe grootheden. Het wordt echter nog veel vreemder als men bedenkt dat de strenge type-controle die toegepast wordt op referenties binnen een module vaak verzwakt (of overgeslagen) wordt bij externe referenties. Dit treedt op bij implementaties van programmeertalen die zelf geen voorzieningen voor afzonderlijke compilatie van modulen kennen (denk hierbij aan PASCAL, C, ALGOL68) maar waaraan op bedrijfssysteemniveau een dergelijke voorziening is toegevoegd. Gebruikers van dergelijke systemen zullen zich maar al te goed de gevallen herinneren waarin het simpele feit dat bij aanroep van een externe procedure een actuele parameter vergeten of van een verkeerd type was leidde tot totale ontsporing van hun programma. Beschouwd vanuit het partiële evaluatieperspectief vormt linkage-editing een onderdeel van het compilatieproces dat eventueel, bij gebrek aan informatie over een gebruikte module, uitgesteld moet worden. Zodra de ontbrekende

informatie beschikbaar komt worden deze restanten van de compilatie uitgevoerd en worden (uiteraard) de benodigde type-controles uitgevoerd. Kortom, de type-controles zijn ook op externe referenties van toepassing (waardoor het systeem homogener wordt) en de implementatie als geheel wordt eenvoudiger (want uitgaande van een partiële evaluator is geen apart linkage-editor meer nodig).

Een tweede toepassing van partiële evaluatie in conventionele programmeeromgevingen betreft het bijhouden van verschillende *versies van programmatuur*. Ik denk hierbij aan het geval dat uit één programma een aantal gespecialiseerde varianten afgeleid wordt. Er zijn verschillende methodes om dit te bereiken waarvan ik alleen *compile-time directieven* en *herstartbare programmadumps* zal behandelen.

Compile-time directieven zijn aanwijzigingen aan een compiler om (onder andere) bepaalde stukken van een programma al dan niet in een compilatie over te slaan. Als een programma, bijvoorbeeld, een versie bestemd voor de VAX en voor de PDP11 bevat, dan zou dit kunnen leiden tot een programmatekst zoals in figuur 6 weergegeven. Hier worden stukken programma geselecteerd op grond van het al dan niet gedefinieerd zijn van de constanten *VAX* en *PDP11*. Dit effect wordt bereikt met behulp van de speciale **ifdef** constructie. Bij toepassing van partiële evaluatie zou hetzelfde effect echter bereikt kunnen worden met het standaard **if**-statement. In figuur 7 is *machine* een extra parameter van het programma. Als de waarde van *machine* gegeven is, levert partiële evaluatie een voor die waarde van *machine* gespecialiseerde versie van het programma op.

```
#ifdef VAX
        wordlength = 32;
#else
#ifdef PDP11
        wordlength = 16;
#endif
#endif
        maxinteger = 2 ** (wordlength - 1) - 1;
```

Figuur 6. Programma-variant gerealiseerd met compile-time directieven.

```
if machine = VAX then
        wordlength := 32;
elif machine = PDP11 then
        wordlength := 16;
fi;
maxinteger := 2 ** (wordlength - 1) - 1;
```

Figuur 7. Programma-variant gerealiseerd met **if**-statement.

Compile-time directieven kunnen alleen gebruikt worden als van te voren bekend is welke versies van een programma nodig zullen zijn. Het komt echter regelmatig voor dat het niet mogelijk is om een programma op een dergelijke manier te parametriseren. Tekstverwerkende systemen maken, bijvoorbeeld, vaak gebruik van macropakketten om verschillende soorten layouts te beschrijven. De combinatie van tekstverwerker plus macropakket kan gezien worden als een gespecialiseerde tekstverwerker voor één layout (namelijk de layout die door het macropakket gedefinieerd wordt).

Dergelijke gevallen worden gekenmerkt door een initialisatiefase waarin 'definities' ingelezen worden, gevolgd door een feitelijke 'verwerkings'-fase. Een traditionele optimalisatie hiervan is de **herstartbare programmadump**: zodra de initialisatie voltooid is wordt het programma met alle daarbij behorende variabelen gedumpt. Deze dump kan later herstart worden op het punt dat de initialisatie juist voltooid was zodat meteen met de verwerking begonnen kan worden. Deze gang van zaken levert uiteraard alleen winst op als de initialisatiefase naar verhouding tijdrovend is. Het zal ook duidelijk zijn dat het creëren van dergelijke dumps zeer speciale, systeemafhankelijke technieken vereist. In het geval van partiële evaluatie kan dit effect echter zonder speciale maatregelen bereikt worden.

7.2. Toepassingen in uitbreidbare programmeeromgevingen

Veel van de toepassingen van partiële evaluatie in conventionele programmeeromgevingen kunnen, mutatis mutandis, ook in uitbreidbare programmeeromgevingen hun weg vinden. Ik zal daarom alleen enkele toepassingen bespreken die typisch zijn voor uitbreidbare programmeeromgevingen.

Een belangrijk probleem in dergelijke systemen is de al eerder genoemde stapeling van interpretatieniveau's. Dankzij partiële evaluatie kan een gedeelte van de interpretatie-overhead al vóór de executie van een programma geëlimineerd worden. In essentie is dit streven ook al in sommige conventionele systemen zichtbaar. LISP wordt, bijvoorbeeld, vaak gebruikt om interpreters voor andere talen (zeg T) in te implementeren. De combinatie van LISP-systeem plus de in LISP geschreven T-interpreter kan gezien worden als een speciaal T-systeem. Door een gespecialiseerde versie van het LISP-systeem te maken, zoals in de vorige paragraaf beschreven, kan een gedeelte van de dubbele interpretatie-overhead teruggewonnen worden. In een uitbreidbare programmeeromgeving kan dit nog veel verder doorgevoerd worden. Immers, de semantiek die met iedere aan het systeem toegevoegde syntaxregel geassocieerd is bestaat uit een *vast* stuk tekst. De syntax-analyse van al dergelijke stukken tekst kan al van te voren uitgevoerd worden. Hierdoor wordt meervoudige interpretatie-overhead geëlimineerd. Men moet er hierbij wel goed op letten dat dit niet tot gevolg heeft dat de omvang van de resulterende semantiekregels onevenredig toeneemt.

Toepassing van partiële evaluatie leidt tot opheffen van het onderscheid tussen *compile-time* en *run-time* zoals dat in conventionele systemen bestaat. In sommige van de eerder gegeven voorbeelden (linkage-editing, compile-time directieven) kwam dit onderscheid, met de ermee samenhangende problemen, al naar voren. Opheffen van dit onderscheid strijkt niet alleen de kreukels in conventionele programmeeromgevingen glad, maar introduceert ook nieuwe mogelijkheden:

- Programmeertalen worden momenteel gerubriceerd naar het moment waarop type-controle van variabelen plaatsvindt: tijdens compilatie (PASCAL, ALGOL68) of tijdens executie (SNOBOL4, LISP). Toepassing van partiële evaluatie leidt tot een systeem van type-controle dat deze controle *zo vroeg mogelijk* uitvoert.
- De scheiding tussen compile-time en run-time impliceert dat de volledige tekst van een programma beschikbaar moet zijn voordat tot executie overgegaan kan worden. Deze beperking is in allerlei toepassingen ongewenst (macroprocessoren, editors, rule-based kennisbestanden, enz.).

Tenslotte laat het zich aanzien dat allerlei *programmageneratoren* (parser- en compilergeneratoren, syntaxgestuurde editors en prettyprinters, enz.) zich op soepele wijze in een op partiële evaluatie gebaseerd systeem laten inpassen. De keuze van specifieke algoritmen die daarbij gebruikt kunnen worden zal grotendeels bepaald moeten worden door het gedrag dat dergelijke algoritmen vertonen onder partiële evaluatie. Algoritmen die eerst een grote hoeveelheid preprocessing doen voordat ze aan het uitvoeren van hun feitelijk taak toekomen zijn hierbij in het voordeel.

8. TOT BESLUIT

Partiële evaluatie is al een vrij oude methode. Als men zich niet van de wijs laat brengen door zaken als *B*-lijnen, AUTOCODE en andere details die te maken hebben met de ATLAS computer, dan blijkt de methode al in het begin van de zestiger jaren in de Brooker-Morris Compiler-Compiler toegepast te zijn [BRO63]. Toepassing op het gebied van code-generatie vindt men in [ERS77], [ERS82], [HAR78] en [MEE80]. Toepassingen in programmeeromgevingen worden beschreven in [BEC76]. Toepassing van partiële evaluatie bij het optimaliseren van taaldefinities wordt beschreven in [EMA80] en [KOM82].

Men kan zich afvragen waarom deze methode dan, als hij al zo lang aan ingewijden bekend was, niet op ruimere schaal toepassing gevonden heeft. Ik denk dat het antwoord eenvoudig is: op korte termijn leiden ad hoc methodes sneller tot succes. De kennis, de mogelijkheden en de wenselijkheden op het gebied van de implementatie van programmeertalen en programmeeromgevingen zijn nu echter in een fase gekomen waarin de toepassingsmogelijkheden van deze ad hoc methodes uitgeput beginnen te raken. Alleen een combinatie van een klein aantal, algemeen toepasbare optimalisatiemethodes zal het mogelijk maken om over de barrières heen te klauteren die zo snel binnen ons gezichtsveld beginnen te komen. Dit verklaart de momenteel toenemende belangstelling voor partiële evaluatie als researchthema.

LITERATUUR

[BEC76] Beckman, L., Haraldson, A., Oskarsson, O & Sandewall, E., "A partial evaluator, and its use as a programming tool", *Artificial Intelligence*, **7** (1976), pp. 319-357.

[BRO63] Brooker, R.A., MacCallum, I.R., Morris, D. & Rohl, J.S., "The Compiler Compiler", in Goodman, R. (ed.), *Annual Review in Automatic Programming*, Vol. **3**, 1963, pp. 229-275.

[CHA80] Charniak, E., Riesbeck, C.K. & McDermott, D.V., *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

[EMA80] Emanuelson, P. & Haraldson, A., "On compiling embedded languages in LISP", *Conference Record of the 1980 LISP Conference*, Stanford University, 1980, pp. 208-215.

[ERS77] Ershov, A.P., "On the partial computation principle," *Information processing letters*, **6** (1977), 2, pp. 38-41.

[ERS82] Ershov, A.P., "Mixed computation: potential applications and problems for study", *Theoretical Computer Science*, **18** (1982), pp. 41-67.

[FOD81] Foderaro, J.K. & Sklower, K.L., "The FRANZ LISP manual", University of California, Berkeley, California, 1981.

[FRI76] Friedman, D.P. & Wise, D.S., "CONS should not evaluate its arguments", **in** Michaelson, S. & Milner, P. (eds.), *Automata, Languages and Programming*, Edinburgh University Press, 1976, pp. 257-284.

[HAR78] Haraldson, A., "A partial evaluator, and its use for compiling iterative statements in LISP", *Conference Record of the Fifth Annual ACM Symposium on principles of Programming Languages*, 1978, pp. 195-202.

[HEC77] Hecht, M.S., *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[HEE81] Heering, J. & Klint, P., "Towards monolingual programming environments", Mathematisch Centrum Rapport, IW 185/81, 1981.

[HEE82] Heering, J., "Taaldefinities als kern voor een programmeeromgeving", deze syllabus, pp. 69-81.

[HEN76] Henderson, P. & Morris, J.H., "A lazy evaluator", *Conference Record of the Third ACM Conference on Principles of Programming Languages*, 1976, pp. 95-103.

[KOM82] Komorowski, H.J., "Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of PROLOG", *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982, pp. 255-267.

[MEE80] Meertens, L.G.L.T., "Code-generatie", **in** Van Vliet, J.C. (ed.), *Colloquium Capita Implementatie van Programmeertalen*, Mathematisch Centrum Syllabus **42**, 1980, pp. 27-45.

APPENDIX: DE PICO-EVALUATORS IN LISP

In deze appendix wordt een volledige implementatie gegeven van *pico-eval*, *partial-pico-eval* en benodigde hulpfunkties. De implementatietaal is FRANZ LISP zoals beschreven in [FOD81]. De voornaamste verschillen met standaard LISP zijn:

(1) Een *if-then-elseif*-constructie die iets leesbaarder is dan de standaard *cond*-constructie.

(2) De *backquote*-constructie die het mogelijk maakt om grote lijsten die grotendeels constant zijn leesbaar op te schrijven. Als *p* bijvoorbeeld de waarde (*x y z*) heeft, dan hebben '(*a b c ,p d*) en '(*a b c ,@p d*) respectievelijk de waarden (*a b c* (*x y z*) *d*) en (*a b c x y z d*).

Hier volgt, tenslotte, de volledige tekst van de evaluators en hulpfuncties:

```
(def pico-eval (lambda(s)
    (If     (is-const s) then (const-in s)
     elseif (is-var s) then (value-of (var-in s))
     elseif (is-asg s) then (assign (var-in s) (pico-eval (expr-in s)))
     elseif (is-if s) then
       (If (eq (pico-eval (expr-in s)) 'TRUE)
        then (pico-eval (series1-in s))
        else (pico-eval (series2-in s)))
     elseif (is-while s) then
       (If (eq (pico-eval (expr-in s)) 'TRUE)
        then (pico-eval (series-in s))
             (pico-eval s))
     elseif (is-series s) then
       (pico-eval (stat-in s))
       (If (not (null (series-in s)))
        then (pico-eval (series-in s)))
     elseif (is-expr s) then
      (eval-dyadic-op (op-in s)
                      (pico-eval (unit1-in s))
                      (pico-eval (unit2-in s)))
     else (error "pico-eval"))))

(def eval-dyadic-op (lambda (oper a b)
    (If     (eq oper '+) then (plus a b)
     elseif (eq oper '- then (diff a b)
     elseif (eq oper '*) then (times a b)
     elseif (eq oper '/) then (quotient a b)
     elseif (eq oper 'mod) then (remainder a b)
     elseif (eq oper '=) then (truth (eq a b))
     elseif (eq oper '<) then (truth (lessp a b))
     elseif (eq oper '>) then (truth (greaterp a b))
     else (error "eval-dyadic-op"))))

(def truth (lambda (val)
    (If val then 'TRUE else 'FALSE)))

(def value-of (lambda (var)
    (If (assoc var ENV) then (cdr (assoc var ENV))
     else (error '(,var not bound)))))

(def assign (lambda (var val)
    (setq ENV (append (list (cons var val)) ENV)) nil))

(def partial-pico-eval (lambda (s)
    (If (is-const s) then
      (prog ()
        (return (const-in s)))
     elseif (is-var s) then
      (prog (v)
```

```
    (setq v (var-in s))
    (If (available (value-of v)) then (return (value-of v))
     else (return (mk-var s))))
elseif (is-asg s) then
  (prog (v lhs rhs)
    (setq v (var-in s))
    (setq lhs (value-of v))
    (setq rhs (partial-pico-eval (expr-in s)))
    (If (available lhs) then
       (If (available rhs) then
          (return (assign v rhs))
        else
          (mark-not-available v)
          (return (mk-asg v rhs)))
     else
       (If (available rhs) then
          (assign v rhs))
       (return (mk-asg v rhs))))
elseif (is-if s) then
  (prog (test)
    (setq test (partial-pico-eval (expr-in s)))
    (If (available test) then
       (If (eq test 'TRUE) then
          (return (partial-pico-eval (series1-in s)))
        else
          (return (partial-pico-eval (series2-in s))))
     else
       (return (mk-if test (series1-in s) (series2-in s)))))
 elseif (is-while s) then
  (prog (test body)
    (setq test (partial-pico-eval (expr-in s)))
    (If (available test) then
       (If (eq test 'TRUE) then
          (setq body (partial-pico-eval (series-in s)))
          (If (available body) then (return (partial-pico-eval s))
           else (return (mk-series body (partial-pico-eval s))))
        else
          (return nil))
     else
       (return (mk-if test (mk-series (series-in s) s) nil))))
 elseif (is-expr s) then
  (prog (left right)
    (setq left (partial-pico-eval (unit1-in s)))
    (setq right (partial-pico-eval (unit2-in s)))
    (If (and (available left) (available right)) then
       (return (eval-dyadic-op (op-in s) left right))
     else
       (return (mk-expr left (op-in s) right))))
 elseif (is-series s) then
```

```
        (prog (first)
          (setq first (partial-pico-eval (stat-in s)))
          (If (available first) then
              (If (null (series-in s))
               then (return first)
               else (return (partial-pico-eval (series-in s))))
           else
              (If (null (series-in s))
               then (return first)
               else
                  (return (mk-series first (partial-pico-eval (series-in s)))))))
      else (error "partial-pico-eval"))))

(def available (lambda (form)
    (or (is-const form) (eq form 'TRUE) (eq form 'FALSE))))

(def mark-not-available (lambda (var) (assign var '*)))

;-----------------------------------------------------------------
; Here follow for each syntactic notion X the functions for
; • syntactic recognition (X, is-X),
; • selection as subphrase (X-in),
; • construction (mk-X).
;-----------------------------------------------------------------

;---------------------- const ----------------------------------

(def const (lambda (s) (and (numberp (car s)) (cdr s))))

(def is-const (lambda (s) (match '((const)) s)))

(def const-in (lambda (s) s))

;---------------------- var ------------------------------------

(def var (lambda (s)
    (If (and (not (numberp (car s)))
             (not (null (car s)))
             (atom (car s))
             (not (member (car s) '(if then else fi while do od ! *eos*))))
     then (cdr s)
     else s)))

(def is-var (lambda (s) (and (not (null s)) (match '((var)) s))))

(def var-in (lambda (s)
    (If (is-var s) then s
     elseif (is-asg s) then (car s)
     else (error "var-in"))))
```

```
(def mk-var (lambda (s) s))

;---------------------- op ----------------------------------------

(def op (lambda (s)
    (If (member (car s) '(+ - * / mod = < >)) then (cdr s) else s)))

(def is-op (lambda (s) (match '((op)) s)))

(def op-in (lambda (s)
    (If (is-expr s) then (cadr s) else (error "op-in"))))

;---------------------- asg ---------------------------------------

(def asg (lambda (s) (match1 '((var) := (expr)) s)))

(def is-asg (lambda (s) (match '((asg)) s)))

(def mk-asg (lambda (lhs rhs)
    (If (listp rhs) then `(,lhs := ,@rhs) else `(,lhs := ,rhs))))

;---------------------- expr --------------------------------------

(def expr (lambda (s)
    (If (and (is-unit (car s))
             (is-op   (cadr s))
             (is-unit (caddr s)))
     then (cdddr s)
     elseif (is-unit (car s))
     then (cdr s)
     else s)))

(def is-expr (lambda (s) (match '((expr)) s)))

(def expr-in (lambda (s)
    (prog (u)
      (If (or (match '((var) := (u := expr)) s)
              (match '(if (u := expr) then (series) else (series) fi) s)
              (match '(while (u := expr) do (series) od) s))
       then (return (simplify u))
       else (error "expr-in")))))

(def mk-expr (lambda (left oper right)
    (prog ()
      (If (atom left) then (setq left (cons left nil)))
      (If (atom right) then (setq right (cons right nil)))
      (return `(,@left ,oper ,@right)))))

;---------------------- unit --------------------------------------
```

```
(def unit (lambda (s)
    (If (or (is-var (car s)) (is-const (car s))) then (cdr s) else s)))

(def is-unit (lambda (s) (match '((unit)) s)))

(def unit1-in (lambda (s)
    (If (is-expr s) then (car s) else (error "unit1-in"))))

(def unit2-in (lambda (s)
    (If (is-expr s) then (caddr s) else (error "unit2-in"))))

;---------------------- if ----------------------------------------

(def if (lambda (s) (match1 '(if (expr) then (series) else (series) fi) s)))

(def is-if (lambda (s) (match '((if)) s)))

(def mk-if (lambda (test-part then-part else-part)
    `(if ,@test-part then ,@then-part else ,@else-part fi)))

;---------------------- while -------------------------------------

(def while (lambda (s) (match1 '(while (expr) do (series) od) s)))

(def is-while (lambda (s) (match '((while)) s)))

;---------------------- series ------------------------------------

(def series (lambda (s)
    (prog (tail)
      (setq tail (stat s))
      (return (If (equal s tail) then s
                  elseif (eq (car tail) '!)
                  then (series (cdr tail))
                  else tail)))))

(def is-series (lambda (s) (match '((series)) s)))

(def series-in (lambda (s)
    (prog (u)
      (If (or (match '(while (expr) do (u := series) od) s)
              (match '((stat) ! (u := series)) s))
       then (return u)
       elseif (match '((stat)) s)
       then (return nil)
       else (error "series-in")))))

(def series1-in (lambda (s)
    (prog (u)
```

```
        (If (match '(if (expr) then (u := series) else (series) fi) s)
         then (return u)
         else (error "series1-in")))))

(def series2-in (lambda (s)
    (prog (u)
      (If (match '(if (expr) then (series) else (u := series) fi) s)
       then (return u)
       else (error "series2-in")))))

(def mk-series (lambda (a b)
    (If (null a) then b
     elseif (null b) then a
     else `(,@a ! ,@b))))

;---------------------- stat ----------------------------------------

(def stat (lambda (s)
    (If (eq (car s) 'if) then (if s)
     elseif (eq (car s) 'while) then (while s)
     else (asg s))))

(def is-stat (lambda (s) (or (is-asg s) (is-if s) (is-while s))))

(def stat-in (lambda (s)
    (prog (u)
      (If (match1 '((u := stat)) s)
       then (return u)
       else (error "stat-in")))))

; Some utility functions for syntax recognition

(def simplify (lambda (form)
    (If (and (listp form) (eq (length form) 1)) then (car form) else form)))

(def head (lambda (short long)
    (segment long (diff (length long) (length short)))))

(def segment (lambda (l n)
    (If (eq n 0) then nil
     elseif (greaterp n 1)
     then (cons (car l) (segment (cdr l) (diff n 1)))
     else (list (car l)))))

(def match (lambda (p s)
    (If (and (atom s) (not (null s)))
     then (eq (match1 p (cons s '*eos*)) '*eos*)
     else (eq (car (match1 p (append1 s '*eos*))) '*eos*))))
```

```
(def match1 (lambda (p s)
    (If (null p) then s
     elseif (atom (car p))  then
       (If (eq (car p) (car s)) then (match1 (cdr p) (cdr s)) else s)
     elseif (eq (length (car p)) 1) then
       (match1 (cdr p) (funcall (caar p) s))
     else
       (prog (z)
       (setq z (funcall (caddar p) s))
       (set (caar p) (head z s))
       (return (match1 (cdr p) z))))))
```

ON THE REQUIREMENTS FOR DYNAMIC SOFTWARE MODIFICATION

Raymond T. Boute
Computer Science Department
Katholieke Universiteit Nijmegen
The Netherlands

ABSTRACT

Dynamic software modification means the replacement or adaptation of software modules while the system is in operation, without impairing its proper functioning. This paper discusses some of the requirements leading towards a systematic method for handling this problem. The primary topic concerns the requirements at the language level, in particular the concept of types and abstract data types in a dynamic environment and the mechanisms required for the dynamic replacement of instances. Following an informal description of the various degrees of abstraction, ranging from axiomatic definition via models to the concrete representation of data types, the relevant issues of type ordering, equivalence, polymorphism and universal operators are analyzed from the viewpoint of dynamic replacement. A secondary topic concerns the support that must be provided by the architecture and the operating system or programming environment for manipulating references to data type instances subject to replacement and for bringing about the actual replacement.

This paper is mostly tutorial in nature. However, even though the problem of dynamic software replacement is as old as automatic computing and many *ad hoc* solutions have been implemented in the past, more systematic strategies have been emerging only recently. Therefore, the views expressed in this paper must be considered as subject to change even in the short term.

1. INTRODUCTION

Dynamic modification of a system means the replacement of software modules while the system is in operation and without impairing the ongoing computations. There are many cases where the ability to perform modifications under such restrictions is a design requirement. Examples are telecommunication and process control systems, on-board computers in spacecraft, large scale traffic control and reservation systems.

Therefore, attempts in this direction have been made since the earliest introduction of computers in applications where continuous service had to be provided. Unfortunately, the strategies used in the past were very configuration-dependent and *ad hoc*, determined by module interface and data representation issues at the lowest levels of the system hierarchy. As a result, any convincing form of validation, not to mention

an attempt at proving correctness, is rendered infeasible in practice, and the resulting solutions are of very limited applicability.

For this reason, several researchers have studied this problem in a more systematic and general framework. The two main lines of work are somewhat complementary, namely the architectural mechanisms needed to support on-line replacement, and the module definition and representation facilities at the language level.

The architectural mechanisms must supply the 'software switch' needed to bring about the replacement. This line of work is best represented by Fabry's paper [1].

The issues at the language or description level concern the strategy of how and when to use such a 'switch' so as to cause no disturbance in the system logic. This includes the definition and the manipulation facilities that must be provided for the modules susceptible to later replacement or modification. Arguments why abstract data types provide the most suitable vehicle in this respect have already been discussed some time ago by Linden [2] in a broader context, namely software modification *in se* (not necessarily dynamic). It will be readily appreciated that the central concerns here are the interface between replaceable modules and the rest of the system, and more importantly the interface between an obsolete module and its replacement. In particular, when the module considered contains internal state information, this must somehow be transferred to its replacement. As pointed out in a paper by Goullon et al. [3], this transfer obviously requires the availability of suitable operations. The additional problem of maintaining consistency during the replacement phase in a fashion that permits use by other system modules concurrently with state information transfer, is discussed by Boute et al. [4]. This reference includes an initial study (with examples) of the extent to which state information transfer is possible without explicit access to the internal representation, and of related problems such as the granularity of the data transfer, information observability and synchronization. It is also indicative of the many problems still remaining.

The present paper is a tutorial discussion of the concepts that support flexible and secure on-line replacement of software modules. It is a reduced version of [5].

The major part (sections 2-4) reviews the high-level aspects of module interface definition in the anticipation of possible replacement, placing them in the broader context of type-oriented programming environments. These sections are organized as follows.

Section 2 starts with a brief initial explanation of the importance of data types in *dynamic programming environments* which will gradually become more apparent in subsequent sections. It further describes the levels of abstraction in module definitions, ranging from traditional *data structures* suitable for representation purposes to *abstract data types* for defining modules in terms of their operations. These concepts provide the basis for specifying modules as replaceable system components.

Section 3 brings up a few related topics that, strictly speaking, are just logical consequences of the concepts introduced in section 2, but whose generality has been somewhat obscured by the restrictive way in which they are embodied in traditional languages. In particular, the concepts of *function* and *type* are restored to the level of generality required in modifiable systems.

Sections 2 and 3 also contain some tutorial material on data types, which may be skipped by readers already familiar with these subjects.

Section 4 discusses the role of *universal operators* in extending the module definition with operations specifically conceived for dynamic replacement.

The last section (5) briefly describes the architectural mechanisms and the support by the programming environment required in performing the actual replacement. This is a rather straightforward matter of using levels of indirection (!), a subject well covered in operating system theory under the headings of memory management and capabilities. A less extensive treatment is therefore deemed sufficient.

The reader must be forewarned that, despite this paper's tutorial nature, the ideas presented here are not all equally well-established, and most may be subject to substantial overhaul even in a short time span. For the present, they can best be viewed as the basic rationale behind the module replacement strategy described in [4].

2. DEFINING AND REALIZING REPLACEABLE MODULES

2.1. Dynamic programming environments

Once written and entered into the system, e.g. by means of an editor, a program traditionally goes through several phases such as compilation, linking and execution, before yielding its intended results. The process by which program statements written in some source language obtain their effect is called *elaboration*. The elaboration of a certain construct or statement is said to be *static* if it takes place at compile time, and *dynamic* if it takes place at run time. It must be noted immediately that this distinction is a consequence of the way in which languages are traditionally implemented. From an abstract viewpoint, the formal definition of the 'meaning' of a program does not involve the concepts of 'compile time' or 'run time', although a distinction can be made between issues that can be resolved from the (source) program text and those that also depend on the input data. From an implementation viewpoint, one can envisage a nearly continuous spectrum of possibilities between interpretation and compilation-cum-execution. In accordance with earlier proposals by Goodwin [6] and Boute [7] intended to eliminate the dichotomy, the issue of 'static' vs. 'dynamic' will be gradually relaxed in subsequent sections.

Assuming for the time being that certain language constructs are elaborated statically and others dynamically, the term 'dynamic' in the context of this paper will specifically refer to the addition, removal or modification of program modules. As we shall see further on, this is tantamount to interpreting 'dynamic' as pertaining to the definition of new data types and instances thereof.

A programming environment can be seen as a system (collection of programs) designed to support the definition and implementation of new programs. In this sense, characterizing a programming environment as 'dynamic' is somewhat tautological, but serves here to emphasize the similarity between programming environments and systems subject to on-the-fly modification.

Beside their importance for programming in general as explained e.g. by Hoare [8], data types are also crucial in dynamic software modification. This role can be illustrated by an appealing analogy with hardware replacement. To a certain extent, 'equipment practice' rules such as the size of printed circuit boards, the shape and number of pins of connectors etc, protect against gross errors when replacing modules. The flexibility of the basic data representation medium in programmed systems, which consists of bits and words, does not provide a similar safeguard: anything fits. A typical example are the early COMMON blocks in FORTRAN, where a collection of contiguous words can be interpreted by the accessing program in any arbitrary way, including overlaps with neighboring words which may belong to other data

items. However, the same flexibility of the basic medium creates the possibility, assuming a good language design, to equip each kind of software module with its own 'personalized' connector, at negligible extra cost. This is one way in which the type concept contributes to handling the dynamic software modification problem. An even more important consideration is that types, in particular abstract data types, constitute a framework for defining and realizing software modules in complex systems, including facilities for replacement. This will be taken up in more detail in the following sections.

2.2. Early data types: domain constructors

Strictly speaking, a *data type* is a set of values called the *domain* (e.g. a set of integers, a set of Booleans), together with a collection of mathematical functions defined on these values, called the *operations* (e.g. +, −, AND, =). The term 'domain' is chosen to indicate that the values are mathematically well-behaved (see e.g. Stoy [10]), but this requirement is usually satisfied in everyday practice or can be enforced by the programming language.

As we shall see in section 2.3, modern languages support user definition of data types to varying degrees. However, early programming languages offered no more than a very limited collection of predefined data types, such as integers and reals with addition, subtraction and multiplication in FORTRAN, and some restricted version of array constructors. Theoretically, this is sufficient to compute anything computable, a rather rudimentary statement applicable to any existing computer or language. However, for most problems, especially when dealing with non-numerical data, this is unacceptably awkward. Languages must support a more hierarchical design style, based on the encapsulation of data items and operations, and their use in building more sophisticated ones, eventually resulting in the final program. A first step towards making languages more suitable for their purpose is the introduction of more general *domain constructors*.

Before proceeding, we briefly establish a few conventions. In conformity with common practice in algebra, we designate a type and the corresponding domain by the same name, because the context usually precludes ambiguity. Thus, given a data type such as *Bool* = ({True, False}; AND, OR, NOT), the name *Bool* can also used to designate the domain {True, False}.
If f is a function, the notation f: A → B indicates that f takes elements from a set A as arguments and maps them onto elements of B. The set A is also called the *domain* of f, but again this double use should not cause confusion. The phrase A → B specifying the domain and the range of an operation is called the *type* of that operation for reasons that will be become clear in section 3.
In later examples, we shall occasionally use the following syntax:

<name> ::= **const** T = <value expression>;
<name> ::= **var** T [= <value expression>];

to introduce a constant or variable name for values in the domain of T. We call the constant or variable an *instance* of type T. For example:

pi: **const** *real* = 3.14159; *x*: **var** *integer*;

The meaning and use of constant and variable names in programs is more precisely described in a separate paper [9].

The term *domain constructor*, borrowed from the highly recommended book on programming languages by Tennent [11], is used here to designate a language construct for defining a new type T based on existing types $T_1, \ldots, T_n$ as follows:

(1) The domain of T is some composition of the domains of $T_1, \ldots, T_n$.

(2) Operations, specific to the constructor, serve to extract or recognize, from a value of type T, the desired component of type $T_1, \ldots, T_n$.

To the extracted or recognized component of type T_i further apply all the operations specific to T_i, unless explicitly stated otherwise.

Examples.

The array and record constructors, available in languages such as ALGOL and PASCAL, are briefly described below in terms of their domains and operations.

A type definition involving the record constructor (**record** ... **end**) as in

type T **is record** s_1: T_1; ...; s_n: T_n **end**;

defines the *domain* of the newly defined type T to be the Cartesian product $T_1 \times \ldots \times T_n$. This means: the set of all n-tuples of the form $(a_1, \ldots, a_n)$ where a_i is in T_i for all $1 \leqslant i \leqslant n$. It further defines a set of *operations*, one for each s_i, called *selection* operations. The selection operation corresponding to s_i and denoted by $.s_i$ is intuitively defined as follows. Given an instance declaration of the form

alfa: **var** T;

then the expression *alfa*.s_i yields (or constitutes) a variable name of type T_i that can be used to designate the *i*th component of *alfa*.

A type definition involving the array constructor (**array** ... **of** ...) as in

type T **is array** I **of** B;

where I is a finite discrete type and B is any type, defines the *domain* of the newly defined type T to be the set $[I \rightarrow B]$. This means: the set of all functions a: $I \rightarrow B$. It further defines an *operation* called *subscripting*, denoted by ( ) or [ ], which is intuitively defined as follows. Given an instance declaration of the form

beta: **var** T;

then the expression *beta*(*i*), where *i* is in I, yields (or constitutes) a variable name of type B that can be used to designate the component with index *i* in *beta*. Here *beta* is viewed as a table containing elements of type B, where each element can be individually designated by supplying its index *i*.

A more rigorous description of these and of other domain constructors can be found in [9]. For our present purpose, it suffices to emphasize the following common characteristic. Although their domains can be defined as abstract value spaces, domain constructors are essentially nothing else than a safer and cleaner form of addressing and representing data structures. The names obtained from the constructor-specific operations, for example *alfa*.s_i and *beta*(*i*), are a language-provided addressing mechanism, replacing the error-prone assembly code that would otherwise have to be used. Other constructors allow the programmer to build complex data structures in still other ways.

However, all of these deal exclusively with representation issues, and not with the properties that are desired for a given application or abstract model thereof. The

programmer constantly has to keep in mind the correspondence between the components of a data structure and the aspects of the model they represent. Since the intended abstract operations of the model are realized by operations on the representation, all parts remain exposed, and it is fairly likely that some programmer some day will disturb the consistency of the data structure by an incorrect program, so that all meaningful correspondence with the model is lost. Also, some modules may rely upon specific representational properties of the data structure unrelated to properties of the model. These representational properties may not hold any more if another representation is chosen later. This would also invalidate the accessing modules, and thus require more modifications than originally intended.

For this reason, domain constructors are insufficient for secure modularization, especially if dynamic modification or replacement is required. The concept of abstract data types overcomes these problems to a large extent.

2.3. Abstract data types

Operations defined by domain constructors can be used only to access the internal *representation* of instances of that type.

An *abstract data type* also consists of one or more domains (previously existing or newly defined) and operations on these domains, but now:

- The operations are user-defined, and reflect the behavior that instances of that type must exhibit in order to perform as the desired model, rather than the representation structure.
- The new domains are not defined explicitly by composing them from old domains, but implicitly by the joint behavior of the values of all domains involved, in the manner defined by the operations.

2.3.1. Definition of abstract data types

The operations are normally defined by means of axioms, with certain restrictions. This is best illustrated by the example below, which is typical for the kind of abstract data type used in programming. It gives the axiomatic definition of a type we call *Bag of* T and which is an adaptation of an example by Guttag & Horning [12].

typemodule *bag*(T: *type*) **is** (* module name *)
  types *Bag* (* new type(s) defined *)
  uses T (* existing types used *)
  operations (* types of the operations *)
    emptybag: $\rightarrow$ *Bag*;
    deposit: T $\times$ *Bag* $\rightarrow$ *Bag*;
    remove: T $\times$ *Bag* $\rightarrow$ *Bag*;
    present: T $\times$ *Bag* $\rightarrow$ *Bool*;
  axioms (* semantic specification *)
  (x, x': T; b: *Bag*) (* i.e. for all x, x', b *)
    present(x, *emptybag*) = False;
    present(x, *deposit*(x', b)) = **if** $x = x'$ **then** True **else** *contains*(x, b);
    remove(x, *emptybag*) = *emptybag*;
    remove(x, *deposit*(x', b)) = **if** $x = x'$ **then** b **else** *deposit*(x', *remove*(x, b));
end typemodule;

Instances of this type are meant to be unordered collections of elements of type T, where the same element can be present more than once. The axioms lend the operators their intuitive meaning:

- *deposit*: yields a new 'bag value' by depositing x in b.
- *remove*: yields a new bag value by removing x from b, or b itself if x is not present.
- *present*: tells whether or not b contains x.

The presence of (T: *type*) in the module definition indicates *type parameterization*, i.e. making this module applicable to various types (discussed in section 3).
A typical example of the use of the above module is:

```
using bags(integer) do (* bags of integers *)
  ourbag: var Bag = emptybag;
  ourbag := deposit(3, ourbag);
  ...
  end
```

It must be emphasized that the values in the domain of *Bag* are implicitly defined by the axioms only, not by an explicit set definition. A fortiori, the representation does not enter into the picture at all.

2.3.2. Implementation of abstract data types

An axiomatic definition as in the preceding example says nothing about the data structure by which values of the defined type are represented internally. Externally they are instances of a new type and can be used only as arguments in the defined operations, not otherwise.

Different languages provide varying degrees of support for the specification and implementation of abstract data types. The following definitions are indicative of the possible levels of abstraction. They should be viewed as marks on a scale, not an exhaustive enumeration.

(1) **Abstract data type**. This implies a type definition in terms of *axioms* for the *operations*, in the sense of section 2.3.1. The *domains* are defined *implicitly* by the axioms.

At present, there exist only very few languages that are satisfied with an axiomatic definition and take it upon themselves to choose a representation and realize the operations. Examples are OBJ, by Goguen et al. [13] and CLEAR, by Burstall and Goguen [14]. Such advanced very high-level languages are mostly experimental and are used in research environments.

(2) **Model type**. Here the type is defined by means of *explicit* (set-theoretical) definition of the *domains*, and a *function* definition for the *operations*.

For the example of section 2.3.1, the domain might be defined as $Bag = T^*$, which is the set of all strings or sequences consisting of elements of type T, including the string of length zero. The operations are then defined as functions on T* and T, e.g. *present*: $T \times T^* \rightarrow Bool$, in such a fashion that the axioms are satisfied. Examples are given in [5] and [9].

Such a *model*, here used in the sense of mathematical logic, can be viewed as an 'abstract realization', and therefore constitutes the first and most important step towards an implementation when genuine abstract data type support as defined in (1)

is not available. Some programming languages e.g. ALPHARD [15], recognize definitions at the model type level as a means for verification.

(3) **Sealed type**. For the application areas where on-the-fly software modification is required (typically in industrial environments), there are several nontechnical reasons such as standardization and support, why commercialized, less powerful languages must be used. Such languages must then at least provide the means for defining the *types* of the operations of an abstract data type. The *effect* (or *semantics*) of the operations can be indicated by comments, although these are not recognized by the language and therefore cannot be used to automatically generate an implementation. Thus, it is up to the designer of the type to decide upon the *representation template*, i.e. the definition (using domain constructors) of the data structure for the internal representation, and to realize the operations by means of function or procedure subprograms which access this representation. The language must enable the module designer to make only these procedures accessible from the outside, and to make the representation inaccessible and invisible. This is called *sealing*, hence the term *sealed type*.

The only commercial (or soon to be commercialized) language that comes somewhat close to this requirement is ADA, designed under contract from the U.S. Department of Defense [16]. In ADA, the corresponding formulation for the previous example would look as follows:

```
generic (type T) package bags is
  type Bag is private;
  function emptybag return Bag;
  function deposit(x: in T, b: in Bag)
              return Bag;
  function remove(x: in T, b: in Bag)
              return Bag;
  function present(x: in T, b: in Bag)
              return Bool;
end bags;
```

Such a package definition, together with comments describing the meaning of the operations (possibly with axioms) is all the programmer-user of the package needs or gets. The indication **private** implies that the user can create instances of this type and use the operations on them, but cannot access the internal representation. The latter is taken care of by the programmer-implementor in a program segment called *package body* which contains the representation template(s) for the private type(s) and the subprogram source code for the operations.

Because a sealed type is incomplete as a definition, it constitutes little more than a secure implementation style. In a package such as illustrated by the above example, one or more newly introduced types are defined as *private*, and instances are explicitly passed as parameters to the operations. For this style, we have introduced the term *type module* [4,17]. If only one new type has to be sealed, the desired effect can be obtained by a different style, making use only of the visibility rules of the language without **private** clauses. This style, which we call *encapsulated data structure*, defines only a *single instance* that behaves in the desired way with respect to the operations, with slightly restricted assignment semantics (implicit). It is *not* a type from which many such instances can be created; in ADA, it does not even have a type. The

package definition contains only the operation types, where the instance is omitted as a parameter because the package itself is the instance. An internal variable declared in the package body using a suitable representation template can be viewed as representing the *state* of the package. It is not visible outside the package body.

In an orthogonal language, the distinction between type module and encapsulated data structure turns out not to be an essential one [18], at least in principle. However, in the context of dynamic software modification, it raises a small implementational issue at the machine representation level, which is taken up again in section 5.

(4) **A data structure** is simply the 'naked' representation of a domain, either of a built-in type of the language or defined with the aid of domain constructors. It is possibly complemented by a collection of procedures implementing the operations, but this does not exclude any other form of access to the components of the structure. For *Bag*, the representation template can be an array, a sequence, a linked list, etc. This is the level of support provided in traditional languages such as PASCAL.

2.3.3. Note on representations

Representations raise an important question concerning the definition of equality for values of an abstract data type. For example, using sequences as a representation, b_1 = (a k 3 z 3) and b_2 = (3 a z 3 k) represent bags that are equal for all intents and purposes as defined by the axioms, yet are different sequences. Strictly speaking, the meaning of equality depends upon the axioms, and requires that the programmer implement it as a separate operator.

If such difficulties already arise when only a single representation template is used for a given data type, they are certainly compounded in case different templates exist simultaneously for the same type, as may happen during modification. This issue is taken up again in the discussion on universal operators in section 4.

3. TYPES IN AN ORTHOGONAL ENVIRONMENT

In view of section 2.3, a program or module is just an instance of an abstract data type and, since modules can be composed hierarchically, the same applies to an entire system. Hence no generality is lost by characterizing a programming environment in terms of the way it handles data types.

Orthogonality implies that no arbitrary restrictions are imposed on the fashion in which concepts may be combined. The advantage is economy of concepts, where the necessary expressive power is obtained by judicious combination. In the next few paragraphs, it will become apparent that the properties needed in a flexible programming environment or a dynamically modifiable system are precisely those achieved by orthogonality, see e.g. [4], [7] and a report by Heering & Klint [19]. This fact should be borne in mind as the basic rationale for this section.

If in an environment new programs and modules are introduced, it simplifies matters if their names can be treated just like other constant or variable names designating instances of data types, rather than being subject to a separate set of rules [7].

Every self-contained part of a program (or 'syntactic unit') with which operations are associated, as required for the program or for the environment, should belong to some type. This holds in particular for types themselves. The logical consequences of this observation will be the main topic of this section.

Sections 3 and 4 are closely related to the question whether certain operations meant to be universal can indeed be defined for all data types.

3.1. Function and procedure subprograms

Since the concept of function is probably the single most important abstraction mechanism in programming, it would appear reasonable to support this concept equally well as, for example, data structures. In particular, a function f: $A \rightarrow B$ has a type, namely $A \rightarrow B$, which f shares with all other functions from A to B. As mentioned earlier, the *domain* of this type is $[A \rightarrow B]$. An orthogonal language should enable the user to write expressions for defining, and names for further designating such types, if necessary with some added syntactic sugar as in the following example:

```
type twoarg is function (A,B) returns C;
 (* domain [A × B → C] *)
type realfun is function (real) returns real;
square = const realfun = (* function expression *)
```

Unfortunately, many languages including ADA and PASCAL do not provide this level of support and give 'second-class treatment' [10] to functions. The only form of definition is that a function name can be bound to a function subprogram as a single (constant) instance, whose type cannot be described or referred to in the language, e.g.:

```
function square(x: real) returns real
  is return x**2 end;
```

With this level of support, functions cannot be passed as parameters, for instance. This not only very restrictive when formulating a program, e.g. when one wants to use a functional style, but even more so in a dynamic programming environment where one wants to manipulate functions as items subject to replacement just as 'passive' data.

Theoretical study and implementation experience in connection with languages such as LISP (functions as variables) and ALGOL 68 (function types) has satisfactorily overcome the practical problems formerly associated with function variables. Some theoretical difficulties that prohibit full definitional support of functions are mentioned in section 4.

Special attention is required in case the replaceable module to be defined is an 'impure function'. In a function realization as a piece of program involving expressions, reference is made to three kinds of constant or variable names: local names for internal use, formal parameters designating arguments and results, and names defined in the surrounding environment. If names of the third kind are present, in which case we are dealing with an 'impure function' or a procedure, provision should be made for these names in the function type definition. The most straightforward way to do this is by means of a record type definition, because an environment or part thereof can be seen as an instance of a record type and vice versa. In a dynamic environment, such a *free variable record* is the only means to provide a full interface definition as is necessary for secure replacement.

3.2. The type *type*.

Mathematically speaking, a type in its purest sense is an algebra [12], that is, it consists of sets of values and operations. An algebra itself constitutes an abstract value in some larger value space, the space of algebraic structures. Practically speaking, in programming languages one handles not only instances of data types (integers, records, bags), but also the types themselves. For example, a domain constructor can be viewed as an operator taking types as arguments and returning a new type as a result.

It is therefore reasonable to consider types themselves as elements in the domain of types. A fringe benefit is that we obtain some uniformization in syntax, viz. one can write:

complex: **const** *type* = **record** *x*: *real*, *y*: *real* **end**;

just as for other (constant) values, instead of

type *complex* **is record** *x*: *real*, *y*: *real* **end**;

The most significant benefit in programming is that types can be passed as parameters to functions. Two important concepts, present in some advanced languages, emerge as special cases of this view:

(1) **Type parameterization** (a type as output parameter). This is a generalization of viewing a domain constructor as a function returning a type. For example, we can define the 'abstract data type generator' *intsequence* as a function:

intsequence: N → *type*

such that *intsequence*(10) is an abstract data type for handling sequences of integers where an upper limit of 10 is imposed on the length. The semantics of the operations for *intsequence* such as *append* are assumed to take this limit into account.

(2) **Polymorphic data types** (a type as input parameter). In 'closed' programs, but even more so in a dynamic environment, there are many data types whose usefulness extends beyond the data types for which they are originally intended. More precisely, the semantics of the operations may depend only on a proper subset of the properties of the data type, rather than on complete information concerning these types themselves. It is therefore useful to have a function whose argument may be any type with the required properties, returning the desired data type, or instance thereof depending on how this function is defined.

An application has already been illustrated by parameterizing bags in section 2.3.2. Note that doing so in ADA requires a special **generic** clause rather than just an orthogonal interpretation of the function concept, which results in loss of generality as well as some undesirable implementation consequences [17]. In particular, it falls short of the degree of polymorphism that is desirable in dynamic environments. Another illustration is the extension of the example in (1) to

sequence: *type* × N → *type*

where the base type (i.e. the type of the items in the sequence) is supplied as a parameter.

Further generalizations, such as those obtained by using partial parameterization, are discussed in [9]. Additional information on polymorphism can be found in the

papers by Milner [20] and by Demers and Donahue [21], and Swierstra's Ph.D. thesis [22].

Polymorphism is important whenever during the design of a module it is not fully known which types will be passed to it by other modules designed at a later date. It provides a more controlled and secure way of coping with this situation than the indiscriminate use of 'total types' [6] whenever only partial information about the type is available.

3.3. Type compatibility

This issue, which has so far been postponed, depends on the definitions of equality and ordering.

(1) **Type equality**. Intuitively speaking, two types are equal if they define the same domains and the same operations. It must be possible to decide this so-called *structure equivalence* from the type definitions. More precisely, equality for types defined via domain constructors has to be specified in the language definition by a set of mutually recursive rules, as in ALGOL 68. For abstract data types, which constitute a more general case than domain constructors in the sense that they are equivalent to theories in formal logic, the derivation of rules for type equality must be based on the corresponding mathematical concepts, that is, on unification algorithms and confluent reductions [23]. This problem will come up again in section 4.

In certain circumstances, the programmer may wish to define a domain as being distinct from every other domain, if otherwise they might intersect or be equivalent. This can be achieved by means of a separate domain constructor which, given a type T, creates a type that has the same structure but is different by definition. An example is:

T': **const** *type* = **distinct** T;

In some languages, e.g. ADA [16], the convention is that every occurrence of a type definition defines a distinct type. This is called *occurrence equivalence* or, somewhat misleadingly, *name equivalence*. This scheme is needlessly restrictive, first of all for conceptual reasons (unorthogonal, lack of generality), but also because in a large dynamic environment equality of types in different modules can then be obtained only by reference to the same type name, with corresponding definition, in a common global type library. Even with hierarchical name structures, this reduces the flexibility of the name space.

(2) **Type ordering**. The following concepts are a direct extension of the structural view of types. In languages with occurrence equivalence for types, they will be embodied only in a very restricted form.

Assume that type A is 'more general' than type B in the sense that

(a) the domain of A includes the domain of B

(b) the collection of operations of B includes the collection of operations of A

where the inclusion need not be proper. Then it is conceptually reasonable to consider a value of type B to be also of type A because

(a) it is in the domain of A

(b) all operations defined for type A are applicable to this value.

In this case, B is said to be a *subtype* of A, or A a *supertype* of B. This implies that a value of type B may appear whenever a value of type A is expected, for instance a value of type B may be assigned to a variable of type A.

Already in a static situation this convention has numerous advantages, because it closely mirrors the mathematical relationship between the domains. In particular, it eliminates many troublesome issues regarding coercion and operator overloading [5,9]. Mathematically speaking, B is said to be a subtype of A if the algebra B can be *embedded* in the algebra A. The real and complex numbers are a classical example.

In a dynamic environment, this view of type ordering supports very well the safeguarding of secure type compatibility when the collection of operations of a module is being changed or, strictly speaking, when the module is replaced by another module having the same domain but another set of operations. The atypical case in this respect is the removal or modification of individual operations. This implies that other modules, namely those using the affected operations of the considered module, are in fact being modified in the context of a more pervasive redesign. The typical case is the extension of the set of operations because new features or new modules requiring other services from the considered module are being added. The updated version is then a subtype of the old one. Type ordering thus guarantees that the constant or variable declarations and the application of operations in older parts of the program remain valid without modification.

(3) **Types related through predicates.** The domain constructor concept can be generalized to encompass type definition by means of a predicate, i.e. a logical expression which the elements of the defined domain must satisfy. Syntactically such a type definition might be written as:

T: **const** *type* = T′ **satisfying** P(*val*);

where *val* is a language-defined identifier designating any value of the type being defined. Such an expression defines the domain of T to consist of all x in T′ that satisfy P(x). For example, the domain of

integer **satisfying** *even*(*val*);

where *even*: *integer* → *Bool* is a programmer-defined predicate, consists of the even integers only.

A special case which is particularly useful in the context of this paper is when T′ = *type*. A predicate can be used to restrict the values of type *type* that may be passed as a parameter to a polymorphic function or data type. For example, assume that *oper* is a language-defined predicate such that *oper*(T, x) returns True if and only if the operator x is defined for values of type T. Then the expression

type **satisfying** *oper*(*Val*, *greater*: *Val* × *Val* → *Bool*);

may be used as a definition for the set of those types that are meaningful type parameters for a polymorphic sort function. The responsibility of ensuring that *greater* is indeed an ordering relation (reflexive, transitive, antisymmetric) obviously resides with the programmer, at least with present-day programming technology.

The use of predicates that impose only restrictions of direct relevance, provides the required flexibility for the addition of modules to a given configuration, and at the same time maintains the security associated with type checking.

4. UNIVERSAL OPERATORS

An operator is called *universal* if it is defined for all types. The most common universal operator is *assignment*. The view that assignment must indeed be considered as an operator is given more substance in [9]. It is possible that, implementation-wise, assignment can be subject to optimization, but the abstract meaning of this operation can be universally well-defined and supported by the language for all types, independently of any representation.

The next common universal operator is equality. It cannot be universally defined nor supported by the language, because equality depends upon the axioms. The sequence representation for the type *Bag* in section 2.3.3. already illustrates this difficulty for 'ordinary' domains, i.e. those which can be readily modeled as a data structure. The solution for this particular case consists in finding a canonical representation, i.e. such that equal values have equal representations (the converse is already satisfied). For bags of integers, this could be achieved by ordering the elements in the sequence representing a value of type *Bag*. Note that this is a suggestion for how to implement the operator *equal*: $Bag \times Bag \rightarrow Bool$, not a definition based upon the axioms. The equality problem arises in a more general context for functions and, as a consequence, for operations and abstract data types. In mathematics, the definition of equality of functions is defined *extensionally*, that is, functions are equal if they yield equal results for equal arguments. In the most general case this cannot be decided from the program text. Again, equality of functions in a higher-order theory (i.e. dealing with functions as first-class entities) requires the existence of canonical representations, which in turn depends upon confluence and unification algorithms [23]. Therefore, equality of functions can be supported only by a language which is built in conformity with such a higher-order theory. This holds a fortiori for the equality of abstract data types, which are higher-order theories themselves.

This digression illustrates the fact that theoretical problems are associated even with seemingly innocent concepts such as equality, and that therefore universal operators cannot in general be supplied by the language. It is up to the programmer to implement them separately for each data type. In particular, this is the case for the universal operators required for software modification, discussed next.

In an earlier paper [4,7], we used the term *type-specific operators* to designate operations specific to the semantics of the defined type in the context of the program, and *reconfiguration operators* for those required by the environment for dynamic modification. These reconfiguration operators must be universal so that the system programs that supervise and effect replacement do not depend on the details of the replaced module. The goal is that all modules can be replaced using the same external strategy, modulo a small, fixed number of variants, such as: whether the replaced item is a function, a type module or an encapsulated data structure. Only within such a framework can replacement be implemented with extra operations on a per module basis, i.e. $O(n)$ only, rather than with additional operations for every pair of modules, i.e. $O(n^2)$ [3].

To decide which universal operators are necessary and sufficient for dynamic modification, considerable additional research is needed. Moreover, the question

strongly depends upon the system requirements, i.e. the kind of modifications to be coped with, and the replacement strategy chosen. A previous study [4] indicates that the following constitute a reasonable set, at least for the strategy assumed there:

- An enumeration type whose values indicate the status of the module, e.g. *nonexistent*, *uninitialized*, *directly replaceable* (internal state need not be copied), or *delegating* (see below).
- A collection of operations to obtain the module's status, to mark the module for replacement (which affects the module's status in a controlled fashion), to supply a delegation module, to transfer state information from the old module to the new module, and to effect the actual replacement. In many cases, exclusion is needed to ensure proper synchronization between state transfer and external use by other modules. Unless the language permits implementation of exclusion as part of the other operations, as in [4], the exclusion operations must also be classified in this category.

The replacement strategy in [4] is based on the concept of *delegation*. Formulated in terms of the encapsulated data structure style, this means the following. If a module is to be replaced by a new module, and while transfer of state information is taking place, all operation invocations by other modules are 'routed' to one of them, via the revocation mechanism mentioned in section 5. However, the actual execution of such an operation may be delegated to the module that possesses the relevant part of the state information. This goes on until the new module contains all the required state information, and the actual replacement can take place. It is an interesting exercise to reformulate this strategy for the type module style of definition.

In [4], the embodiment of this strategy is such that the implementation of the delegate operator in the old module is granted temporary access to the public operations of the new one for updating the latter's state. It further has access to the internal representation of its own module, which may be necessary to obtain the state information. Recall that, in the typical case, the type of the new module is a subtype of that of the old module; therefore the new module can be passed as a valid parameter to the delegate operator.

Whether or not access to the internal state representation is required depends on whether this state is *observable*. Observability means that the state can be deduced from the values returned by type-specific operations, when applied in some judicious order. It is clear that this state information need only be obtained up to equivalence, because different representations can designate the same abstract state or value (see section 2.3.3). For the same reasons, the type-specific operations may not suffice to observe the state. For example, it is in general not possible and usually not practical to deduce the 'contents' of a bag by means of the available operations (*present*, *deposit*, *remove*). However, it is always possible to define an additional type-specific operation that permits observation of the (abstract) state in a representation-independent fashion. This can be proved using a so-called *history file* consisting of the names of the operators and corresponding actual arguments over the past history of the module, which have brought the module to its present state. Formal language theory provides the means for reducing such history files considerably. With suitable protection mechanisms [4], access to the operator returning history files can be granted to *delegate* operations exclusively.

The same concept of history file can be used to prove that *controllability*, i.e. the

ability to bring a module into any state reachable by type-specific operations only, is always satisfied as a condition.

The transition from these existence proofs to an optimal or even practical derivation of an abstract value equivalent to a history file, is clearly nontrivial.

5. ARCHITECTURE AND OPERATING SYSTEM SUPPORT

The preceding sections dealt with the higher-level issues in dynamic software modification, in particular the concepts that ideally should be supported by higher-level languages to obtain a good balance between flexibility and type security of on-line module replacement. As we have seen, there are still many unresolved issues at this level.

The present section briefly discusses the lower-level mechanisms in the computer architecture and the operating system that are helpful to implement the required facilities. Evidence thus far seems to indicate that the mechanisms for supporting data types and memory management proposed for advanced computer architectures, adequately serve our purposes also. The most relevant concept is that of *capabilities*, which are extensively discussed in the literature, for instance in a paper by Fabry [24] and in a tutorial paper presented at the 1981 EUROMICRO conference [25]. Therefore we can limit ourselves to the following summary overview.

5.1. Using capabilities for dynamic modification

Capabilities have two separate aspects [25], namely addressing and protected type support, which lend themselves well to being discussed independently from each other.

(1) **Capability addressing** is a flexible and powerful means for providing a virtual address space which can cope simultaneously with dynamic sharing and relocation of data structures [24]. For each instance of a data structure, called an *object* in this context, an *object descriptor* defines the starting address or *base address*, and the *size* of this instance in the linear address space provided by the memory hardware. All descriptors are grouped in a single *descriptor table*. Further, each module has its own *capability list*. Each entry or *capability* in this list is an index into the descriptor table, indicating the descriptor for an object that may be accessed by the considered module. The *logical address*, i.e. the address that a module can use in an instruction to reference an object, is then an index into the module's capability list. The capabilities available in such a list and the order in which they appear constitute the logical address space, which may be different for each module. Relocation, i.e. moving an object to another place in physical memory, requires the updating of only a single descriptor in the descriptor table. The access path from logical address to object is thus:

$$\text{logical address} \rightarrow \text{capability} \rightarrow \text{descriptor} \rightarrow \text{object}$$

The first two are associated with the accessing module, the last two with the accessed object.

The indirection level introduced by the descriptor is not only a flexible mechanism for relocation, but also for replacing the object by a new version without the need for recompiling or relinking the modules that have access to it. This is a special case of modifying the access granted by a capability, an action called *revocation*. It is is studied in Redell's thesis [26]. In this respect, capabilities constitute one of the basic

mechanisms for dynamic modification.

Remarks.

(a) Capability lists and descriptor tables are also considered as objects in such an architecture. This yields a uniform addressing mechanism. Only those modules that have a capability for the descriptor table can modify descriptors. Such capabilities are only granted to operating system modules such as those that supervise replacements.

(b) In addition to base address and size, a descriptor may contain a synchronization structure that can be used by modules having the required capabilities to do so, e.g. for temporarily excluding access by other modules while piecewise modification or state information transfer is taking place. The strategy of locking and replacing parts of objects must be carefully designed so as to avoid both deadlocks and unwanted access. This topic is briefly treated in [1] at the architectural level and in [4] at the language level.

(2) **Capability type support at run time.** Assume that each object descriptor additionally includes a reference to a *type description*, characterizing the type of the given object, and that each capability includes a *rights description*, characterizing the way in which this capability can be used to access the object under consideration. The type description itself may be a representation of the parse tree for the type definition in the source text, together with some semantic attributes, or it may be reduced to a label unique for the type under consideration, or it may be omitted completely depending on the language and the requirements.

Similarly, the rights description associated with a capability may in some cases be a simple tag, which is checked by the hardware by comparing it with the object tag. A more refined rights description is the representation of a list of operations that may be applied to the object referenced by a capability, whereby different capabilities may grant different degrees of access to the same object. This is precisely the mechanism needed for differentiating between access to type-specific operations by other (user) modules defined in the program, and access to the replacement operations by system modules that possess the required rights. Note that an identical rights and type mechanism is also used at the machine level for marking and accessing architecture-defined (as opposed to user-defined) data structures.

One must be very cautious with this kind of architecture: the need for run-time type support is an order of magnitude smaller than computer architects often seem to think. Since most type issues can be resolved completely statically from the program text, there is plenty of room for optimization, in the sense that most overhead can be factored out.

We conclude this architectural section with a brief discussion of the representation of objects from the viewpoint of dynamic modification. An abstract data type can be represented by an object which is a capability list for the corresponding operations. Other modules that might use the type-specific operations can reference these operations only in a 'call subprogram' instruction, as enforced by the rights mechanism. The objects representing the operations themselves are usually code segments, an architecture-defined type tagged as such.

With the type module style of definition, the type description referenced by the object descriptors of the instances must include the template description or suitable encoding thereof, in view of possible representation modification. Such information

may be necessary to select the matching implementation of the invoked operator, in case several templates exist for the same type, e.g. during modification. This reflects the dual aspect of the instance: an abstract value to the user modules, a data structure to the operations.

With the encapsulated data structure definition, the situation is somewhat simpler because every instance contains the list of operations as the part accessible by the user modules, the state information data structure being the inaccessible part. The implementation of the operations in this list is thus always associated with the correct representation template, because together they constitute a single unit.

5.2. Operating system support.

First of all, the operating system is intended to bridge the gap between the mechanisms required by the language implementation and those actually available in the architecture. Thus, the operating system kernel has to provide the mechanisms not directly available in the architecture.

Next, the environment must contain utilities to bring about the desired replacements. These utilities are implemented as an operating system module that has access to the universal reconfiguration operators of all modules in the system. Its external interface consists of a set of operations available to the user, through which the replacement information is supplied.

(1) In a programming environment, these operators probably need consist of little more than a controlled access to the replacement operators of the various modules. Replacement itself is then achieved by the interactive interpretation of source language statements, e.g. by means of an assignment or procedure invocation. Examples are given in [4] and in a paper by J. de Man [27].

(2) In an embedded system, often no language interpreter or compiler is available. Replacement then typically amounts to reading from a tape or downloading via a data network, followed by dynamic linking of the new module, and eventually the desired replacement. The utilities for replacement then consist of two parts:

- One part resides in the manufacturer's software distribution system, and has access to configuration data, module version numbers, etc. for all systems in the field. On the basis of this information together with the replacement data supplied by the programmer, a *replacement load file* is generated. This contains the object code of the new module, and replacement information such as which variant of the replacement strategy must be used and whether or not representations must be converted on demand or when used.
- The other part resides in the system in the field, and is in fact only a more advanced version of a dynamic linker/loader. It has access to the information in the replacement load file, and to the universal replacement operators of all modules in the system. On this basis it automatically performs the dynamic relinking and replacement in the same way as the programmer under (1) might have done.

The detailed embodiment of these utilities, especially for case (2), depends very much on the particular policy adopted for the environment and on the organization of the configuration data base.

CONCLUSION

This paper constitutes a tutorial on the requirements for dynamic software modification at two levels: the language level and the machine/operating system level. Not every single one of these concepts is necessarily a strict *conditio sine qua non* under all circumstances. Compromises both in security and in flexibility may always be made when justified by the application. Some fundamental problems concerning representation equivalence and observability clearly require further study.

REFERENCES

[1] Fabry, R.S., "How to design a system in which modules can be changed on the fly," *Second International Conf. on Software Engineering*, 1976, pp. 470-476.

[2] Linden, T.A., "The use of abstract data types to simplify program modifications," *Proc. Conf. on Data: Abstraction, Definition and Structure*, *SIGPLAN Notices*, **11**(1976), 6, pp. 12-23.

[3] Goullon, H., Isle, R., & Löhr, K.-P., "Dynamic restructuring in an experimental operating system," *Third International Conf. on Software Engineering*, 1978, pp. 295-304.

[4] Boute, R.T., De Man, J., & Peeters, H., "Secure on-the-fly software modification," *Fourth International Conf. on Software Engineering for Telecommunication (SETSS)*, 1981, pp. 49-53.

[5] Boute, R.T., "Language and system support for dynamic software modification," Internal Report, Department of Computer Science, Katholieke Universiteit Nijmegen, May 1982.

[6] Goodwin, J.W., "Why programming environments need dynamic data types," *IEEE Transactions on Software Engineering*, **SE-7**(1981), 5, pp. 451-457.

[7] Boute, R.T., "Building a uniform programming environment based on data abstraction," *International Computing Symposium (ICS81)*, 1981, pp. 415-424.

[8] Hoare, C.A.R., "Structured data types", in: Dahl, O.-J., Dijkstra, E.W., & Hoare, C.A.R., *Structured Programming*, Academic Press, 1972.

[9] Boute, R.T., "Intuitive programming language semantics," Course notes, Cours Postgrade en Informatique Technique, Ecole Polytechnique Fédérale de Lausanne, 1982.

[10] Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, 1977.

[11] Tennent, R.D., *Principles of Programming Languages*, Prentice-Hall, 1981.

[12] Guttag, J.V., & Horning, J.J., "The algebraic specification of abstract data types," *Acta Informatica*, **10**(1978), pp. 27-52.

[13] Goguen, J.A., & Tardo, J.J., "An introduction to OBJ-T," *IEEE Conf. on Specifications of Reliable Software*, 1979, pp. 170-189.

[14] Burstall, R.M., & Goguen, J.A., "The semantics of CLEAR - a specification language," Internal Report CSR-65-80, University of Edinburgh, February 1980.

[15] Wulf, W.A., et al., "Abstraction and verification in ALPHARD: Introduction to language and methodology," Report, Carnegie-Mellon University, June 1976.

[16] U.S. Department of Defense, *Reference Manual for the ADA Programming Language*, Government Printing Office, Washington, July 1980.

[17] Boute, R.T., "Simplifying ADA by removing limitations," *SIGPLAN Notices*, **15**(1980), 2, pp. 17-28.

[18] Geschke, C.M., & Mitchell, J.G., "On the problem of uniform references to data structures". *SIGPLAN Notices*, **10**(1975), 6, pp. 31-42.

[19] Heering, J., & Klint, P., "Towards monolingual programming environments," Report IW 185/81, Mathematisch Centrum, 1981.

[20] Milner, R., "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, **17**(1978), pp. 348-375.

[21] Demers, A.J., & Donahue, J.E., "Data types, parameters and type checking" and "Type completeness as a language principle," *Conference Record of the Seventh Annual ACM Symposium on the Principles of Programming Languages*, ACM, 1980, pp. 12-23 and pp. 234-244.

[22] Swierstra, S.D., *LAWINE, an Experiment in Language and Machine Design*, Thesis, Technical University of Twente, 1981.

[23] Huet, G., "Confluent reductions: abstract properties and applications to term rewriting systems," *JACM*, **27**(1980), 4, pp. 797-821.

[24] Fabry, R.S., "Capability-based addressing," *CACM*, **17**(1974), 7, pp. 403-412.

[25] Boute, R.T., "Foundations of the next-generation microprocessors," *Proc. 7th EUROMICRO Symposium*, 1981, pp. 271-287.

[26] Redell, D.D., *Naming and Protection in Extendible Operating Systems*, Thesis, University of California, Berkeley, 1974.

[27] De Man, J., "Experiments with dynamic software modification," *Proc. 8th EUROMICRO Symposium*, 1982, pp. 273-278.

FORMAL SPECIFICATION AND IMPLEMENTATION OF OPERATIONS IN INFORMATION MANAGEMENT SYSTEMS

Erik Sandewall
Software Systems Research Center
Linköping University
Linköping
Sweden

ABSTRACT

Among information management systems we include general purpose systems such as text editors and data editors (forms management systems) as well as special purpose systems such as mail systems and computer based calendars. Based on a method for formal specification of some aspects of IMS, namely the structure of the data base, the update operations and the user dialogue, this paper shows how reasonable procedures for executing IMS operations can be written in the notation of a first-order theory in such a way, that the procedure is a logical consequence of the specification.

1. SPECIFICATION OF IMS AND THE FORMAL IMPLEMENTATION PROBLEM

Information management systems (IMS) include general-purpose systems such as text editors and data editors (e.g. forms management systems), and special purpose systems such as mail systems and computer based calendars. An IMS provides an interactive service for 'moving data around' or 'general housekeeping': entering data into the computer; changing it; displaying part of the data through a 'window' on the screen while it is being entered or changed; rearranging it and printing it out on various media.

The computing world abounds with IMS: there are IMS for various kinds of information (text, structured data, graphical data, etc.) as well as for various applications. As Boehm has shown [BOE80] the implementation of an interactive application program consists to a large extent of implementing a number of IMS services. In the academic environment as well, every hacker writes his own editors: not just once, but often several times. Programming environments, which are presently an area of high research interest, are IMS for software.

In spite of this proliferation of IMS, there is a striking lack of systematization or formal understanding of what this kind of software really does. In fact, the lack of formal understanding is probably one reason for the proliferation: various IMS perform

This research was supported by the Swedish Board of Technical Development.

very similar tasks and it is reasonable to believe that there could be a design which is simpler and more powerful at the same time. A formal specification of IMS can serve this purpose if it is clear and easy to understand, and if it can accommodate the application specific details as well as the generalities that are present in all IMS.

In a previous paper [SAN82] we have described a method which allows us to *express in precise terms* the services which are provided by actual IMS (both those services which are characteristic for most IMS as well as a framework for characterizing application-specific services). In particular, the following things are characteristic for IMS (see [SAN82]):

- the existence of a *data repository* where information is stored;
- *update operations* on the stored data;
- the existence of a *focus of attention* (often represented as the cursor position) relative to which the edit operations are performed;
- *display operations* on the screen - involving a traversal of the structure at hand so that its parts can be displayed individually, and layout planning so that the whole display makes sense;
- dialogue interpretation, particularly the *common command loop*. An additional problem is the *prompting* situation where the user is supposed to provide an answer to a question or a specific piece of data (in data entry). However, even in prompting situations the user must be able to override the context by entering special commands (often implemented using control characters).

Our approach to IMS *separates* the specification of the contents of the display from the specification of the effects of operations on the data repository.

The topic of the present paper is to describe a method for transforming the specification of a set of operations (i.e. of their effect on the data repository) into a procedure which executes them. We use the term *implementation* for the transformation from a specification to a procedure which satisfies it. In the present paper, the resulting procedure is expressed in a simple language which we introduce here, but which has the characteristic properties of conventional programming languages: local variables, assignment statements (although using 'single assignment'), operations which 'update' the data repository, conditionals, recursion, etc. At the same time, the language for expressing procedures is chosen from a restricted first order theory, and the relation between the specification and the implementation is one of logical consequence. The paper contains specifications of the proposed languages, as well as some examples of how an operation can be implemented in the language.

2. NOTATION

In order to write specifications for the IMS operations, we must first define the domains for data structures in the date repository of the IMS. Following the advice of Blikle [BLI82], we shall express our domain equations in set theory notation. The notation will therefore be the standard notation of predicate logic and set theory, with only a small number of notational extensions which agree with Blikle and/or agree roughly with the practice of the denotational semantics literature. We use the following notation:

Predicate logic

$\wedge$ and

$\vee$ or

$\Rightarrow$ implies

$\equiv$ logical equivalence

$\exists$ existential quantifier

$\forall$ universal quantifier (free variables are viewed as universally quantified)

Set theory

$\cup$ union of sets

$\in$ membership in sets (particularly domains)

$\subset$ subset relation between sets

$<x,y,\ldots>$
The sequence whose members are x, y, etc. $<x>$ is distinct from x. $g.x$ is the result of applying the function g to the argument x. Also written $g[x]$. For functions of several arguments only the latter notation is used.

hd $hd.<x_1,x_2,\ldots,x_n>=x_1$

tl $tl.<x_1,x_2,\ldots,x_n>=<x_2,\ldots,x_n>$

pfx $pfx(x,<x_1,\ldots,x_n>)=<x,x_1,\ldots,x_n>$

conc concatenation of sequences:
$<x_1,\ldots,x_k>\ conc\ <x_{k+1},\ldots,x_n>=<x_1,\ldots,x_n>$.
This operation is extended to sets of sequences in the obvious way:
$A\ conc\ B=\{a\ conc\ b \mid a\in A \wedge b\in B\}$.

rev reversal of sequences

subst substitution in a sequence structure: $subst[old,new,x]$ replaces every occurrence of *old* by an occurrence of *new* in the structure x.

$\times$ Cartesian product: $A\times B\times\ldots\times D$ is defined as the set of all $<a,b,\ldots,d>$ where $a\in A$, etc.

! set of onetuples: $A!=\{<a> \mid a\in A\}$

* $A^*=\{<a_1,\ldots,a_n> \mid a_i\in A \wedge n>0\}$

$^+$ $A^+=\{<a_1,\ldots,a_n> \mid a_i\in A \wedge n>1\}$

$\rightsquigarrow$ pseudo-mapping: $A\rightsquigarrow B \mid e$ is the set of all total functions from A to B such that $f[a]\neq e$ for only a finite number of arguments. If f is a pseudomapping for which e is *nil* we shall not distinguish it from the set of all pairs $<x,f[x]>$ where $f[x]\neq nil$. In particular, no distinction will be made between the pseudomapping which maps everything to *nil* and the empty set.

Named domains and selector functions

The symbols introduced above are used for writing domain equations for a collection of *named domains*. Often a domain A is defined by an equation of the form $A=B\times C\times D\ldots$. The components of a member of A can then be selected using the functions *hd* and *tl* defined above. It is convenient to introduce the following, more mnemonic notation:

$s-$ $s-b$ is a function $A \rightarrow B$ which decomposes an object in A and determines its B component (assuming there is exactly one such component), and similarly for $s-c$, $s-d$, etc. Thus with the given definition for A:

$$a \in A \Rightarrow s-c.a = hd.tl.a .$$

3. HIERARCHIES

In characterizing an IMS the structure of the information in its data repository is specified first and the operations on that structure next.

As fundamental information structure we use the *hierarchy*. This is a tree having data elements (integers, strings, or entities) as leaves. All of its nodes (both leaves and branch points) are associated with a set of attributes, each attribute being a pair of a name and a value. The following domains will be used:

M *Atoms* or data elements are objects which are indivisible from the point of view of this theory.

E *Entities* are atoms which are used for fixed purposes in the information structure. They will be written like identifiers in programming languages, e.g. *john*, *red*, *linenumber*. We have $E \subset M$. The distinguished object *nil* is a member of M but not of E, and is identified with the empty sequence and the empty set. Integers and strings are also examples of atoms which are not entities.

The composite domains: T (trees), H (hierarchies), A (attributes), and L (labels), are defined by the following equations:

$$\begin{aligned} H &= \{h\} \times L \times (T \cup M) \quad . \\ T &= H^* \\ L &= E \rightsquigarrow M \mid nil \\ A &= E \times M \ . \end{aligned}$$

Here h is a symbol which is not otherwise used. It serves as a mark on each hierarchy. The definitions mean that:

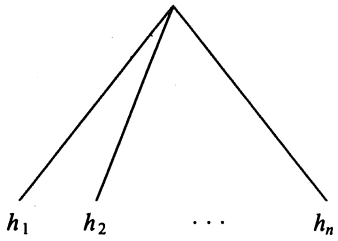
A *tree* is a sequence of hierarchies. In particular, *nil* is a member of T.

A *hierarchy* is formed from two elements, where the first one is a label and the second element is either an atom or a tree.

A *label* is a pseudo-mapping from entities to atoms, i.e. a certain set of attributes. (In other papers on the *use* of IMS we shall in fact need non-atomic attribute values in labels, but the above definition is sufficient for our present purpose).

An *attribute* is formed from two elements, where the first one is an entity and the second one is an atom.

We shall use a graphical notation for these structures, where atoms are written as text. A tree which is a sequence of hierarchies is written as follows (figure 1):



with the members of the sequence at the lower ends of the successive arcs (from left to right, of course).
A hierarchy is written as follows (figure 2):

where the box represents the label and provides space for writing (some of) the attributes.

Hierarchies may be written as formulas, using the notation that was introduced in the previous section, but we also introduce the following infix notation for improved legibility:

: is used as an infix symbol for forming hierarchies:
$J \in L \wedge x \in (T \cup M) \Rightarrow J:x = <h, J, x>$
and also for forming attributes:
$e \in E \wedge m \in M \Rightarrow e:m = <e, m>$.

; is used as an infix symbol for the function *pfx*, restricted to the domain $H \times T$, for constructing trees.

For example, $x;y;z;nil = <x, y, z>$.
If h_1, h_2 and h_3 are hierarchies and J is a label,

$$J:(h_1;h_2;h_3;nil)$$

is another hierarchy.

We use the following *precedence rules* for the infix operators:

$x;y;z = x;(y;z)$
$J:x;y = (J:x);y$
$x;J:y = x;(J:y)$
$a.b.x = a.(b.x)$.

4. SURROUNDINGS

The concept of a *cursor* is fundamental in most practical information management systems. It is a point in the data repository relative to which the operations are performed. We introduce the cursor as a distinguished object, which shall be written # . It is not a member of any of the domains that were introduced in the previous section. We shall use two domains defined informally as follows:

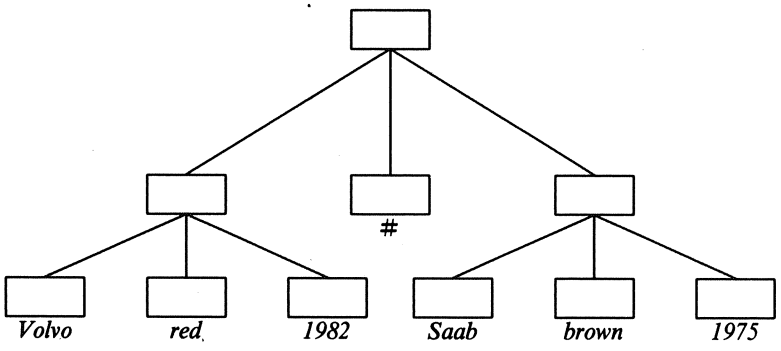
A *surrounding* is similar to a hierarchy, except that # occurs exactly once in a position which would otherwise have been taken by an atom or tree. The cursor may not occur in an attribute.

A *perspective* is also similar to a hierarchy, except that # occurs in exactly one position which would otherwise have been taken by a hierarchy.

In other words, the difference between surroundings and perspectives is that in a surrounding the cursor # 'has' a label (there is a substructure consisting of a label and the cursor). The cursor with 'its' label (resp. the cursor itself as the case may be) may be located between two hierarchies in a tree (= sequence of hierarchies). Thus it is located *between* structures rather than *at* a structure.

The surrounding is a basic concept in the IMS: the operations which the user invokes interactively while he is working with the system, such as inserting or deleting at the position of the cursor, or moving the cursor, are functions from surroundings to surroundings.

The following is an example of a surrounding (figure 3):



For a strict definition, we introduce variants of the recursively defined domains H and T as follows:

$$H' = \{h\} \times L \times (T' \cup \{\#\})$$
$$T' = H^* \ conc \ H'! \ conc \ H^*$$

and

$$H'' = (\{h\} \times L \times T'') \cup \{\#\}$$
$$T'' = H^* \ conc \ H''! \ conc \ H^* \ .$$

Thus every member of T' is a sequence of surroundings, exactly one of which contains the cursor symbol (not more than one of it) and similarly for T''. (Remember that $A! = \{<a> \mid a \in A\}$.) Then H' is the domain of surroundings and H'' is the domain of perspectives. We shall write

$U = H'$
$P = H''$.

For the specification of operations on surroundings it is convenient to introduce functions allowing us to describe a surrounding relative to the cursor position, so that the structures that are adjacent to the cursor appear close to the top of the expression rather than deep down in a substructure. We introduce the following functions:

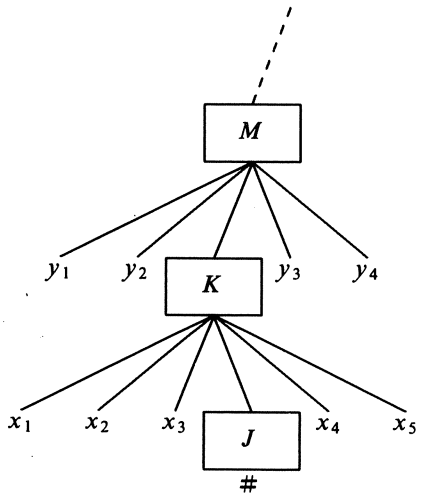
: $L \times P \rightarrow U$
$J:p = subst[\#, J:\#, p]$

per $T \times U \times T \rightarrow P$
$per[l,u,r] = subst[\#, rev.l \ conc \ \# ; r, u]$.

It is easily seen that the ranges of these functions are U and P, respectively. A surrounding as in figure 4 can now be written

$J:per[l, K:p, r]$

where r is the sequence of 'sister' hierarchies to the right of the cursor, in their ordinary order; J is the label just 'above' the cursor symbol in the diagrams; l is the sequence of 'sister' hierarchies to the left of the cursor, in reverse order; and K is the label immediately above J in the diagram. It is the label above the members of l and r in the hierarchy from which the surrounding was formed. Finally, p may be # or a new expression formed using *per*. In this way, a surrounding can be written so that the information close to the cursor appears on the top level of the expression. This is important when specifying IMS operations that have effects close to the cursor.



$J:per[l, K:p, r]$
where
$r = <x_4, x_5> = x_4; x_5; nil$
$l = <x_3, x_2, x_1> = x_3; x_2; x_1; nil$
$p = per[y_2; y_1; nil, M:p', y_3; y_4; nil]$

Figure 4

In the applications, attributes are used for a number of purposes: for specifying the field names for fields in a record; for specifying record types and the choice of keys; for specifying the position of various substructures on the screen or in a printout, etc. But there will also be many situations in which there are no attributes, i.e. in which the label is the empty set.

An expression $J:p$ is *fully inverted* if and only if either p is the constant # or if it has the form $per[l,u,r]$, where u is fully inverted. It is easily seen that each member of U can be written as a fully inverted expression in exactly one way.

Let us give one brief example of how this structure may be used. A conventional record may be represented by a hierarchy in which each daughter has the label

$$\{fld:n\}$$

(where *fld* is a constant and n is a variable). Such an expression will be abbreviated by capitalizing the symbol for n, for example

$$Year=\{fld:year\}\ .$$

The surrounding in figure 3 above, if provided with reasonable field names, can now be written:

$$nil:per[(Manuf:Volvo;Color:red;Year:1982;nil);nil,nil:\#,\\ (Manuf:Saab;Color:brown;Year:1975;nil);nil]\ .$$

If the cursor is moved to a position between the nodes for *Volvo* and *red*, the surrounding becomes:

$$nil:per[Manuf:Volvo;nil,\\ nil:per[nil,nil:\#,\\ (Manuf:Saab;Color:brown;Year:1975;nil);nil],\\ Color:red;Year:1982;nil]\ .$$

5. DEFINITIONS OF OPERATIONS AND THE APPROACH TO THEIR COMPILATION

The simple interactive operations in an IMS are those which add, delete, and modify structures immediately before or after the cursor and those which move the cursor to a new position, for example one step forward or backward. Thus a simple view of an IMS is that it is a system which at each moment has a *state* which is a member of U. It receives from the user successive operations which are mappings $U \rightarrow U$ and changes state accordingly. These operations can be conveniently specified using the notation of the previous sections. For example, the operation *nx* that moves the cursor one step to the right is characterized by the axiom:

$$nx.C:per[l,u,x;r]=C:per[x;l,u,r]$$

plus one other axiom which specifies what happens when the cursor is already at the right end of the sequence, and which for example may be chosen as:

$$v=C:per[l,u,nil]\Rightarrow nx.v=v\ .$$

When a command driven system such as an IMS is implemented, it is natural to organize the program around a *case* statement which contains one branch for each possible operation, each branch being the implementation of the axiom(s) specifying the

corresponding operation. The topic of the present paper is to show how the transition from specification to implementation for an operation can be done within a single logical system.

Before we go into the details of compilation, let us specify a number of additional operations which have been implemented in the present system in order to provide some intuition for what the specifications may look like. We only specify the main case in the definition, corresponding to the first line in the definition of *nx* above, and omit the specifications of exceptional cases.

The *bk* operation moves the cursor one step backward:

$$bk.C:per[x;l,u,r]=C:per[l,u,x;r]\,.$$

The *dw* operation travels downward along the substructure to the right of the present cursor position:

$$x\in T \Rightarrow dw.C:per[l,u,J:x;r]=C:per[nil,J:per[l,u,r],x]\,.$$

The *up* operation leaves a substructure and positions the cursor after it:

$$up.C:per[nil,J:per[l,u,r],x]=C:per[J:x;l,u,r]\,.$$

The *rs* operation resets or 'rewinds' the cursor to the beginning of the present subtree level:

$$rs.C:per[x;l,u,r]=rs.C:per[l,u,x;r]$$
$$v=C:per[nil,u,r]\Rightarrow rs.v=v\,.$$

The $in[x]$ operation inserts an element before the cursor:

$$in[x].C:per[l,u,r]=C:per[nil:x;l,u,r]\,.$$

The *del* operation deletes the element immediately after the cursor:

$$del.C:per[l,u,x;r]=C:per[l,u,r]\,.$$

We can now proceed to the issue of compilation. Our method is based on making a transformation between two subsets of the language of first-order predicate calculus (FOPC), namely the subset used for specifications and a subset which corresponds to a conventional programming language. The *specification* for an operation *op* is assumed to be written in the form

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

where each of the A_i is a *specification clause* having the form

$$P_1 \wedge P_2 \wedge \ldots \wedge P_m \Rightarrow op.x=y$$

where x and y are surrounding-valued *expressions* and where the *literals* P_i are formed using the notation that was introduced in previous sections.

As the recursive specification of *rs* above shows, *op* may be used also for forming the expression y. Of course, other operations, defined in one or more other clauses of the specification, may occur as well. We do not impose such constraints as would guarantee in general that a specification is solvable; that has to be verified separately for each individual specification. The same applies to the programs that implement the specification.

The *procedure* for *op*, by contrast, is restricted to the form

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

where each A_i is a procedure clause of the form

$$P_1 \wedge \ldots \wedge P_n \Rightarrow op.u^0 = u^n$$

where u^0 and u^n are now restricted to *single variables* and where there are also a number of restrictions on the literals P_i. They may for example have the form

$$u^{k+1} = o.u^k$$

where u^{k+1} and u^k are single variables and o is an expression. The conversion from specification to program is a kind of pattern compilation: whereas the specification uses construction functions such as *per*, the program contains e.g. tests for membership in a domain and decomposition functions such as *rg*, defined by

$$rg.C : per[l,u,r] = r \ .$$

The language for procedure clauses resembles a simple, fairly conventional programming language (more precisely: a single assignment language), while it is at the same time represented entirely within FOPC. Various kinds of P_i correspond to various kinds of programming language statements: variable assignments, operations on the data base, conditionals, etc.

The representation of the procedure for an operation will be introduced in a stepwise fashion, by defining a sequence of sublanguages (each sublanguage being a first-order logic with certain syntactic restrictions).

6. DECOMPOSITION AND MODIFICATION FUNCTIONS

In the sublanguages we need the following predicates:

On T $\quad ispfx[t] \equiv (\exists x,y)\, t = x;y$
(In other words, t is not the empty sequence. The name for the predicate was selected because $x;y$ is also written $pfx[x,y]$.)

On P $\quad isper[p] \equiv (\exists l,u,r)\, p = per[l,u,r]$
(In other words, p is not $*$.)

On U $\quad nontop[C:p] \equiv isper[p]$.

We also need decomposition functions for each domain except M and its subsets. For the domains defined in section 3 this can be done by the ordinary conventions:

For H: if $h = J:x$, then
$j = s - l.h$
$x = s - tm.h$ (with a simple generalization of the name convention for selection functions).

For T: if $t = h_1; h_2; \ldots$, then
$h_1 = hd.t$
$h_2; \ldots = tl.t$. (The functions *hd* and *tl* are therefore in T restricted to those t that satisfy $ispfx.t$.)

For L, the . primitive serves to identify one component of the label at a time.

For A: if $a = e : m$, then
$e = s - e.a$
$m = s - m.a$.

When objects in the domains P and U are written using fully inverted expressions, the convention for forming decomposition functions of the form $s -$ do not apply as usual, since the functions : and *per* do not directly correspond to composition rules of the abstract syntax for these domains. However, since each surrounding has just one representation as a fully inverted expression, similar functions are well defined, e.g. the 'right list of sisters function',

$$rg.J : per[l, u, r] = r \ .$$

We shall use the mnemonic names *lbl* (label), *pc* (perspective content), *lf* (left), *ow* (owner), and *rg* (right) for decomposing a surrounding according to the definitions in the following table. The table also defines the modification functions *slf* (set left), *srg* (set right), *slbl* (set label), and *embed*. These functions are used for obtaining the state transitions required by an operation. For each function we specify its name, the types of its domain and range, the precondition under which it is defined (as an additional restriction on the domain) and, on the next line, the equation that specifies the relationship between argument and value.

name	**type**	**precondition**
lbl	$U \to L$ $lbl.J : p = J$	
pc	$U \to P$ $pc.J : p = p$	
rg	$U \to T$ $rg.J : per[l, u, r] = r$	*nontop*[*u*] required for *rg.u*
lf	$U \to T$ $lf.J : per[l, u, r] = l$	*nontop*[*u*] required for *lf.u*
ow	$U \to U$ $ow.J : per[l, u, r] = u$	*nontop*[*u*] required for *ow.u*
slf	$T \to (U \to U)$ $slf[t].C : per[l, u, r] = C : per[t, u, r]$	*nontop*[*u*] required for *slf*[*t*].*u*
srg	$T \to (U \to U)$ $srg[t].C : per[l, u, r] = C : per[l, u, t]$	*nontop*[*u*] required for *srg*[*t*].*u*
slbl	$L \to (U \to U)$ $slbl[J].C : p = J : p$	
embed	$L \times T \times T \to (U \to U)$ $embed[J, l, r].u = J : per[l, u, r]$	

The following properties of these functions are readily inferred from the definitions (all constrained by the preconditions of the functions):

$$nontop[srg[x].u]$$
$$nontop[slf[x].u]$$
$$nontop[u] \Rightarrow nontop[slbl[J].u]$$
$$nontop[embed[J,l,r].u]$$

$$lf.srg[x].u = lf.u$$
$$lf.slf[x].u = x$$
$$lf.slbl[J].u = lf.u$$

$$rg.srg[x].u = x$$
$$rg.slf[x].u = rg.u = rg.slbl[J].u$$

$$lbl.srg[x].u = lbl.u = lbl.slf[x].u$$
$$lbl.slbl[J].u = J$$

$$ow.srg[x].u = ow.u = ow.slf[x].u = ow.slbl[J].u$$

$$lf.embed[J,l,r].u = l$$
$$rg.embed[J,l,r].u = r$$
$$lbl.embed[J,l,r].u = J$$
$$ow.embed[J,l,r].u = u$$

These definitions should be included as proper axioms in a forthcoming first-order IMS theory, together with e.g. axioms which effectively are translations of the domain equations, such as

$$r \in T \Rightarrow ispfx[r] \vee r = nil \ .$$

Although the precise formulation of such a theory must wait for a later occasion, we can already observe here that, in the IMS theory, the relation between the specification of an operation and a procedure that implements it should be one of logical consequence, i.e. the implementation should be a consequence of the specification.

7. LPSL - LINEAR PROGRAM SUBLANGUAGE

An *LPSL program* is conjunction of *LPSL clauses*, each of which has the form

$$(\forall\, v^1, v^2, \ldots, v^m, u^0, u^1, \ldots, u^n)$$
$$P_1 \wedge P_2 \wedge \ldots \wedge P_p \Rightarrow op[v^1, \ldots, v^h].u^0 = u^n$$

The operation *op* is said to *have* this clause. The clause must satisfy:

I. Simple constraints:

$$m, n, p \geqslant 0$$
$$0 \leqslant h \leqslant m \ .$$

II. The literals P_i must have one of the following forms:

1. $u^k = o.u^{k-1}$

where o is an expression for a mapping $U \rightarrow U$, formed using one of the functions *ow*, *slf*, *srg*, *slbl*, *embed*, which have just been defined, or operations which *have* other

clauses in the same LPSL program.

2. $v^k = x$
where x is an expression whose value has a type other than U or P, and which is formed using composition and/or decomposition functions as defined above.

3. An arbitrary predicate expression z, formed using some of the predicates defined above (*ispfx*, *isper*, *nontop*) or an ordinary predicate such as equality.

The function *per* may not be used to form expressions, i.e. surroundings can only be modified, not created afresh. Also, the expressions x and z in cases 2 and 3 may not use any surrounding valued function except *ow*.

III. The constituent expressions (o, x, z) must satisfy certain constraints which we shall now define. For a given LPSL clause, with the variables and indices specified above, we define a sequence $c_0, c_1, \ldots, c_p$ of *current variable sets*, which are sets of variable symbols, (not sets of the objects that the variables denote), and which are defined as follows:

$c_0 = \{v^1, \ldots, v^h, u^0\}$;

if P_i has the form $u^k = o.u^{k-1}$ and
$c_{i-1} = c \cup \{u^{k-1}\}$, then
$c_i = c \cup \{u^k\}$;

if P_i has the form $v^k = x$, then
$c_i = c_{i-1} \cup \{v^k\}$;

if P_i is a predicate expression z, then
$c_i = c_{i-1}$.

Clearly every c_i contains exactly one u^k variable. The constraints on the sequence of P_i are now the following:

a) If P_i is of type (1) above, u^{k-1} must be a member of c_{i-1}. Intuitively speaking, u^k stands for the state of the data repository after the 'operation' P_i. The succession of such states as are obtained by successive update operations are presented in our logic by a sequence of u_k variables. The actual implementation in a conventional programming language can contain a single variable u and perform successive assignments to it and/or perform successive operations with side-effects on its value.

b) If P_i is of type (2) above, the variable to the left of the equality sign must not be a member of c_{i-1}. In other words, it must be a new addition to c_i. Intuitively, P_i is an assignment to a (programming language) variable to which no previous assignment has been made.

c) All variables which occur in o, x, or z (as the case for P_i may be) must be members of c_{i-1}. In programming language terms, this means that variables must have a value assigned to them before they are used.

If P_i is of type (3) i.e. a condition z, it should be viewed as the condition in an *if* statement (of a conventional programming language). The subsequent $P_{i+1} \ldots$ constitute the *then* branch of the *if* statement. The *else* branch may be specified by another LPSL clause.

d) A decomposition function as described above may be used in P_i only if its precondition occurs among or if it is a consequence in the IMS theory from

$P_1 \wedge P_2 \wedge \ldots \wedge P_{i-1}$. This is our counterpart of type checks in programming languages.

e) $u^n \in c_p$.

In order to make condition (d) effectively decidable in general, it would have to be strengthened to for example 'can be inferred in at most 1000*p deduction steps from...'. This practical matter will be bypassed here. This ends the list of constraints.

For example, the following is an LPSL clause:

$$
\begin{aligned}
&(\forall\, v^1, u^0, u^1, u^2)\\
&\quad nontop[u^0] \vee\\
&\quad v^1 = rg.u^0 \vee\\
&\quad ispfx[v^1] \vee\\
&\quad u^1 = slf[hd[v^1]; (lf.u^0)].u^0 \vee\\
&\quad u^2 = srg[tl[v^1]].u^1 \Rightarrow nx.u^0 = u^2 .
\end{aligned}
$$

If an IMS theory is designed in the way suggested above, this LPSL clause must be a consequence in the theory of the following specification clause

$$nx.C : per[l, u, x; r] = C : per[x; l, u, r] .$$

To verify that constraint (d) on LPSL clauses is satisfied throughout, we notice that the precondition $nontop[u^0]$ legitimizes the use of $rg.u^0$, $lf.u^0$, and $slf.[\ldots].u^0$. The precondition $ispfx[v^1]$ legitimizes the use of $hd[v^1]$ and $tl[v^1]$. Finally, $u^1 = slf[\ldots].u^0$ implies $nontop[u^1]$ (above) which legitimizes $u^2 = srg[\ldots].u^1$.

The idea behind the LPSL clause is that, in addition to being a consequence of a specification clause, it should express a procedure for executing the operation in the cases covered by the specification clause. In other words, there should be a simple, essentially syntactic, transformation transforming an LPSL clause into a *reasonable* procedure in a conventional programming language doing the same thing. The intuition as to how the LPSL constructs are related to programming language constructs has already been given. The constraints that have been imposed on LPSL clauses were dictated by this goal. On the other hand, we are not yet ready to give a strict proof of this correspondence between LPSL and programming languages. The set of constraints that has been given here should therefore be seen as provisional. However, for the simple examples in this paper it should be fairly clear how a specification clause can be transformed into an LPSL clause and how an LPSL clause can be transformed into a procedure.

8. BPSL - BRANCHING PROGRAM SUBLANGUAGE

The specification of an operation may consist of several clauses, e.g.

$$
\begin{aligned}
&nx.C : per[l, u, x; r] = C : per[x; l, u, r] \wedge\\
&nx.C : per[l, u, nil] = C : per[l, u, nil] .
\end{aligned}
$$

When these are transformed into LPSL, we often obtain LPSL clauses in which the first few A_i are the same. It is natural to introduce a sublanguage in which these can be shared and which thus contains the counterparts of *if* statements in conventional programming languages. This is what BPSL does. The present section presents BPSL, although still in a somewhat sketchy way.

A *BPSL program* is a set of BPSL procedures for different operations. A *BPSL procedure* for an operation *op* has the form

$$(\forall\, v^1, v^2, \ldots, v^m, u^0, u^1, \ldots, u^n)\ op[v^1, \ldots, v^h].u^0 = / \ R$$

where R is a *computation rule* in BPSL having the form

$$P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_p \rightarrow u$$

with $p \geqslant 0$, while the successive P_i satisfy the same conditions as in LPSL and u is either a variable u^m with $m \leqslant n$ or an expression

branch
if R_1
if R_2
...

Each of the R_i is called an *if* clause and again is a computation rule in BPSL. When several *if* clauses are nested inside each other, the possible ambiguity is resolved by indentation: *if* clauses in the same u have their initial *if* keyword directly underneath each other.

We must also impose on computation rules in BPSL a syntactic restriction corresponding to the restriction on LPSL clauses, but in order to make it understandable we must first make clear the meaning of BPSL procedures.

First, an example of a BPSL procedure:

$(\forall\, v^1, u^0, u^1, u^2)\ nx.u^0 = / \cdot$
  $nontop[u^0] \rightarrow$
  $v^1 = rg.u^0 \rightarrow branch$
    $if\ ispfx[v^1] \rightarrow$
      $u^1 = slf[hd[v^1]; (lf.u^0)].u^0 \rightarrow$
      $u^2 = srg[tl[v^1]].u^1 \rightarrow u^2$
    $if\ v^1 = nil \rightarrow u^0$

This procedure is equivalent to the specification of *nx* that was given at the beginning of this section. The first *if* clause handles the case of the first line in the specification, while the second *if* clause handles the second line. Notice also that the second *if* clause uses u^0, not u^2, i.e. variable numbers are incremented in parallel in the branches.

The meaning of a BPSL procedure is defined by transformations to an equivalent set of LPSL clauses, as follows:

If v is a variable,

$$op.u = / \ v$$

has the same meaning as

$$op.u = v \ .$$

The expression

$$x = / P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_p \rightarrow u$$

has the same meaning as

$$P_1 \wedge P_2 \wedge \ldots \wedge P_p \Rightarrow x = / \ u \ .$$

The expression

$$x =/ \textit{branch}$$
$$\quad \textit{if}\ u^1$$
$$\quad \textit{if}\ u^2$$
$$\quad \ldots$$

has the same meaning as

$$(x =/ u_1) \wedge (x =/ u_2) \wedge \ldots .$$

Using these rules, it is clearly possible to rewrite each BPSL procedure as a conjunction of LPSL clauses (apart from the constraints). We shall call this the *flat form* of the BPSL procedure. The above example of the BPSL procedure for *nx* can thus be rewritten as the conjunction of two LPSL clauses, one of which is identical to the clause for *nx* that was given in the previous section. We impose on BPSL procedures the natural constraint that each of the terms in the flat form must satisfy the constraints imposed on LPSL clauses.*

If a BPSL procedure is to be a usable program, it must take into account all cases that may occur in the IMS in which it is used. The completeness criterion is that in any expression

$$P_1 \rightarrow \ldots \rightarrow P_i \rightarrow \textit{branch}$$
$$\quad \textit{if}\ Q_1 \rightarrow v^1$$
$$\quad \ldots$$
$$\quad \textit{if}\ Q_n \rightarrow v^n$$

the various criteria Q_i must satisfy (again in the IMS theory)

$$P_1 \wedge \ldots \wedge P_i \Rightarrow (Q_1 \vee \ldots \vee Q_n) .$$

The simplest way of satisfying this condition is if $n = 2$ and $Q_2 = \neg\, Q_1$, which means we effectively have an ordinary *if ... then ... else ...* expression. A more common case seems to be that the Q_i represent the various possible cases in the syntax for an element, e.g. whether a perspective is # or not, or (in the last example) whether a sequence is empty or not.

In our BPSL procedure for *nx* above, the property $\textit{nontop}[u^0]$ is required by subsequent operations. It means the procedure is not complete in this sense. However, all IMS operations defined in section 5 assume the predicate *nontop* to be valid for their arguments and preserve that property. It would therefore be reasonable to treat it as an invariant of the IMS and to require only that BPSL procedures shall be well defined and complete relative to the invariant. (This relaxes requirement III.(d) in the definition of LPSL.)

---

* except, since different *if* clauses may require a different number of u variables, a clause in the flat form may end on

$$\cdots \Rightarrow op[v^1, \ldots, v^h].u^0 = u^k$$

where $k \leqslant n$ without necessary equality. Compare with the definition of LPSL clauses above.

9. SECOND EXAMPLE

The operation *up* may be specified as:

$$up.J:per[nil,K:per[ll,uu,rr],r]=J:per[(K:r);ll,uu,rr] \wedge$$
$$up.J:per[nil,K:nil,r]=J:per[nil,K:nil,r] \wedge$$
$$up.J:per[x;l,u,r]=up.J:per[l,u,x;r]$$

and a derived expression in BPSL is:

$$(\forall\, v^1,v^2,u^0,u^1,u^2,u^3)\ up.u^0 = /$$
$$\quad nontop[u^0] \rightarrow branch$$
$$\quad if\ ispfx[lf.u^0] \rightarrow$$
$$\quad\quad u^1=bk.u^0 \rightarrow$$
$$\quad\quad u^2=up.u^1 \rightarrow u^2$$
$$\quad if\ lf.u^0=nil \rightarrow$$
$$\quad\quad v^1=lbl.u^0 \rightarrow$$
$$\quad\quad v^2=rg.u^0 \rightarrow branch$$
$$\quad\quad if\ nontop[ow.u^0] \rightarrow$$
$$\quad\quad\quad u^1=ow.u^0 \rightarrow$$
$$\quad\quad\quad u^2=slf[(lbl.u^1):v^2;(lf.u^1)].u^1 \rightarrow$$
$$\quad\quad\quad u^3=slb[v^1].u^2 \rightarrow u^3$$
$$\quad\quad if\ pc[u^1]=nil \rightarrow u^0$$

where *bk* was defined in section 5 as the inverse operation of *nx*.

10. IPSL - THE IMPLICIT-SURROUNDING PROGRAM SUBLANGUAGE

LPSL and BPSL use explicit variables $u^0,u^1,\ldots$ for the successive surroundings that are the states of the machine. When an expression in these sublanguages is transformed into a conventional program, as is readily done, it is of course possible to use a single variable for the 'current u' in the implementation, but the notation is still unnecessarily cluttered by all the u variables. An implicit-surrounding program sublanguage IPSL, in which references to the u^i surroundings do not have to be written out, can be defined by a reversible transformation from BPSL to IPSL.

Each BPSL procedure

$$(\forall\, v^1,\ldots,v^m,u^0,\ldots,u^n)\ op[v^1,\ldots,v^h].u^0 = /\ R$$

is transformed into

$$Op[v^1,\ldots,v^h] = /\ (\textbf{local}\ v^{h+1},\ldots,v^m)\ R'$$

where R' is obtained from R by the *is* transformation.

A computation rule in BPSL,

$$P_1 \rightarrow \ldots \rightarrow P_p \rightarrow u$$

is transformed into

$$P_1' \rightarrow \ldots \rightarrow P_p'$$

if u is a single variable, and into

$$P_1' \rightarrow \ldots \rightarrow P_p' \rightarrow u'$$

if u is a branch expression. In the latter case, the *if* clauses in the branch expression are all transformed using the same *is* transformation.

A literal P_i in a computation rule is transformed into a corresponding literal P_i' by the following transformation, according to the various cases that were specified in section 7:

An expression

$$u^k = x$$

where x has the form $o.u^{k-1}$, is transformed into an expression

$$x'$$

where the transformation from x to x' will be specified immediately.

An expression

$$v^k = x$$

is transformed into an expression

$$v^k = x' .$$

A predicate expression z, finally, is transformed into an expression z'. In all cases, the transformation from x or z to x' or z' is defined recursively as follows:

Constants are unchanged. Variables v^j are also unchanged. Variables u^j for surroundings cannot be transformed.

Functions $U \rightarrow U$ with an explicit variable as argument are capitalized and the argument is omitted. Thus:

$$\begin{aligned}
&nontop[u] \rightarrow Nontop\\
&lbl.u \rightarrow Lbl\\
&pc.u \rightarrow Pc\\
&rg.u \rightarrow Rg\\
&lf.u \rightarrow Lf\\
&ow.u \rightarrow Ow\\
&slf[t].u \rightarrow Slf[t']\\
&srg[t].u \rightarrow Srg[t']\\
&slbl[J].u \rightarrow Slbl[J']\\
&embed[J,l,r].u \rightarrow Embed[J',l',r'] .
\end{aligned}$$

For those cases in which the argument to a function from U to U is not a variable symbol, we introduce the operator $\otimes$ defined by

$$f.g.x = (f \otimes g).x$$

so that e.g. $nontop[ow.u^0]$ can be rewritten as $Nontop \otimes Ow$.

Other functions and predicates (e.g. *ispfx*, *hd*, ;) retain their argument structure, but each of the arguments undergoes the same transformation.

Finally, we add some syntactic sugar: if the final step in a branch has the form

$$if\ Q \rightarrow u^m$$

where u^m is a single variable, then it may be rewritten more suggestively as

if $Q' \rightarrow$ *Ident*

rather than

if Q' .

For example, the definition of *nx* that was given in BPSL in section 8, will be rewritten as:

```
Nx = / (local v1)
  Nontop →
  v1=Rg → branch
    if ispfx[v1] →
      Slf[hd[v1]; Lf] →
      Srg[tl[v1]]
    if v1=nil → Ident .
```

In this way we have moved our notation closer to the notation found in conventional programming languages, but this is only a matter of syntax. The semantics of first-order logic is retained.

11. TRANSFORMATIONS TO STACK MACHINE SUBLANGUAGES

The sublanguages introduced in the previous sections correspond to conventional programming languages in the sense that their literals correspond to different kinds of 'statements' for variable assignment, operations on the data base, conditionals, etc. An application of this correspondence is a translator from (e.g.) BPSL to a conventional programming language. Such a translator only performs a syntactic transformation into a subset of the target language. In this way we obtain a transformation from a specification language to a programming language. The latter can be compiled for execution on a conventional machine. These transformations are illustrated in figure 5.

However, we could achieve greater conceptual economy by considering e.g. BPSL to be *the* programming language for our use. We must then apply conventional compilation techniques to BPSL. The present section will show how this can be done by an extension of the same strategy as was used in previous sections, i.e. by making transformations into yet another subset of first-order logic, which intuitively corresponds to the machine language for a stack machine.

We define the following domains:

$B = \{\textbf{true}, \textbf{false}\}$

$Q = B \cup M \cup H \cup T$

Q is the domain of practically everything. It is needed for defining counterparts of registers and stacks.

$S = U \times Q^* \times Q$

S is the domain of *states* for the stack machine. In a state $<u, d, q>$, the surrounding u is the present contents of the data repository, d is the present stack, and q is the present contents of a distinguished register. Whenever we want to refer to the contents of the top of the stack, we make an operation that pops the stack into the register and then refer to the present contents of the register. We shall discuss the need for such a

register below.

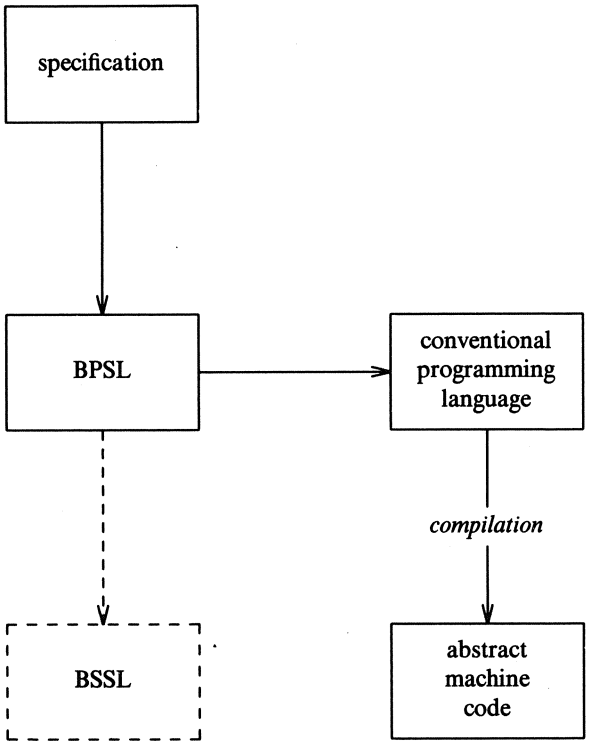


Figure 5

We shall now define a sublanguage, LSSL, which is analogous to LPSL except that it uses states whenever LPSL uses surroundings. As a consequence, it uses a set of operations on states which is different from (although related to) the operations on surroundings that are used in LPSL. Later on, the additional steps that are taken from LPSL to BPSL and IPSL will have direct counterparts for LSSL.

We define the following new predicates and functions on states:

name	**type**
push	$Q \rightarrow (S \rightarrow S)$ **push**$[x].<u,d,q>=<u,x;d,q>$
pop	$S \rightarrow S$ **pop**$.<u,x;d,q>=<u,d,x>$
reg	$S \rightarrow Q$ **reg**$.<u,d,q>=q$
test	predicate on S, defined as: **test**$.<u,d,q> \equiv q =$ **true**

Furthermore, for the predicates and functions that were defined in previous sections with U as domain, we define bold-face counterparts on S. For the predicates as well as for those functions whose range is not U, the counterpart pushes an element on the stack:

nontop $nontop.u \Rightarrow \mathbf{nontop}.<u,d,q>=<u,\mathbf{true};d,q>$
$\neg\, nontop.u \Rightarrow \mathbf{nontop}.<u,d,q>=<u,\mathbf{false};d,q>$

lbl $\mathbf{lbl}.<u,d,q>=<u,(lbl.u);d,q>$

pc $\mathbf{pc}.<u,d,q>=<u,(pc.u);d,q>$

rg $\mathbf{rg}.<u,d,q>=<u,(rg.u);d,q>$

lf $\mathbf{lf}.<u,d,q>=<u,(lf.u);d,q>$.

For those functions whose range is U, the counterpart modifies the surrounding in the state:

ow $\mathbf{ow}.<u,d,q>=<ow.u,d,q>$

slf $\mathbf{slf}.<u,x;d,q>=<slf[x].u,d,q>$

srg $\mathbf{srg}.<u,x;d,q>=<srg[x].u,d,q>$

slbl $\mathbf{slbl}.<u,x;d,q>=<slbl[x].u,d,q>$

embed $\mathbf{embed}.<u,r;J;l;d,q>=<embed[l,J,r].u,d,q>$.

Predicates and functions with other domains than U are given boldface counterparts that take their arguments from the stack of the state and also return their value (represented using the domain B in the case of predicates) on the stack. For example, we define:

pfx $\mathbf{pfx}.<u,y;x;d,q>=<u,(x;y);d,q>$

hd $\mathbf{hd}.<u,x;d,q>=<u,(hd.x);d,q>$

ispfx $ispfx.x \Rightarrow \mathbf{ispfx}.<u,x;d,q>=<u,\mathbf{true};d,q>$
and a similar definition for the case of $\neg\, ispfx.x$.

equal $x=y \Rightarrow \mathbf{equal}.<x,y;x;d,q>=<u,\mathbf{true};d,q>$
and so forth.

Starting with an LPSL expression representing operations on surroundings, we can rewrite it as an LSSL expression representing operations on stacks by changing every u^k variable to a corresponding s^k variable, and by converting composite expressions into sequences of stack operations. For example, the expression

$$u^2=srg[tl[v^1]].u^1$$

(in the BPSL procedure for nx) is transformed into

$$s^2 = \mathbf{push}[v^1].s^1 \rightarrow$$
$$s^3 = \mathbf{tl}.s^2 \rightarrow$$
$$s^4 = \mathbf{srg}.s^3 .$$

The first operation pushes the value of v^1 on the stack; the second one performs a *tl* operation on the top element of the stack, and the third one pops the top element off the stack and inserts it as the new left sister list of the present surrounding.

More generally, the various types of literals in LPSL are transformed as follows:

Type (1) literals in LPSL are reduced to a sequence of literals in LSSL which first pushes elements on the stack with operations such as **push** with a single variable parameter, **rg**, **lf**, etc., later modifies them with operations that accept/put arguments/results from/on the stack, and finally deletes them with an operation such as **srg**.

Type (2) literals in LPSL are similarly transformed into a sequence of literals in LSSL, the last two clauses of which are:

$$s^{k+1} = \mathbf{pop}.s^k \rightarrow$$
$$v = \mathbf{reg}.s^{k+1} .$$

Finally, type (3) literals are transformed into sequences of LSSL literals ending on:

$$s^{k+1} = \mathbf{pop}.s^k \rightarrow$$
$$\mathbf{test}.s^{k+1} .$$

With this notation, we can e.g. rewrite the LPSL program for *nx* as

$$(\forall\, v^1, s^0, \ldots, s^{15})$$
$$s^1 = \mathbf{nontop}.s^0 \wedge$$
$$s^2 = \mathbf{pop}.s^1 \wedge$$
$$\mathbf{test}.s^2 \wedge$$
$$s^3 = \mathbf{rg}.s^2 \wedge$$
$$s^4 = \mathbf{pop}.^3 \wedge$$
$$v^1 = \mathbf{reg}.s^4 \wedge$$
$$s^5 = \mathbf{push}[v^1].s^4 \wedge$$
$$s^6 = \mathbf{ispfx}.s^5 \wedge$$
$$s^7 = \mathbf{pop}.s^6 \wedge$$
$$\mathbf{test}.s^7 \wedge$$
$$s^8 = \mathbf{push}[v^1].s^7 \wedge$$
$$s^9 = \mathbf{hd}.s^8 \wedge$$
$$s^{10} = \mathbf{lf}.s^9 \wedge$$
$$s^{11} = \mathbf{pfx}.s^{10} \wedge$$
$$s^{12} = \mathbf{slf}.s^{11} \wedge$$
$$s^{13} = \mathbf{push}[v^1].s^{12} \wedge$$
$$s^{14} = \mathbf{tl}.s^{13} \wedge$$
$$s^{15} = \mathbf{srg}.s^{14} \Rightarrow \mathbf{nx}.s^0 = s^{15}$$

We can now see why even for a stack machine the register component of the state is needed: for type (2) literals, we must pop a value from the stack and bind that value to a *v* variable, but at the same time we must bind the new state containing the popped stack to an *s* variable. This is taken care of by the register as shown in the example. Since we do not perform any operations on the contents of the register nor allow the register contents to be pushed back on the stack, we still have essentially a stack

machine.

Since LSSL is similar to LPSL, we can now introduce branch constructs into LSSL. This results in a sublanguage BSSL, in the same way as for BPSL. We must notice, however, that in BPSL it was the first literal after the *if* that constituted the test, whereas in BSSL the test (using the operation **test** defined above) can occur only after a number of operations involving the stack. It should be thought of as an exit from the *if* clause or, more precisely, an exit that resets the state of the machine to what it was when the *if* clause was entered.

The reserved word *if* in the BPSL syntax is therefore inappropriate and we shall use >> for the same purpose of marking the start of an *if* clause in BSSL.

Finally, we can eliminate the state variables from the notation by introducing ISSL by the same conventions as for IPSL, i.e. we capitalize functions and predicate names to indicate that a state parameter is implicit. At the same time, we allow comments to be written at the end of each line. We then have something which feels like the machine language of a stack machine, except for the way variables are handled:

Nx = / (*local* v^1)	
Nontop →	check whether *surr* is C:*per*$[l,u,y]$ rather than C: #
	push truth value on stack
Pop →	pop truth value to register
Test →	if not satisfied, exit
Rg →	push y on stack
Pop →	move y to register
v^1=**Reg** → *branch*	bind v to y
>> **Push**[v^1] →	push y on stack
Ispfx →	check that y is not nil
Pop →	pop result of test to register
Test →	exit if test fails
Push[v^1] →	push y again, assume it is $x\,;r$
Hd →	change y into x on top of stack
Lf →	push l above x on top of stack
Pfx →	now $x\,;l$ is on top of stack
Slf →	assign $x\,;l$ as new list of left sisters in surrounding
Push[v^1] →	push $y=x\,;y$ on top again
Tl →	now r is on top of stack
Srg	assign r as new list of right sisters in surrounding; current state is result
>> **Push**[v^1] →	push y on stack again
Push[*nil*] →	push *nil* above it
Equal →	compare
Pop	
Test	if not equal then exit, otherwise current state is result.

The machine is a bit clumsy because of all the transfers to and from the register, but this can easily be remedied by introducing and using a few more operations. In principle we already have a machine here to which most of the intuitions of regular stack machines apply.

12. PROPOSED CONTINUED WORK

The present paper has been semiformal. The next step would be to verify that, for strict definitions of all languages involved, the transformations between the languages are possible in all cases. We have tried to convince the reader that such proofs will be possible, but there are some minor details which have been bypassed in the present paper. For example, we do not allow surroundings to be pushed on the stack. This is in principle a reasonable constraint, but as a compensation we must find some way of allowing the user to access the components of the owner of the current surrounding, for example in the expression

$$nontop[ow.u^0]$$

in the example in section 9. This requires some simple additions to the repertoire of operations.

The ISSL language, which was the last transformation step in the present paper, is not in all respects a reasonable machine language. A few more transformations and modifications should be made:

- Introduce a real register machine and/or smooth the stack/register transfers in the present treatment.
- Treat program variables in a variable stack in a separate component of the machine state rather than modeling them by predicate-logic variables.
- If, with the current definition of BSSL, the **test** statement fails during execution of an *if* clause an exit from the *if* clause should be performed and the next *if* clause should be entered with the machine in the state it was in when the first *if* clause was entered. In an implementation this would require that a copy of the state be made, but it is intuitively clear that the surrounding and stack at the time of exit are the same as at the time the first *if* clause was entered. In this and other ways, machine behaviour more like that of a real machine should be obtained, while retaining the close correspondence through all levels from specification to executable machine language.

REFERENCES

[BLI82] Blikle, A., "Desophisticating denotational semantics," to appear in: *Proceedings of the IFIP World Computer Congress*, 1983.

[BOE80] Boehm, B.E., "Developing small-scale application software products," **in** Lavington, S.H., (Ed.), *Information Processing 83*, North-Holland, 1980.

[SAN82] Sandewall, E., "An approach to information management systems," Report LiTH-MAT-R-82-19, Software Systems Research Center, Linköping University, July 1982.

MC SYLLABI

1.1 F. Göbel, J. van de Lune. *Leergang besliskunde, deel 1: wiskundige basiskennis.* 1965.

1.2 J. Hemelrijk, J. Kriens. *Leergang besliskunde, deel 2: kansberekening.* 1965.

1.3 J. Hemelrijk, J. Kriens. *Leergang besliskunde, deel 3: statistiek.* 1966.

1.4 G. de Leve, W. Molenaar. *Leergang besliskunde, deel 4: Markovketens en wachttijden.* 1966.

1.5 J. Kriens, G. de Leve. *Leergang besliskunde, deel 5: inleiding tot de mathematische besliskunde.* 1966.

1.6a B. Dorhout, J. Kriens. *Leergang besliskunde, deel 6a: wiskundige programmering 1.* 1968.

1.6b B. Dorhout, J. Kriens, J.Th. van Lieshout. *Leergang besliskunde, deel 6b: wiskundige programmering 2.* 1977.

1.7a G. de Leve. *Leergang besliskunde, deel 7a: dynamische programmering 1.* 1968.

1.7b G. de Leve, H.C. Tijms. *Leergang besliskunde, deel 7b: dynamische programmering 2.* 1970.

1.7c G. de Leve, H.C. Tijms. *Leergang besliskunde, deel 7c: dynamische programmering 3.* 1971.

1.8 J. Kriens, F. Göbel, W. Molenaar. *Leergang besliskunde, deel 8: minimaxmethode, netwerkplanning, simulatie.* 1968.

2.1 G.J.R. Förch, P.J. van der Houwen, R.P. van de Riet. *Colloquium stabiliteit van differentieschema's, deel 1.* 1967.

2.2 L. Dekker, T.J. Dekker, P.J. van der Houwen, M.N. Spijker. *Colloquium stabiliteit van differentieschema's, deel 2.* 1968.

3.1 H.A. Lauwerier. *Randwaardeproblemen, deel 1.* 1967.

3.2 H.A. Lauwerier. *Randwaardeproblemen, deel 2.* 1968.

3.3 H.A. Lauwerier. *Randwaardeproblemen, deel 3.* 1968.

4 H.A. Lauwerier. *Representaties van groepen.* 1968.

5 J.H. van Lint, J.J. Seidel, P.C. Baayen. *Colloquium discrete wiskunde.* 1968.

6 K.K. Koksma. *Cursus ALGOL 60.* 1969.

7.1 *Colloquium moderne rekenmachines, deel 1.* 1969.

7.2 *Colloquium moderne rekenmachines, deel 2.* 1969.

8 H. Bavinck, J. Grasman. *Relaxatietrillingen.* 1969.

9.1 T.M.T. Coolen, G.J.R. Förch, E.M. de Jager, H.G.J. Pijls. *Colloquium elliptische differentiaalvergelijkingen, deel 1.* 1970.

9.2 W.P. van den Brink, T.M.T. Coolen, B. Dijkhuis, P.P.N. de Groen, P.J. van der Houwen, E.M. de Jager, N.M. Temme, R.J. de Vogelaere. *Colloquium elliptische differentiaalvergelijkingen, deel 2.* 1970.

10 J. Fabius, W.R. van Zwet. *Grondbegrippen van de waarschijnlijkheidsrekening.* 1970.

11 H. Bart, M.A. Kaashoek. H.G.J. Pijls, W.J. de Schipper, J. de Vries. *Colloquium halfalgebra's en positieve operatoren.* 1971.

12 T.J. Dekker. *Numerieke algebra.* 1971.

13 F.E.J. Kruseman Aretz. *Programmeren voor rekenautomaten; de MC ALGOL 60 vertaler voor de EL X8.* 1971.

14 H. Bavinck, W. Gautschi, G.M. Willems. *Colloquium approximatietheorie.* 1971.

15.1 T.J. Dekker, P.W. Hemker, P.J. van der Houwen. *Colloquium stijve differentiaalvergelijkingen, deel 1.* 1972.

15.2 P.A. Beentjes, K. Dekker, H.C. Hemker, S.P.N. van Kampen, G.M. Willems. *Colloquium stijve differentiaalvergelijkingen, deel 2.* 1973.

15.3 P.A. Beentjes, K. Dekker, P.W. Hemker, M. van Veldhuizen. *Colloquium stijve differentiaalvergelijkingen, deel 3.* 1975.

16.1 L. Geurts. *Cursus programmeren, deel 1: de elementen van het programmeren.* 1973.

16.2 L. Geurts. *Cursus programmeren, deel 2: de programmeertaal ALGOL 60.* 1973.

17.1 P.S. Stobbe. *Lineaire algebra, deel 1.* 1973.

17.2 P.S. Stobbe. *Lineaire algebra, deel 2.* 1973.

17.3 N.M. Temme. *Lineaire algebra, deel 3.* 1976.

18 F. van der Blij, H. Freudenthal, J.J. de Iongh, J.J. Seidel, A. van Wijngaarden. *Een kwart eeuw wiskunde 1946-1971, syllabus van de vakantiecursus 1971.* 1973.

19 A. Hordijk, R. Potharst, J.Th. Runnenburg. *Optimaal stoppen van Markovketens.* 1973.

20 T.M.T. Coolen, P.W. Hemker, P.J. van der Houwen, E. Slagt. *ALGOL 60 procedures voor begin- en randwaardeproblemen.* 1976.

21 J.W. de Bakker (red.). *Colloquium programmacorrectheid.* 1975.

22 R. Helmers, J. Oosterhoff, F.H. Ruymgaart, M.C.A. van Zuylen. *Asymptotische methoden in de toetsingstheorie; toepassingen van naburigheid.* 1976.

23.1 J.W. de Roever (red.). *Colloquium onderwerpen uit de biomathematica, deel 1.* 1976.

23.2 J.W. de Roever (red.). *Colloquium onderwerpen uit de biomathematica, deel 2.* 1977.

24.1 P.J. van der Houwen. *Numerieke integratie van differentiaalvergelijkingen, deel 1: eenstapsmethoden.* 1974.

25 *Colloquium structuur van programmeertalen.* 1976.

26.1 N.M. Temme (ed.). *Nonlinear analysis, volume 1.* 1976.

26.2 N.M. Temme (ed.). *Nonlinear analysis, volume 2.* 1976.

27 M. Bakker, P.W. Hemker, P.J. van der Houwen, S.J. Polak, M. van Veldhuizen. *Colloquium discretiseringsmethoden.* 1976.

28 O. Diekmann, N.M. Temme (eds.). *Nonlinear diffusion problems.* 1976.

29.1 J.C.P. Bus (red.). *Colloquium numerieke programmatuur, deel 1A, deel 1B.* 1976.

29.2 H.J.J. te Riele (red.). *Colloquium numerieke programmatuur, deel 2.* 1977.

30 J. Heering, P. Klint (red.). *Colloquium programmeeromgevingen.* 1983.

31 J.H. van Lint (red.). *Inleiding in de coderingstheorie.* 1976.

32 L. Geurts (red.). *Colloquium bedrijfsystemen.* 1976.

33 P.J. van der Houwen. *Berekening van waterstanden in zeeën en rivieren.* 1977.

34 J. Hemelrijk. *Oriënterende cursus mathematische statistiek.* 1977.

35 P.J.W. ten Hagen (red.). *Colloquium computer graphics.* 1978.

36 J.M. Aarts, J. de Vries. *Colloquium topologische dynamische systemen.* 1977.

37 J.C. van Vliet (red.). *Colloquium capita datastructuren.* 1978.

38.1 T.H. Koornwinder (ed.). *Representations of locally compact groups with applications, part I.* 1979.

38.2 T.H. Koornwinder (ed.). *Representations of locally compact groups with applications, part II.* 1979.

39 O.J. Vrieze, G.L. Wanrooy. *Colloquium stochastische spelen.* 1978.

40 J. van Tiel. *Convexe analyse.* 1979.

41 H.J.J. te Riele (ed.). *Colloquium numerical treatment of integral equations.* 1979.

42 J.C. van Vliet (red.). *Colloquium capita implementatie van programmeertalen.* 1980.

43 A.M. Cohen, H.A. Wilbrink. *Eindige groepen (een inleidende cursus).* 1980.

44 J.G. Verwer (ed.). *Colloquium numerical solution of partial differential equations.* 1980.

45 P. Klint (red.). *Colloquium hogere programmeertalen en computerarchitectuur.* 1980.

46.1 P.M.G. Apers (red.). *Colloquium databankorganisatie, deel 1.* 1981.

46.2 P.G.M. Apers (red.). *Colloquium databankorganisatie, deel 2.* 1981.

47.1 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60: general information and indices.* 1981.

47.2 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 1: elementary procedures; vol. 2: algebraic evaluations.* 1981.

47.3 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 3A: linear algebra, part I.* 1981.

47.4 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 3B: linear algebra, part II.* 1981.

47.5 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 4: analytical evaluations; vol. 5A: analytical problems, part I.* 1981.

47.6 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 5B: analytical problems, part II.* 1981.

47.7 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 6: special functions and constants; vol. 7: interpolation and approximation.* 1981.

48.1 P.M.B. Vitányi, J. van Leeuwen, P. van Emde Boas (red.). *Colloquium complexiteit en algoritmen, deel 1.* 1982.

48.2 P.M.B. Vitányi, J. van Leeuwen, P. van Emde Boas (red.). *Colloquium complexiteit en algoritmen, deel 2.* 1982.

49 T.H. Koornwinder (ed.). *The structure of real semisimple Lie groups.* 1982.

50 H. Nijmeijer. *Inleiding systeemtheorie.* 1982.

51 P.J. Hoogendoorn (red.). *Cursus cryptografie.* 1983.

CWI SYLLABI

1 Vacantiecursus 1984 *Hewet - plus wiskunde.* 1984.

2 E.M. de Jager, H.G.J. Pijls (eds.). *Proceedings Seminar 1981-1982. Mathematical structures in field theories.* 1984.

3 W.C.M. Kallenberg, et.al. *Testing statistical hypotheses: worked solutions.* 1984.

4 J.G. Verwer (ed.). *Colloquium topics in applied numerical analysis, volume 1.* 1984.

5 J.G. Verwer (ed.). *Colloquium topics in applied numerical analysis, volume 2.* 1984.