Programming Language Systems

Editors
M.C. Newey, R.B. Stanton
and G.L. Wolfendale

This book was published by ANU Press between 1965–1991.

This republication is part of the digitisation project being carried out by Scholarly Information Services/Library and ANU Press.

This project aims to make past scholarly works published by The Australian National University available to a global audience under its open-access policy.

Programming Language Systems

Editors
M.C. Newey, R.B. Stanton
and G.L. Wolfendale

Australian National University Press
Canberra 1978

First Published in Australia 1978

Printed in Australia for the Australian National University Press, Canberra

© 1978 M.C.Newey, R.B.Stanton, G.L.Wolfendale and the several authors, each in respect of the paper(s) contributed by him; for the full list of names of such copyright owners and the papers in respect of which they are copyright owners see the table of contents at page vii of this volume.

This book is copyright. Apart from any fair dealing for the purpose of private study, research, criticism, or review, as permitted under the Copyright Act, no part may be reproduced by any process without written permission. Enquiries should be made to the publisher.

National Library of Australia
Cataloguing-in-Publication entry

Programming Language Systems.

ISBN 0 7081 0493 2

1. Programming (Electronic computers) - Addresses, essays, lectures. 2. Programming language (Electronic computers) - Addresses, essays, lectures. I. Newey, Malcolm Charles, ed. II. Stanton, Robin Barrington, joint ed. III. Wolfendale, Garth Leslie, joint ed.

001.642

Library of Congress No. 78 60544

United Kingdom, Europe, Middle East, and Africa: Eurospan Ltd., 3 Henrietta St, London WC2E 8LU, England
North America: Books Australia, Norwalk, Conn., USA
Southeast Asia: Angus & Robertson (S.E.Asia) Pty Ltd, Singapore
Japan: United Publishers Services Ltd, Tokyo

Designed by ANU Graphic Design
Cover illustration by Ian Sharpe

Preface

The papers in this volume were the basis of talks given at a conference of the same name held at the Australian National University from 24th to 25th of February, 1977. The submitted manuscripts were subjected to minor editing with an eye to consistency of conventions. Professor Dijkstra's fluent talks required only minimal changes in the conversion to written form; the alterations were at the level of removing remarks provoked by a temperamental sound system.

Contents

1 *The Development of Programming Methodology*
Edsger W. Dijkstra 1

2 *A Formalism for the Derivation of Programs*
Edsger W. Dijkstra 15

3 *Phrase Structures in Pascal*
Lloyd Allison & W. David Wilde 29

4 *Data Transformations and Program Transformations*
Chris J. Barter 39

5 *Lessons from Automatic Preprocessing of Symbolic Computing Problems* John A. Campbell 61

6 *An Implementation of Janus*
K. Robert Elz & Peter C. Poole 71

7 *Data Abstraction Facilities*
Jan B. Hext 89

8 *A Review of Proposals for Introducing Dynamic Arrays into Pascal* Barbara P. Kidman 107

9 *Operating System Command Languages - Some Recent Developments* Cliff B. Mason 119

10 *What's So Special about LISP?*
Peter W. Milne 143

11 *Semantics - The Scott Era*
Malcolm C. Newey 149

12 *On the Semantic Characterisation of Some Advanced Control Structures* Paul A. Pritchard 159

13 *Towards a Programming System*
Ken A. Robinson 167

14 *Recursion as an Alternative to Back-tracking and Nondeterministic Programming* Jeff S. Rohl 177

15 *Some New Techniques for Verifying Programs*
Rodney W. Topor 191

16 *Computer Language Systems and Cognitive Processes*
Garth L. Wolfendale 203

17 *An Entomological Approach to the Design of Programming Language Systems* Garth L. Wolfendale 221

Contributor's Affiliations 235

1

The Development of Programming Methodology

Edsger W. Dijkstra

[This paper is a transcription of Professor Dijkstra's first talk.]

I have been asked to keep this performance less technical than the next one, and I have decided to do that by first of all giving you some historical introduction into what happened with respect to programming methodology so that you may get a feeling for the nature and significance of its development. I hope to illustrate some of the consequences with two small programs, one of them being known to you, the other probably unknown. I will develop these programs with a certain amount of handwaving. You must forgive me that I do so; tomorrow I hope to show how the handwaving done today can be made quite hard, if so desired.

Sometime in the fifties I was programming very hard. In many respects it was a very sobering activity. I still remember that in a very heavily used sine routine that I had made, after a year of usage a bug was found, and lo and behold we then investigated it very very carefully and discovered that out of the two billion arguments that the sine routine could accept there was exactly one going wrong, by a most unhappy build-up of rounding errors which caused an undetected overflow. I had a few more experiences of that kind, and by the time that I also made a compiler which was then in daily use, I got seriously worried. About 1960-61 I got the feeling that creating confidence about the correctness of your program could very well be the hardest part of a programmer's job. It took a long time before I could share my worries with others; essentially it took up to 1968 at the NATO Conference in Garmisch where the software crisis was actually openly admitted. Ironically enough we have also to thank IBM for this. They announced the 360 in 1964. I studied its functional specifications for a full week and was absolutely horrified, because it was patently clear that it was impossible to make a decent operating system for that machine. Well, by '68 it was also clear to other people that there were some problems. I was quite amazed to hear one year later that the USSR had decided to build its Ryad series as a bit compatible copy of the 360. I thought that to be the greatest American victory in the Cold War.

It was only when the software crisis, as such, was generally admitted that it became possible to try to do something about it on a larger scale. The years went by. Initially, in the infancy of programming methodology we were very much concerned about aesthetic considerations. We tried to find out which programs we liked and which we hated and we tried to formulate why we liked them and why we hated them and slowly in the beginning of the seventies it became more

mathematical (harder). There was a moment when it became obvious that proofs of correctness should play a central role in the whole game and since then programming has emerged as a tough engineering discipline with a strong mathematical flavour.

Not all people like that conclusion; that's quite understandable of course, because of its implications. One of the implications presumably is that the subject as such - how to write good programs - is too difficult for what is called the average programmer. Thus the implications have serious social consequences. Well of course one cannot refuse to draw a conclusion because its consequences are unattractive from some point of view.

Now what happened during those years? One of the first major steps was the Invariance Theorem which was introduced already in 1968 (but not immediately noticed) by C.A.R. Hoare. Let's consider some sort of repetitive construct: if G holds do S and repeat S until G doesn't hold any more. If this terminates you are through with the application of S, and to give you an example, the wisdom of many physicians consists of the following prescription: while you are suffering from a headache (take one aspirin and wait 30 seconds). Now it so happens that this process is certain to terminate (one way or another) and it's sure that upon termination you are not suffering from a headache. There's slightly more and that is that any state of affairs that is not ruined by the action taken, if true to start with will be true afterwards, for instance, age > 20. (We all know that taking an aspirin doesn't make you any younger.) Upon completion you know that age > 20 still holds. Now that's the idea of the invariant relation: whatever statement about the state of affairs that's true to start with and is not destroyed holds upon completion of the repetitive construct. That was a theorem, the power of which was not immediately realised. It derives its power from the fact that you can make a statement regardless of the number of times a step is repeated. As we discovered later, it can even be used very fruitfully in all those cases where the number of iterations is not uniquely determined by the initial state.

Another theorem that I may formally prove tomorrow is what is called the Linear Search Theorem. It is a very very silly one and it is so obvious that every one agrees with it intuitively, yet it is very useful to know it. It is the following theorem: Suppose you have the program:

```
begin i := 0; while b(i) ≠ 0 do i := i+1 end
```

If this terminates you know i equals the minimum value ≥ 0 such that $b(i) = 0$. This linear search theorem tells us that if we are looking for the minimum value satisfying some criterion, if we don't know any more, then we should investigate the values in increasing order. Alternatively, if we are looking for a maximum value satisfying some criterion we should do it downwards. Now simple as the theorem is (once it has been stated you understand it immediately), its knowledge has a profound influence, and I'm going to use that theorem in the development of a program for the following problem:

Let c(1)...c(n) be a permutation of the numbers 1,...,n but not the alphabetically last one - where the alphabetic ordering is simply that 1 precedes 2, 2 precedes 3, 3 precedes 4,... So, of all the n! permutations, first come all the ones starting with a "1" then all the ones starting with a "2" etc. It is asked to write a little program that will transform the array c(1)... c(n) into the immediate alphabetic successor. This is a program such that if we started with 1, 2, ..., n and we apply it n! - 1 times we end up with n, n - 1...3,2,1, which is the alphabetically last one. It would be a technique for generating all permutations. I have written down a permutation of the 10 decimal digits - each of them occurs once and only once, and in this case you can equate the alphabetic order with the numeric order. The permutation is 4905268731. If you regard it as one number then the question becomes what is the smallest larger ten digit number in which each digit occurs only once. Let us first figure out if we can do it ourselves. Given that number - what it the smallest larger ten digit number in which each digit occurs only once? Is anyone of you going to propose the answer?

Answer given: 4905271368

How did we do that? Well, it is not too difficult because we want the immediate successor. So we want to keep as many digits constant as we can. But if, for instance, we keep everything constant to the left of the digit 3 then the only thing we can do is permute the remaining ones, and you see because these are in descending order if you permuted them you would always get a smaller value. So. we have to change more. If we do it between 6 and 8 we still have a decreasing sequence on the right. But if you make the barrier between 2 and 6 you see that 6 has a larger right hand neighbour so now we can permute the tail and increase its value. Now suppose that this is the element c_i - how do we define that i? Well, the value of i is the maximum value (because you want to leave as much constant as you can) such that $c_i < c_{i+1}$. It's the right-most element with a larger right hand neighbour. Now given that formulation of the value of i a knowledge of the linear search theorem immediately dictates how we establish the value of i; that is:-

```
i := n - 1;  (because it must have a right hand neighbour
              and n - 1 is the maximum value for which
              this occurs)
```

<u>while</u> $c_i \geq c_{i+1}$ <u>do</u> i := i - 1 ;

Now we are certain that c_i is less than c_{i+1}. You see that to determine the value of i is a practically mechanical affair as soon as we had written the definition. The next step is to determine which value will replace c_i; let it be c_j. i.e. how do we find that 7? Well, it has to replace the c_i so we are looking for the c_j that first exceeds c_i, (because this is the first digit that changes and it must increase because otherwise we don't get a successor) but we are looking for the smallest such c_j because we want to increase the value as little as possible. So we discover that now we have to

scan this tail in order. (Because we wish to have the smallest we have to investigate the c's in increasing order.) But we have just established that the tail was monotonous, therefore investigating these values in order of increasing value means trying them in the order of decreasing j (we have nicely used the monotonicity). Again, it is practically a mechanical affair, the maximum value of j is clearly n and as long as $c_j < c_i$, j is decremented (we need the smallest c, and so have to investigate the c values in increasing order).

```
j := n;

while c_j ≤ c_i do j := j - 1;
```

So now we know which value is to replace c_i. Here comes the first moment of invention and it is the idea that we had better place that 7 in position i; but what then do we do with the 6? We swap in c the elements i, j.

```
c: swap (i,j)
```

In the example, we now have 4905278631.

Now that is OK and the only thing we have to do is to sort the tail in increasing order because we want the minimum value. But the tail was in decreasing order to start with. The question is: is it still in decreasing order after we have done this swap? I claim yes, it is. The swap has not destroyed the monotonicity of the tail and therefore we don't need to do a complete sort; a reflection suffices. So i and j become i+1 and n respectively, and after this initialisation as long as i < j we swap in c the elements i and j and afterwards i,j are incremented and decremented respectively; that completes the reflection of the tail independently of whether the tail has an even or odd number of values. So here is the whole program!

```
begin
    i := n-1;
    while c_i ≥ c_i+1 do i := i-1;
    j := n;
    while c_j ≤ c_i do j := j-1;
    c:swap (i,j);
    i,j := i+1, n;
    while i < j do
      begin c:swap (i,j); i,j := i+1, j-1 end
end
```

Ain't it a beauty?

Now this is an example of exploiting the linear search theorem. The first effort of proving the correctness of programs wasn't attractive at all. Proofs were ugly and laborious and it was felt to be an incredible unimposable extra burden on the programmer's shoulders. Things however changed drastically by the time that it was discovered that one is looking for a program and a proof that establishes that program's correctness. In this sense you want to have a matching pair, the program and its correctness proof. It turns out that if you start with a given program without any annotation, and you try to find the correctness proof you may have a very hard time - particularly because the program may be wrong. But even if it is correct you have a very hard time. However the other way round is much, much easier: given the shape of the correctness proof, deriving the program to which this correctness proof is applicable is nearly mechanical. The whole thrust that developed after that discovery was that we tried to develop the program and the correctness proof hand in hand (as a matter of fact the correctness proof running ahead of the program).

The next discovery was a technique for defining the semantics of both programming languages and simple programs which at first seemed unusual. Suppose you have a program and that the machine is well defined etc. We all know that if you start the machine in an initial state it will end up if it's a terminating program in a final state. Therefore we are very much used to viewing the question as: this being the initial state, the input, what will be the final state: the output? We have a feeling that that I/O relation or initial/final states relation establishes the semantics of the program completely. It does! However, stated in this way it's a very clumsy way to manipulate, because given a rather lengthy complicated program and an initial state we all know that to establish what the final state is will be very hard - so hard that in practice we invoke a computer to establish that relation. Suppose that we want to get some output. Which input would provide it? Suppose that we want to get at some class of final states. Which is the set of initial states that would lead us there? Now this backwards way of reasoning, amazingly enough, although awkward for a machine is extremely amenable to symbolic treatment on the blackboard. When you start thinking about it, you discover it's not so amazing at all; you may be aware of Wilkinson's work on error analysis which also works backward - the idea is if you have to compute y which is a function f of x - due to rounding errors you find yourself computing y' which is a distortion: you haven't the f of x but you have computed the distortion. For a long time people have tried to make estimates of the difference between y and y' asking themselves "How wrong is my answer" That was very hard. It turned out much easier to answer the question: of which perturbed argument would y' be the exact answer? So you regard y' as f at a perturbed argument and then impose upper bounds between the true argument and the perturbed argument. If this difference is well within the measuring accuracy of the input you had better be content.

Now here comes the method which I would like to give now so that it can sink in for tomorrow. If S is a statement and

R is some sort of desired post-condition we denote by wp(S,R) the weakest precondition such that activation of S is certain to establish R. For instance let S be "man empties whisky bottle" and let the desired final condition be "man drunk". What is the weakest precondition? (Of course we are considering a binary man - either he is sober or he is drunk, and we are considering a binary whisky bottle - either it is empty or it is full). Quite clearly, the weakest precondition is "bottle full or man drunk" (maybe he's drunk to start with). If you know the corresponding weakest precondition for *any* post condition then you know everything about the semantics of that activity. e.g. what is the necessary and sufficient condition such that a man emptying a whisky bottle will end up with the moon being square? The condition is that it be square to start with. Which just implies (I'm saying that the moon is circular) that if you instead wish that upon completion of this activity the moon be circular it should be circular to start with. This in itself tells us that the man drinking whisky doesn't change anything with regard to the shape of the moon.

In terms of these weakest preconditions we can define a programming language... there is a statement that you all know for which the weakest precondition that upon completion R will hold is that R holds to start with, whatever R may be. We know that statement very well. We know its names: *skip*, the empty statement, No-op. Now my claim is that this definition completely gives the semantics of doing nothing. I even go further; by now we have defined a programming language. A rather primitive one - you can only write one program in it and the program can't do a thing. Here is another statement for which, by definition, the weakest precondition (regardless of the post condition) is *false* - it means there exists no initial state that will ever have anything established. We know that one well, it's called *abort*.

By now we have defined a bigger programming language. We can write two programs - the one can do nothing and the other one can't even do that. I will define one more, the assignment statement x := e. By definition the weakest precondition is a copy of R in which each occurrence of x is replaced by the expression e. In words: if finally relation R should hold for the final value of x then initially that same relation should hold for the value of the right hand side before the assignment. In order to convince you that indeed this captures what we are talking about (under the assumption that you have some intuitive appreciation of the assignment statement), consider x := x+1 and let our post condition R be $x \leq 10$. By definition we conclude that we rewrite this with each occurrence of the variable in the assignment replaced by the right hand side, giving $x+1 \leq 10$, i.e. $x \leq 9$; and indeed this is the necessary and sufficient condition to guarantee that upon completion of this assignment this post condition will hold.

Now I think that we have now enough gadgetry on the blackboard so that I can derive one other program - binary search. Presumably you have not seen it. If I had more time I would now stop speaking and ask you to take a blank sheet of paper and code for yourself the binary search as I am

going to specify it. I will use capital letters for constants and small letters for variables.

We are given $A_1 < A_2 < \ldots < A_n$ and X such that $A_1 \leq X < A_n$ (A_n may be viewed as some sort of plus infinity). We are requested to establish the relation R so that the boolean variable *present* is equal to $\exists\ i : 1 \leq i \leq n$ and $A_i = X$. Now this is true or not depending on the given values. Well if it's true this boolean should get the value *true*. If this is false the boolean *present* should get the value *false*. I should now stop speaking and ask you all to code this in your favourite programming language and the first one who has a solution in which he has sufficient confidence that he dares to show it on the blackboard should do so. I will not do this experiment this time. I have done it many, many times and it takes too long. I've done it once with about 30 experienced programmers. It took 20 minutes before the first solution appeared on the blackboard, and for the remaining part of the 45 minutes I had given to the experiment people in the audience argued whether the solution was correct of not. Now you may laugh about it but it's kind of alarming. Particularly if you are addressing an audience of professors of computing science - it takes very long then.

The first remark is that it will be very hard to establish that X occurs somewhere in the array without also finding the value i. So the intermediate goal R_1 could be to establish the value i such that $A_i = X$. That is, of course, too strong because it's not necessarily known that such a value exists. So we must make this a little bit weaker and we compute i in such a way that it will point to a place in the array where X would be if it occurred. So let R_1 be $A_i \leq X < A_{i+1}$. Then my problem has been solved because the following assignment will do it:

$$present := (A_i = X).$$

Well, clearly if $A_i = X$ then it assigns rightly the value *true* to *present*. However if $A_i \neq X$ we know that $A_i < X < A_{i+1}$. In that case the fact that the array is ordered implies that X doesn't occur to the left and it doesn't occur to the right either. So if we succeed in establishing R_1 our problem has been solved.

Another question is how do we establish R_1? Now I ask you to remember the use we made of the invariance theorem. If something is true to start with and we manage to have a repetitive mechanism that doesn't destroy its truth, then it will be true afterwards. Now in the case of a repetitive construct that invariant relation must be established to start with, and it must be true at the end. So it captures both the initial and final state of the repetitive construct - it's a generalisation of both and we see one of the tricks of finding the invariant relation. Because initially the array is ordered with $A_1 \leq X < A_n$, and $A_i \leq X < A_{i+1}$ will be our final state R_1 that we are heading for. Now compare the indices of the A's and leave open their differences. Plug in variables there, i and j, and let's take $P : A_i \leq X < A_j$ as our invariant relation, because it has the delightful property that the only difference between P and R_1 is that j is

not necessarily equal to i+1. But we see that (P and j = i+1) clearly implies R_1. So we can use it at the end, (this P) and it's also trivially established.

So now we have the shape of our program. After the assignment i,j := 1,n, P has been established and the only thing we have to do is some sort of repetitive construct where as long as j differs from i+1 we better do something. Upon completion we still have P but also the negation of j ≠ i+1 and we have established that R_1 holds: P and j = i+1. So that's the structure. Now the only thing we have to do is ... well you start with i = 1 and j = n and you have to bring them closer together keeping P invariant. I propose that we take some additional value h and we'll figure out what that will be later. We'll see that it will be in between i and j. We can do that because what we also keep invariant is that i < j, and if i < j and i+1 ≠ j then they must differ by at least 2, so there exists a value in between - really in between.

Clearly by the assignment i := h you bring them closer. The question is will we destroy P? (This whole theory was under the assumption that P was indeed an invariance relation.) And now comes the formal machinery. If we wish P to hold at the end we better figure out the weakest pre-condition such that i := h is certain to establish P. Let's do it: wp("i := h", P) - well, what was the recipe? - rewrite the expression and replace each occurrence of the variable assigned to by the right hand side. So let's do it. We find $A_h \leq X < A_j$. So this is the weakest precondition such that the assignment i := h will establish P. Now P holds to start with, so the $X < A_j$ part is indeed implied by P as it occurs in P itself. $A_h \leq X$ isn't, so the text had better check that. Only in the case that $A_h \leq X$ are we entitled to do i := h. We do exactly the same thing with the other one: j := h. You will find the left hand inequality is implied by P but $X < A_h$ is not. So only if $X < A_h$ are you allowed to do j := h. So we use an alternative construct:

<u>if</u> $A_h \leq X \rightarrow$ i := h ▯ $X < A_h \rightarrow$ j := h <u>fi</u>

If a test is true you are allowed to do the corresponding statement. But behold, the tests are each other's complement so there is always a way through it - this will not lead to abortion.

Now our program is correct - the only thing I have to do is to invent an expression for h. How about i-1? It's O.K. but of course if the array is long and if the value of X occurs in the far far end it takes you a very long time. So we had better redo it to j-1! Well, the obvious solution is (i+j) <u>div</u> 2 and that concludes, with some handwaving, a design of binary search.

```
begin
    i,j := 1,n;
    do j ≠ i+1 → h := (i+j) div 2;
                 if A_h ≤ X → i := h
                  [] X < A_h → j := h
                 fi
    od;
    present := (A_i = X)
end
```

There are all sorts of people who, when they see this program, start arguing that if at an earlier stage you already hit the value X exactly you had better jump out because you have already found it. As such this program has been used very often as an argument for intermediate exits. It is. The supposed need is, however, a complete fallacy because a 3-way decision of course takes more time than a 2-way decision. Therefore if you include an intermediate exit (for the case that X is found early) it will give you a fixed overhead. Now if you start to make estimations of the expected number of steps that you might save you will discover that there is a probability of a half of saving nothing, a probability of a quarter of saving one step and a probability of one eighth of saving two steps, a probability of one sixteenth of saving three steps. This series converges very, very quickly. The result is that although the expected value of your gain is bounded your loss is unbounded. So it is no improvement at all.

So much for the display of the informal development of programs, which is however a way of programming which as you might understand has been vigorously influenced by the formal techniques. You see one of the reasons that the formal techniques suddenly became so popular (when we realised that they could be applied) was that they immediately established a greater consensus of opinion than would ever have been possible with eloquence or anything of that sort.

There was a time that WG2.1 (an IFIP working group) thought that it could design a language; of course it couldn't, as is clearly shown by ALGOL 68 which came out of that group. The annoying thing during those discussions was that we really had at that time no yardstick for comparing quality of programming languages. Someone came along with a feature and there was another bright boy who said "Oh, I can use that!" He showed it and he said "Oh that's a useful feature - that's included". This was the appeal made to convenience. But all too often "convenience" was confused with the conventional - it still is.

You would never see a proposal for a programming language design with the requirement that it will be easy to learn - for God's sake keep the programmer uneducated. And it was in that uncertain period that suddenly somebody said "But we must be able to define the semantics of your constructs, formally.

If that formal definition becomes cumbersome please regard that as a warning that's not to be ignored." This did hold for programming language constructs, also for programs as such. In a sense it was quickly accepted to regard a program as more beautiful if its correctness proof was more straightforward. I think indeed that that is a fair statement. To put it in another way: Amazingly enough even a number of pure mathematicians regarded in their own work elegance as a dispensable luxury. My claim is that elegance is in computing science, and computing practice, not a dispensable luxury but a must, a matter of life and death.

I would like to leave room for questions, remarks, objections or you name it.

Question

You've got a program on the board there. Each line of that program is an operation followed by a statement in curly brackets which is meant to be a property that is preserved. Am I to believe that there's no human error in any of those lines?

Answer

No, of course not. There are some mathematicians who claim that they never make an error. But they are fools. Claims to infallibility should be left to experts in that field, like the Pope. Of course there are things like writing errors, printing errors, reproduction errors, even simple trivial oversights. If you make them they may be very hard to find. But really there are two completely different types of bugs - the one which as soon as you have located it you say "Oh that's stupid, that's just a spelling error." And there's another one which as soon as you have located it you say "Gee! I better redo my homework." Obviously such proving techniques are no guarantee even against the logical errors but they help a tremendous lot. So that the probability that you have to redo your homework is greatly reduced - drastically or fantastically.

For the writing errors and spelling mistakes - there are different techniques to guard against those. There is really only *one* technique. That one technique is some form of exploitable redundancy. You may have a certain amount of redundancy in your programming language itself, for instance, that it must be syntactically correct. You may have plugged some tests in your program itself - the type of defensive programming you are driven to when you know that the program will be executed by an unreliable machine. The kind of course you are taking then is such that it is highly improbably that if a program is erroneous it will still be executed in the right way. The other trick that I have found extremely effective, if it really matters, is coding in duplo - with two persons sitting opposite each other, claiming to produce the same text. Then both texts are punched and compared mechanically. That's the way in 1960 we made the ALGOL 60 compiler. G.S. Honevelt and I were sitting opposite to each other and we claimed to write the same booklet; had them both punched. That's the same way we made the THE operating system.

It's quite a simple trick by which you weed out the great majority of writing errors, which are easy to remedy but sometimes extremely hard to find.

Question

Did you say that in computing, elegance is a matter of life and death?

Answer

Yes.

Question (continued)

My question is: how do we measure elegance?

Answer

Why do you want to measure it? Mathematicians agree about the elegance of proofs.

Question (continued)

So are you saying it's sociological?

Answer

There is a whole theory about the sociology of science which really boils down to the fact that doing science is nothing more than reaching a consensus. I may quote Lin Yutang who makes all sorts of useful remarks: among the thirty three joys of living he lists for instance - "to see someone's kite lying broken". But he has a very inspiring paragraph on the aims and purposes of education and makes it quite clear that the purpose of university education is to learn good taste in knowledge. What students have to do primarily according to Lin Yutang is to learn what to hate and what to love. And I think he is right. You learn what is a beautiful program. You have to learn what is a beautiful proof. If you can measure it with a measure on which you all agree then it is O.K. You see, there is something that's called metrics. Have you ever seen the writings of Tom Gill? Or are you far away enough, here? I won't go into it. His writings are terrible because what he proposes are all sorts of techniques of attaching a number to a program and then saying that it measures some quality. But that is like the soft science nonsense, where people have to fill in a great number of forms and you count and add up the divide and then you find an intelligence quotient. The basic assumption underlying the soft sciences is that as soon as you divide two large numbers the quotient is meaningful. Then something has been produced. Thank goodness something has been measured. No! A *number* has been produced - that's something completely different. Physicists know how to measure something. But before a physicist proposes a method to measure some property, he has done extensive experiments to justify the introduction of the quantity at all. And he defines the quantity in such a way that thereafter a proposed technique for measuring can be judged as inadequate or adequate. However as soon as you

start blending measurement and definition you are on a very slippery slope. So I think I would rather not measure it.

Question

I have a question about the invariance relation. You claim that it is easier to discover the invariance relation if it's not given, like in this problem, rather than discover the process first and then discover the invariance relation after.

Answer

Oh yes. It's much easier to choose an invariance relation first and then formally derive the program from this than the other way round. As a matter of fact the other way round the problem is undecidable. The second advantage is that as programmers we have many responsibilities, but two main responsibilities have to do with correctness and efficiency. Now in the case that you have important concerns you must try to separate them and deal with each in turn. And both correctness and efficiency are so important that we had better deal with them in turn. You must realise that efficiency is only defined with respect to the implementation, with respect to the computational histories. However correctness need not be; and what I hope to do tomorrow is to show how the semantics of a programming language can be defined without any reference to an underlying computational model. That is, even forgetting that the program texts we are manipulating also admit the interpretation of executable code. If you take the goal of separating your concerns seriously then while making a program you must be able to forget the class of computational histories. By means of this technique it is possible to design programs where the class of computational histories really baffles the imagination.

For a long time it has been said that programming is very hard because you had to visualise everything that could happen. One of the greatest advantages is that if you forget that programs can be executed at all you don't make that admission. I'm very serious. One of the most common ways of wasting your time and being misled that I have found is that when people have to cope with a class of cases, they hope to understand that class by a set of instances. They hope that, miraculously, that set was representative. But if it wasn't, there is a bug in the program ("that's a very special case you see.'"). The thing that goes wrong is always a very special case. But the only effective way of thinking about a large class of instances is never to focus your attention upon a single instance but always deal with the class in terms of its definition and that's what you are doing when you play the game this way. You are slowly pushing to the background of your mind that there are long sequential programs. Historically this formalism started in the design of operating systems where the amount of concurrency and nondeterminacy completely baffles the imagination. If you try to visualise what's going to happen with such a multi-dimensional process ... well, my head can't cope with it. I'm absolutely convinced that in the case of a repetitive construct one should first design and choose the invariance relation and then the program follows fairly naturally.

Question(Summary)

I can see how you apply your methods for small programs. I can't see how you push them through for something bigger.

Answer

I regard what you said not as an objection but as a very valid observation. Obviously we have played extensively with very small programs because otherwise you can't carry out the experiments. And the technique of program development, even language choice etc. has been the result of thousands of little programs being made by different people and compared and discussed. Of course, this is not our final goal. The original goal was inspired by the fact that all sizeable systems were error loaded, bug ridden. The initial intention was (on account of the observation that at that moment we did not know well enough how to make sizeable programs) to figure out what the education of a competent programmer would encompass. For five to seven years we have played with small programs. Eventually of course we have to tackle larger ones.

The major difference between a traditional mathematician and a computing scientist as far as I am concerned is that a traditional mathematician has a tendency to abstract from problems of scale and not to distinguish between being able to solve something and being in principle able to solve something. But we have. And it's all very nice if you can give a straightforward treatment of 8 line programs. The truth is, of course, that in 800 line programs you do it the same way. The problem however is that it is very expensive to do significant experiments with larger programs - very time consuming. I have pushed the limit slightly further for myself and I have already discovered a few of the causes that may ruin the approach when you try to tackle a large program. One of the reasons why these little programs worked was that with integers you have a ready-made notation. As soon as you have a program that's doing something more general you find that you have to invent your own notation - just to describe the subject matter of the computation. And it so happens that a choice of a wrong notation can easily double the length of your formulae. The larger your program the more opportunity you have of choosing a wrong notation - the more opportunity you have of doubling the length of your formal labour. If you have done it five times you have lengthened the whole thing by a factor of 30. In all probability you have done a PhD thesis! The game, of course, will only work provided that the formal labour can be kept proportional to the text of the program.

The amount of reasoning should be proportional to the length. It should not be proportional to length squared. If you are not careful you will discover that it is proportional to e^{length}. There are a few very encouraging things I can show you tomorrow. The one and only fundamental theorem about the alternative construct, which has two great virtues: First of all, it's quite clear that the formal labour you have to do is proportional to the number of alternatives. Well, that's reasonable. Of much greater importance is that the kind of statement that you have to prove about the

individual alternatives is of the same degree of complexity as the conclusion you can then make about the whole construct. And therefore the whole construct itself remains as easy to use.

Von Neumann, in his writings, remarks that in the case of simple mechanisms it's often easier to describe how they work; in the case of complicated mechanisms, it is usually easier to describe what they do. Now you will discover the following thing: a mechanism is only useful by virtue of its properties. It may be very hard to achieve them. But when, finally, the thing has been wrapped up, it should be a nice smooth thing again. Otherwise there is no point in making it.

Question (continued)

I think my question was that if you go backwards it looks like it will be difficult to follow the rigorous approach on a big system or big program.

Answer

No, I don't think so. If you make an operating system you can fairly well make overall statements which are quite clear - there may be no deadlock, each program has to be executed correctly, no program may get lost, no infinite overtaking. You can place a number of criteria. You tell yourself that if the operating system satisfies those conditions you will be very, very pleased.

Question (continued)

Are those invariants in the design of the operating system the same as for a program?

Answer

Yes. And for any large systems there are very clear requirements, and if there are not you should not start programming it. As soon as you are programming a banking system you have to take care of the law of conservation of money.

2
A Formalism for the Derivation of Programs
Edsger W. Dijkstra

[An edited transcript of Edsger Dijkstras second talk]

In the last century there was a fierce battle between Kronecker and Cantor. The battle was won by Kronecker. As a matter of fact, Cantor ended up in a mental asylum. I am afraid that I should be regarded as a dyed-in-the -wool intuitionist, because even Kronecker went too far for my taste. A famous quotation by Kronecker is "God created the natural numbers, everything else is man's creation". If I really think about it, I come to the conclusion that the good Lord was no fool. He didn't make all the integers; he just created an awful lot of them, but enough. There was a little fallen angel, turned into a devil, who said to mankind "Well, you see the pattern there ..."

We had this morning a survey of Scottery. I think it's an absolute mistake and I'll tell you why. Once you have swallowed the continuum, there is not the slightest objection to introducing infinity as well. The whole of plane geometry is extremely simplified as soon as you introduce a line of infinity where every pair of lines has at least one point of coincidence etc. Indeed, once you have swallowed the continuum the introduction of infinity simplifies matters. But as long as you continue to live in a discrete world, making that discrete world infinite creates more problems than it offers solutions. Whether an axiom set is complete or not, whether it is consistent or not - all these questions only come up as soon as you allow stacks of infinite height or depth. (I think good old Kronecker would enjoy this speech.)

Our promise was that today we should play a more formal game, so I am going to define not the syntax but the semantics of a programming language (leaving the syntax to BNF addicts). You have already seen that the statement *skip* is defined by the following definition: for all post-conditions R, the weakest pre-condition which ensures that the post-condition will be established is exactly R to start with. We have also seen the assignment statement. Recall that $R_{E \to x}$ is by definition a copy of R (providing it is written in the first order predicate calculus) in which each occurrence of the variable assigned to, i.e. x, is replaced by the right hand side E.

The next thing I am going to do is to define for you the semantics of the semicolon. Whenever we consider something as an object composed out of parts the interesting properties of the composite object should be functionally dependent on the interesting properties of the components. The way of composition should describe that functional dependence. So under the assumption that we fully know the semantics of S_1

and S_2, which are the two parts, we will now define the semantics of $S_1;S_2$. By definition it is given by wp("S_1; S_2",R) = wp(S_1,wp(S_2,R)). This looks like functional composition. A mathematician will immediately recognise that this makes the semicolon an asymmetric but associative coupling of mechanisms. If you have an operational appreciation of what a semicolon does for you I can justify this choice: eventually we wish that R holds and what we did finally was to execute S_2 and according to the definition of the semantics of S_2 this final execution of S_2 will indeed establish R provided that the initial state satisfies wp(S_2,R) - that was the definition of the weakest precondition. However, if prior to that we have executed S_1 and asked ourselves "Well, what should be the initial state so that the execution of S_1 indeed establishes that ?", the answer is that prior to this there should hold the weakest precondition so that S_1 establishes the desired condition. Plugging in the weakest pre-condition for S_2 in the position of the post-condition for S_1 is no more than identifying the final state of S_1 with the initial state of S_2. That is what is usually regarded as doing things in succession.

You can take this justification in two ways. One is that you see that that formal definition of the semicolon indeed captures what I meant. I prefer to look at it the other way around and to say that this is what I mean, and then I see that there is a feasible implementation of it. This is a shift of attention which is symptomatic of the fact that programming is becoming a more-or-less grown-up science. In the old days we regarded it as the function of our programs to instruct our machines. Logic was regarded as a descriptive science. By now, we regard it as the function of our machines to execute our programs.

That is a definition. Now I need three abbreviations. This is one of them. I will denote by IF the following construct

$$\underline{\text{if}}\ B_1 \rightarrow S_1 \,[\!]\ \ldots\ [\!]\, B_n \rightarrow S_n\ \underline{\text{fi}}$$

It is my alternative construct. For all i, $B_i \rightarrow S_i$ is called a guarded command. B_i is called its guard and S_i is a sequence of statements only eligible for execution in those initial states where the guard is true. By those fat bars (which of course have been borrowed from the vertical bar of BNF) we denote an *unordered* set of guarded commands. The idea is that in order to execute this alternative construct the guards are evaluated. If one or more of them are true one of the corresponding statements is selected and executed. If none of the guards is true then the whole construct leads to abortion. We need one further abbreviation. I will use BB for B_1 or B_2 ... or B_n. BB means at least one of the guards is true; BB indicates that the construct itself will not lead to immediate abortion.

I now give the formal semantics. First of all, at least one of the guards should be true. Furthermore, for all j in the range $1 \leq j \leq n$ (i.e. for all the alternatives) the truth of the guard (which operationally means that the

corresponding S_j *could* be selected) had better imply the weakest pre-condition such that S_j (which is now selectable) will indeed establish R. More formally, wp(IF,R) is defined as

$$\exists_j: 1 \leq j \leq n: B_j \text{ \underline{and} } (\forall j: 1 \leq j \leq n: B_j \Rightarrow wp(S_j,R)).$$

Now with this definition a not-too-difficult theorem can be proved: If, for all j such that $1 \leq j \leq n$, (P and B_j) $\Rightarrow$ wp(S_j,R), then (P and BB) $\Rightarrow$ wp(IF,R). That is, if the combined truth of P and each guard ensures that the corresponding statement will establish R, then we can conclude for the whole construct that if P holds to start with and at least one of the guards is true then indeed the whole alternative construct will establish R. Note that this is a theorem.

The above theorem is an extremely attractive one. In order to use it we have to establish the antecedent, which means that n conditions of the form P and $B_j \Rightarrow$ wp(S_j,R) have to be proved but the n proofs are completely independent (you can deal with them one after the other). If you had n mathematicians at work, as I have at Mathematics Inc., then you could just give an alternative to each of your subordinates, and if each proved their part of the antecedent, you could conclude about the whole construct a conclusion of the same degree of complexity as the parts.

What eventually we wish to do is to build, in a reliable fashion, more and more complicated programs by the trick of combining a number of parts into something that from then onwards will be regarded as a whole. The technique is applied iteratively; at each stage a number of wholes are regarded as components and combined again. If the overall property was an order of magnitude more complicated than the properties of the parts, then repeating this mechanism once or twice would really explode your formalism. One of the virtues of this alternative construct is that the description of the relevant properties of the whole is of the same degree of complexity as that of its parts.

Let us use this. I shall try to stick to the rule that I will use capital letters for constants, small letters for variables and P, Q, R for predicates. Let it be required to establish the relation (m = X or m = Y) and (m $\geq$ X and m $\geq$ Y). I don't know whether for all values of X and Y this equation has a solution or not. We'll discover that. If this, viewed as an equation in m, would not have solutions for certain values of X,Y we could never write a program that would always establish it. There are a few sparks of invention indeed. One of these is that we should consider assignments to m; otherwise it must be rather hard to achieve. Now we can try all sorts of things to assign to m. Personally I like 13. But let us make it 11. m := 11. We can ask ourselves "Will this do the job?". What is the initial condition under which this statement will indeed establish R? It is by definition (11 = X or 11 = Y) and 11 $\geq$ X and 11 $\geq$ Y. Now I have a program, by plugging in that weakest pre-condition as the guard: if (11=X or 11=Y) and (11$\geq$X and 11$\geq$Y) $\rightarrow$ m := 11 fi.

Here I have a program, and if it terminates successfully it will beautifully establish R. However, as you might realise, there are plenty of values of X,Y that will make this single guard BB false. This would lead to abortion. So in order to make BB weaker we had better provide more alternatives. Can anyone suggest another alternative?

Answer given from audience: *X.*

That's a very good idea. Let's forget about the program just done. Spark of invention: we are considering what would happen with the assignment m := X. Again you ask yourself "Under what circumstances would this establish R ?" By the definition of the axiomatics of the assignment statement we just rewrite R replacing m by X. We find that it is (X = X <u>or</u> X = Y) <u>and</u> (X ⩾ X <u>and</u> X ⩾ Y). Both (X = X <u>or</u> X = Y) and X ⩾ X are by definition true - forget about them. The only thing that remains is X ⩾ Y.

So here we go. The latest condition should be put into the guard and we get the following program: <u>if</u> X ⩾ Y → m := X <u>fi</u>. You may realize that BB is not identically true - depending on the values of X and Y this program may lead to abortion. (e.g. if X < Y). So again we had better weaken BB - by providing another guard. We don't need to do the whole thing again because as you all have noticed R is symmetrical in X and Y. Fortunately without further thought we can provide another alternative: Y ⩾ X → m := Y. Our program now reads

<u>if</u> X ⩾ Y → m := X ▯ Y ⩾ X → m := Y <u>fi</u>.

Now, is this program better? Yes, it's perfect - because at least one of the guards is true; so BB is identically true; abortion is excluded, and we have proved for any values of X and Y this equation in m has a solution (we have even given a constructive proof of that existence).

As I said, for any values of X and Y at least one of the guards is true. Maybe both are true, in which case I don't care which one. For example, if X = 7 and Y = 7 then m becomes 7 whatever way it is done.

Question from audience: Is *true* a possible value of a guard?

Yes. For example, if you write <u>if</u> *true* → S <u>fi</u> and plug that in the definition of the semantics, you find that this is semantically equivalent to S.

Question continued: Why not, in the program you have just developed, let the second guard be *"true"*?

Because then it would always be allowed to choose the second guarded command. You see, the fat bar separates an <u>unordered</u> set of guarded commands.

Question continued: I understand that; but if you alter your testing order ...

No! No! No! No! Mind you, there is a completely operational view to it. Please notice that this formula which defines the semantics of the alternative construct is completely symmetrical in n alternatives. I am not going to commit myself to how the possible non-determinacy is being solved. You can make that arbitrary decision that they will be evaluated in order and the first one that is true will be picked. In this case you have the conditional expression that has been propagated by LISP. But according to my definition, it would be fatal to write *true* here. We could not then conclude that it would be correct! Besides that, it would destroy the symmetry, wouldn't it? That, as a matter of fact, is one of my aesthetic objections to LISP.

We have now seen an alternative construct which is not fully deterministic. You can argue whether this program is non-deterministic - because the final state is always the same one. For any pair of X and Y there is only one solution. If there is any non-determinacy involved it's not a property of the program but a property of the implementation. We could write a truly non-deterministic program, e.g. if we wish to code x := ±1. The following program would do it:

$$\underline{\text{if}}\ \mathit{true} \rightarrow x := 1\ [\!]\ \mathit{true} \rightarrow x := -1\ \underline{\text{fi}}.$$

The only thing I can say is that upon completion x is ±1 (both guards are true).

A little bit more notation: I will denote by DO the repetitive construct:

$$\underline{\text{do}}\ B_1 \rightarrow S_1\ [\!]\ \ldots\ [\!]\ B_n \rightarrow S_n\ \underline{\text{od}}.$$

Operationally it means the following thing: go on executing the alternative construct:

$$\underline{\text{if}}\ B_1 \rightarrow S_1\ [\!]\ \ldots\ [\!]\ B_n \rightarrow S_n\ \underline{\text{fi}}$$

until this actually leads to abortion - when all the guards are false. This is a much more complicated construct. $H_0(R)$ is defined to be R and non BB. For $K > 0$, $H_K(R)$ is wp(IF, $H_{K-1}(R)$) or $H_0(R)$. Now wp(DO,R) is defined as $\exists K : K \geq 0 : H_K(R)$. Operationally, you can interpret the $H_K(R)$ as such a machine state that the repetitive construct will finish within at most K steps and will end up in a state in which R holds (but in which BB doesn't - because otherwise the thing would make at least one more step). The recurrence relation guarantees that, in at most K steps, the thing will establish R - either it will do so in 0 steps or it will do it in at least one step, the result of which must be to establish the condition that no more than K -1 steps are then necessary. Then the definition of the whole thing is that there exists a number of steps that will be sufficient to establish R - obviously that's necessary. (This, by the way, completes the definition of the programming language I am talking about. So if you don't write too big, it fits on the back side of an envelope.)

Now in order to formulate the relevant theorem about the repetitive construct I have to introduce still one more gadget and that is wdec(S,t); S is a statement and t is an integer function defined on the state space. wdec(S,t) is the weakest pre-condition such that the execution of S will decrease t by at least one. (t is an integer function so the proviso "at least one" can be omitted.) That is the way you can interpret it operationally. It is <u>defined</u> as $wdec(S,t) = wp(S, t < \gamma)_{t \to \gamma}$. (The subscript $t \to \gamma$ indicates γ is replaced by t.)

The special case that will suffice for our purposes is the simple case of an assignment statement, x := E. It follows from the above that wdec(x := E,t) is simple $t_{E \to x} < t$. (Of course, $t_{E \to x}$ is just the term t with each occurrence of x replaced by E.) In this case the predicate transformer that gives you the weakest pre-condition for the actual decrease is quite simple. Now I have the gear for formulating the important theorem.

From (i) P <u>and</u> BB ⇒ wp(IF,P)

(note that P holds to start with and if the repetitive construct makes a further step then that step is certain to establish P. It is this condition that gives P the role of the invariant relation that we mentioned yesterday.)

(ii) P <u>and</u> BB ⇒ wdec (IF,t)

(i.e. there must exist an integer function t which is certain to be decreased by at least one. In this connection t is called the invariant function.)

(iii) P <u>and</u> BB ⇒ $\exists K : K < t$

(i.e. t should be bounded from below.)

we can deduce

P ⇒ wp(DO,P <u>and</u> <u>non</u> BB)

(i.e. if P holds to start with the repetitive construct will terminate with P still holding and all the guards false.)

From a point of view of usage this is a still more delightful theorem - because a lot of the stuff you have to prove in order to use it is more complicated to formulate than the final conclusion. The intricacies of the construction as such justify its termination and then you can forget about it. Note that the antecedent which has to be proved about the alternative construct is exactly the type of conclusion that our previous theorem was able to make about this construct, so we are on safe ground.

Now we will formally derive a program with a repetition in it. Originally I intended to derive a program computing the greatest common divisor. However, you have seen this program in another part of this Conference. I remember the fact that one or two years ago there was a symposium in

Saarbrucken on the semantics of programming languages and eleven of the fourteen lecturers used Euclid's algorithm as the carrier of the talk. So I thought that this was only going to be the second time it has been used today. I will now need the fact that wdec(x := E,t) is $t_{E \to x} < t$. In order to modify the presentation our target will be not to compute the GCD of 2 numbers but of three. Our target relation R will be that x = GCD(X,Y,Z) where we know already that X, Y and Z are positive integers. (There are days when I am not even quite sure that the negative integers exists.)

This morning we were shown subgoal induction as a simplified form of the method of inductive assertions, which is what I'm using - it can't be very simplified because the two are completely equivalent! The remark was made that the kind of relation P that we shall formulate in the course of our repetitive solution has to be invented. This is true - you have to invent it. There are some rules for such inventions. If I know nothing about the GCD and I finally must have a relation R then I am looking for a relation P such that P _and_ _non_ BB will imply this. In that case a relation P with GCD(X,Y,Z) = GCD(x,y,z) is indeed an invention. However, it's not surprising to have it because: what do you know? At the beginning you know nothing. The standard way of establishing P is x,y,z := X,Y,Z. If you do that, that is the only conclusion you can draw, so it is about the only relation in which GCD(X,Y,Z) occurs that you can write down at all. Now we know more because X,Y,Z are positive and we had better keep x,y,z that way (during this program we can forget about negative numbers). So let's take this as P (our invariant relation).

The idea is that the GCD(X,Y,Z) = GCD(x,y,z) _and_ x > 0 _and_ y > 0 _and_ z > 0. What we are now going to do is to massage the value of the triple x,y,z in such a way that the GCD does not change. I massage until the GCD is trivially known; I know that one of the properties of the GCD is that GCD(x,x,x) = x. Again a spark of invention is required - we have to decide what kind of operations we are going to consider. If you know anything about the properties of the GCD it is strongly suggested to have a look at x := x-y. Now we ask ourselves what is the weakest pre-condition such that x := x-y will establish P because this is one of the targets of my construct. Here we go: it requires that GCD(X,Y,Z) = GCD(x-y,y,z) and that x-y > 0 and that y > 0 and that z > 0. You know that all are implied by P except for x-y > 0. So as far as keeping P invariant is concerned we are led to the program _do_ x-y > 0 → x := x-y _od_.

Now obviously that is not enough; in the meantime we have established P. But the requirement of the theorem forces us to invent an integer function which is non-negative and which is decreased. We are looking for an integer function whose being non-negative is implied by P _and_ _non_ BB. We only have three variables which are non-negative, x,y and z, and a suitable candidate for t seems to be x+y+z; at least we don't destroy symmetry. Notice that P implies that t is non-negative. We now ask ourselves under what conditions do we know that x := x-y will effectively decrease x+y+z? According to our definition we have to replace the variable assigned to by

the left hand side, giving $x-y+y+z$ which is to be less than the original expression $x+y+z$. Having done that we see that therefore we require $y > 0$. So we have derived that $x+y+z$ is decreased by at least one by the statement $x := x-y$ provided $y > 0$. Do we need to worry about that? No, because $y > 0$ is implied by P. Here we have a program which if you start with X,Y,Z has termination guaranteed. The question is: can we conclude R? Regretfully we cannot. The program is certain to terminate with P still valid and $x \leqslant y$, i.e. <u>non</u> BB.

We can't conclude R. What does that mean? Simply that we are establishing P <u>and</u> <u>non</u> BB, but we can't conclude R as we desired. That means <u>non</u> BB is too weak; therefore BB is too strong; therefore BB should be weakened - this is done by providing other alternatives.

Question from audience: Does the formalism tell you that we can't conclude R?

No, that is what mathematics tells us. If you can conclude from P and the negation of the guard $x > y$ that $x = GCD(X,Y,Z)$ then that's fine with me but I don't have the mathematics available to do that. The only way I know how to do that is that I know (P <u>and</u> $x = y = z \Rightarrow R$). Just the knowledge of the triple (x,y,z) and $x \leqslant y$ is insufficient. So, you need to understand the problem you are dealing with.

I think we have done all invention needed. For P is a symmetric relation in x,y,z. We have used only two variables and there are three variables. So the only decent way to deal with the problem is to permute them cyclically - this is the best way not to destroy any symmetry you had. So add $y > z \rightarrow y := y-z$, and there is a similar argument that this will decrease t and not destroy P. Similarly we need $z > x \rightarrow z := z-x$. So the whole program is

```
x,y,z  := X,Y,Z;

do x > y → x := x - y

[] y > z → y := y - z

[] z > x → z := z - x

od
```

We now have enough to get x,y,z equal to the GCD of X,Y,Z. Termination is certainly in the above algorithm guaranteed, and P has not been destroyed. The fact that termination is guaranteed implies all the guards end up false hence eventually $x \leqslant y \leqslant z \leqslant x$, i.e. they are all equal. Ain't it a beauty?

You may not be aware that the execution need not be deterministic. There are all sorts of states depending on x,y,z such that two of the guards can be true: in this case one of the corresponding statements is selected for execution. I don't care which one. So, this supports our earlier state-

ment that whatever can be said of significance and meaning in the field of computing science can be illustrated with Euclid's algorithm, even non-determinacy.

This was the amount of formal treatment that I wanted to show to you.

Question from audience: If, say, the first two guards are true, does this mean you take the old value of y on the right hand side?

No! A decent implementation is that if two guards are true one of the statements with the true guard is executed - I don't care which one. There is no form of simultaneous relaxation or anything of that sort. It wouldn't harm in this case but don't let yourself be misled by that. Suppose that you go to the market with the intention of buying a watermelon. There are two watermelon prices: one lot at $10 each, the other lot at $25 each. We ask ourselves what is the condition that you can buy one with the money in your pocket? It is that *cash* ≥ $10. This gives the guard which allows you to buy one of the cheaper lot. The other guard is that cash ≥ $25. So if you send someone to such a shop with the instruction to buy a watermelon then the program he has to execute is

<u>if</u> *cash* ≥ $10 → Buy a $10 watermelon
[] *cash* ≥ $25 → Buy a $25 watermelon <u>fi</u>

If he has less than $10 in pocket both guards are false; he can't buy a watermelon; he has to abort. Suppose now that you enter the shop with $30 in your pocket. You have the choice of cheap or expensive. But clearly you can't buy both.

Let's get back to our GCD example. Suppose the first and the last guards are true. If the first is selected then x is made smaller so the last guard is only made "truer". In this algorithm there is never any competition, so if you have a choice it does not matter. It is not as if choosing one condition will invalidate the other - which you have in the case of the watermelons; this being the pattern you have in all forms of resource sharing.

It would take too long to do, but it can be proven that, although a program written down according to my mechanism may be non-deterministic to the extent that a whole bunch of answers is possible, *if* I start computation in an internal state such that I can prove termination, *then* the collection of possible outputs is finite.

Further questions from audience:

Question: Can I ask whether your existential quantifier in your theorem is classical or intuitionist?

It is very classical.

Question: Do you *mind* that it is classical?

Personally, I insist on knowing the upper bound on the

corresponding variable value.

Question: Why did you introduce that weak form of non-determinacy rather than the meaty sense where the right choice has to be made at every step?

That last form is alright; *provided* that either you have a mechanism that will explore *all* the possibilities (so that the sequence of right choices is among them) *or* there is a kind demon with sufficient clairvoyance to guide the machine toward your goal.

The reason that my form of non-determinism was introduced (more or less by accident) was that it seemed to make formal derivation of programs easier. Furthermore it enabled me to map whole sequences of otherwise trivially equivalent but different programs on to the same text. There are all sorts of programs where there can be variations which are just the programs you get when you resolve the non-determinacy in a very specific fashion. From a point of view of efficiency, if one choice is much more efficient than another then of course you had better strengthen the guard and weed out the non-determinacy. The second reason for introducing it was that it was the kind of non-determinacy with which I was familiar on account of my experience in the design of operating systems.

Question: It seems that you introduce some notion of parallelism, in that in DO statements you can execute all statements in which the guard is true.

No! That is a very operational statement. If I were to design a programming language that allowed for a certain amount of concurrency I would do it differently. Particularly in the case of the repetitive construct, guards being true and a statement being selected may force guards to become false, and must if the thing is going to terminate. So, in a sense with the repetitive construct you have exactly that competition problem we were talking about.

Question: My question concerns the purpose of that programming language. You said in your examples that a spark of invention is required to massage the problem into this form. Is there some reward for having the solution program in this form?

Yes. Both the program and its verification should be enjoyed. I'm not being facetious. I'm terribly serious here. Those facts about proofs of termination are such that doing three steps backwards you will discover the repetitive construct, that I illustrated with Euclid's algorithm. Mind you, I'm not interested in GCD's at all, but the structure of these theorems, which are exactly the kind of things that you have to prove in operating systems. The immediate analogy is between the guards in this sequential program and the synchronising conditions in a multiprogramming system. The only difference is that in a multiprogramming system you don't wish to stop - your target is that at least one of the guards remain true so that the whole thing can go on forever. One of the things that this shows is that what in a sequential

program is your desired state, i.e. termination of a repetitive construct, is in the case of an operating system what you try to avoid - in that case it is called deadlock. Mathematically the phenomena of deadlock and termination of that little program are exactly the same problems.

Question: It seems to me that by focusing on the weakest pre-condition you are contributing to the elegance of the solution program. Is that true or not?

I think so. When designing programs it is not unusual that the actual initial state satisfies a more complicated, stronger condition than the weakest pre-condition that is needed for termination. i.e. the program works under considerably wider circumstances than you intended to use it for. Indeed, the formulation of those wider circumstances has a much smoother boundary to it.

Question: Suppose in writing a program in your style I carried out a lot of logical deductions and I made a fundamental error. Might I not end up with a program which was harder to debug?

I don't have the experimental evidence to say either yes or no. I don't think so, by the way, because if you play this game well, in my experience, independent of the size of the program you will find yourself writing an essay about ten times as long as the raw code in which it culminates. On account of this factor some people write the method off as impossible, because they conclude that then it is ridiculous for programs of one hundred thousand lines to have a million lines of comment.

This argument is ridiculous however. The justification of design being 10 times as long as the raw code in which it culminates just means that the final raw code is a very dense condensation of our intellectual labour. That is a good thing because the raw code has to be fed into the machine and storage is expensive.

Question: Has your work been applied to the automatic proving of the correctness of programs?

Yes and no. For instance, I taught the linear search theorem in January 1974 in Albuquerque and very quickly it was built into an automatic verification system with great success. Probably it has inspired automatic verifiers.

I myself have not the slightest experience with verifiers. First of all I don't see the point of automatic verifiers. It strikes me as putting the cart before the horse - as I said you have to make a matching pair of program and proof and it is much more effective to choose the proof and derive the program than the other way around. Secondly, I don't feel inclined at all to try to automate what I like to do myself. In a sense an automatic verifier is a contradiction in terms because a proof to me is a reflection of my understanding and if there is anything I cannot delegate, whether to someone else or to a machine, it is understanding something. I can give up and say "Please will you understand this on my behalf?", but to understand something and to be

responsible for your actions can not be delegated - like falling in love.

Question: One problem I have with this is that examples cited always have a very clear cut criterion for whether or not the results are correct. Take quite a different sort of program such as the simulation of, let us say, a barber's shop. I have no criterion like yours for whether or not it is correct.

Correctness is always with respect to certain specifications. What you are pointing out is that there are all sorts of situations where you are not sure what your specifications are. Then the very first thing to do is to struggle until you have a set of rigid specifications and you tell yourself "This is the goal my piece of machinery has to satisfy". That is exactly the purpose of your specifications. When a person tries to make a program and says "Well, I'm not exactly sure what the specifications are, and whether it is okay or not" it is essentially like proving a theorem but not caring about the exact formulation of the theorem ....

Question continued: Where I'm not quite clear is if you are working backwards from something, the something you have is usually an end product of the program ...

I start by being eloquent so that by the time I am working backwards my goal has been very well specified in good English or symbolic logic or a mixture of the two, because from then onwards I can use a formal discipline. I don't deny the problem of the type when some customer enters your shop and he has a problem and he has to be helped. I have done that for years. The first thing you do is to talk with the man for hours and hours in order to help him to understand his problem and to come up with a clear cut definition of the goal. When he says "Yes, that's what I want." you can start to code.

Question: In your book you make the point that rather than presenting some programs you would rather present the mechanisms by which you arrive at those programs. I agree with you 100% but you have presented a programming language which I find fascinating and I'd like to get to the thought processes which led you to it. For example, did you decide on non-determinism early on or did you do some development in parallel which led you to believe you couldn't find an equivalent set of axioms for a traditional language?

I know where the guarded commands in a sense were invented - it was a very short period in November 1973 at a WG2.3 Conference in Blanchland. It took place during a few evenings. On one hand you can say their choice is a result of two evenings - it is equally fair to say their choice was a result of 43 years. I could try to reconstruct what went through my head but it would be too long a story to tell now. I know a few fatal mistakes I could have made. One of the ideas I considered for about half an hour was that obviously there is no point in continuing the repetition when the state does not change any more, so for a while I played with the idea that a repetitive construct should terminate under

those conditions in which the execution is idempotent. However that is a disastrous idea because you can have a cycle and oscillate between the two.

What eventually did it was the realization that I knew how to look at Euclid's GCD program in two ways: $x > y \rightarrow x := x - y$ or $y > x \rightarrow y := y - x$. Here we have two sequential programs, one is an x decreaser and the other one is a y decreaser. They have to be synchronised in such a way that the x and y remain positive. So they continue until deadlock occurs. That's the way you view this program from the world of operating systems, from which the whole experience of the formal derivation of the synchronisation conditions was taken over lock, stock and barrel. I had done this many years before, and it only took a ridiculously long time to discover that all this experience could be shipped over to the normal task of writing sequential programs.

Question: Have your methods been used on real problems, problem of big magnitude?

I admit that up to now we have not learnt to apply these methods from beginning to end to a very complicated problem. But they have been used in serious applications. Sometime ago I visited one of the Burrough's plants and gave a talk in the morning. Three guys who had listened to it had just completed an interrupt handler and during lunch they decided to try to prove its correctness. That afternoon they concluded that it involved the analysis of a big directed graph. So they wrote a little program to generate and then analyse the directed graph. The program was run overnight and the output revealed 3 bugs in the interrupt handler. These were repaired, the program was given new input and before noon the correctness proof had been carried out. Very wisely, these guys focused their formal labour on one of the most sensitive parts of the design - because if there's a bug in the interrupt handler, that's not nice.

That was two years ago. My impression is that as more and more people try them the applicability and significance of these techniques will grow. You can't expect to change the world overnight. First of all one has to get used to the idea. Secondly, it requires a considerable amount of experience to do it well.

3
Phrase Structures in Pascal

Lloyd Allison & W. David Wilde

ABSTRACT

This paper describes an extension to Pascal which enables the user to define syntax as phrase structures, and to invoke a top-down fast-back analyser. No modifications to the Pascal compiler are necessary, user programs being preprocessed into legal Pascal. Most of the implementation is written in Pascal, requiring only three small assembly code routines. The technique described should be applicable to other programming languages.

1. INTRODUCTION

The processing of highly structured data, such as program text, is a non-trivial task in "conventional" programming languages such as Fortran, Algol or Pascal. Such processing is much easier in a language with phrase structures or pattern matching facilities, examples being the Brooker-Morris Compiler-Compiler [4], or BCL [3], which automatically attend to the necessary housekeeping.

Because Pascal [7] has many desirable features for both teaching and systems programming, and because it is available on many machines (in particular the Melbourne University Computer Centre, CDC CYBER 73, and the Computer Science Department's Interdata 8/32) the idea of a Pascal-based syntax system was very attractive. The primary aim was to produce a compiler tool that would be readily usable by newcomers to Pascal, although other applications are anticipated.

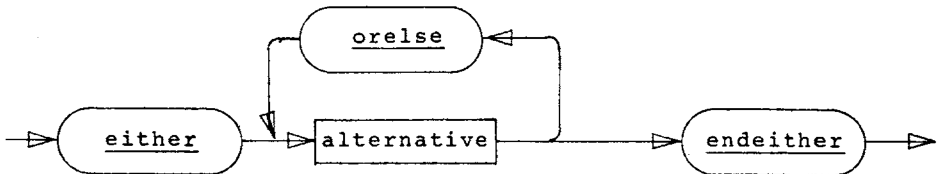
With similar motivation, Housden has shown a method of incorporating BCL-style phrase structures into Fortran [2] and Algol 60 [1], and it was a Pascal equivalent to the Algol system that was required. However, the Algol system uses label-valued procedure parameters to direct the back-tracking, and these are absent from Pascal where a '<u>case</u>' replaces complex jumps. In spite of this a clean and simple solution was found, and should be "transferrable" to any (stack-based) language with a '<u>case</u>' or equivalent, such as Algol 68 or BCPL.

In common with all those mentioned above, the present system effects a top-down fast-back parser. Detailed descriptions and realistic applications of such parsers can be found in Gries [5], Knuth [6] and Housden [1],[2], and are not included in this paper. Only simple examples and explanations are given in section 2 for completeness.

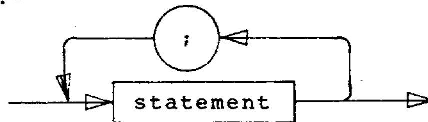
2. NEW FACILITIES

The heart of the phrase system is a new (to Pascal)

'either' control structure:



alternative:-



The 'either' structure becomes a new alternative to statement [7] in effect, and 'either', 'orelse', and 'endeither' become extra reserved words.

Briefly, an alternative is a sequence of statements following an 'either' or an 'orelse'. An alternative can 'succeed' by simply executing to its end, or 'fail' by calling the procedure 'reject'. When the 'either' is executed, alternatives are tried in sequence until one succeeds, in which case the 'either' succeeds. If no alternative succeeds, then the 'either' fails, in which case the alternative at the (dynamic) 'either'-level above fails. An alternative may, of course, contain possibly recursive procedure calls, and the procedures so called may contain further 'either's and/or 'reject's.

Failure of an alternative involves back-tracking or resetting the "parsing machine" to the state it held on entry to the 'either'; the next alternative is then attempted. In particular the current character pointer 'chpoint' is reset, thus nullifying any input by the failed alternative.

Any procedure or function may 'reject' the current alternative, there being no distinction between 'phrases' which may do this and procedures, as occurs in the Algol extension [1].

The other major aspect of the system is text input. The function 'inch' delivers the next character on the input file and advances over it. The function 'peep(n)' looks n characters ahead without advancing. These system routines operate on a hidden buffer along which backtracking automatically takes place when necessary.

Note that due to slight personal preferences the particular syntax of the 'either' was provided with a closing bracket ('endeither'), and that an alternative is a sequence of statements rather than a single statement. In this way the 'either' syntax is closer to the Pascal 'repeat ... until' than to the 'if ... then ... else ...'.

Example

```
A1      function digit : integer;
A2         var i : integer;
A3         begin i := ord(inch)- ord('Ø')
A4              if (i<Ø) or (i>9) then reject
A5                                else digit := i
A6         end;
A7      function fixed : integer;
A8         label 99;
A9         var i : integer
A10        begin i := digit;
A11            99 : either i := i*10+digit;
A12                        goto 99
A13                 orelse fixed := i
A14                 endeither
A15        end
```

This standard example defines the syntax and value of unsigned decimal integers. Should the first call of 'digit' by 'fixed' (line A10) result in 'digit' failing then 'fixed', and any alternative calling 'fixed', will fail. If the second call (inside the 'either', line A11) on 'digit' fails then only that alternative of the 'either' will fail and processing will continue at the next alternative (line A13); if the call succeeds the following statement (line A12) is obeyed.

Following the notation of BCL where possible, some standard procedures are:

```
procedure osp;
     label 99;
     begin 99 : either if inch = ' ' then goto 99
                       else reject
                orelse null
                endeither
     end (* can be more efficient! *)
```

The execution of 'reject' on line 3 causes line 5 to be obeyed. Procedure 'null' is exactly that, and so cannot fail. 'osp' absorbs any spaces that are next on the input stream.

```
procedure nl;
     begin if ord(inch)<>eolncode then reject
     end
```

'nl' fails unless the next "character" is an end of line. Standard Pascal does not include a newline, carriage return, or similar character. Consequently an ordinary character, say '$', is "sacrificed" and 'eoln's are detected and converted to this character by the input routines.

```
procedure string(a : alfa; n : integer);
     var i : integer; b : array(1..10) of char;
     begin unpack(a,b,1); (*Pascal,ref 7*)
           for i :=1 to n do
                if b(i)<>inch then reject
     end
```

'string' checks that the first 'n' characters of a 10 character string - 'a : alfa' - are present. Unfortunately Pascal does not allow "variable" length strings, which necessitates the second parameter.

As a further example:

```
program tidy(input,output);
      function month : integer;
             begin
                  either string('january...',7);month:=1
                         ......
                  orelse string('december..',8);month:=12
                  endeither
             end;
      function fixed : integer

             (as before, also 'osp' and 'nl'

      begin
           repeat osp; write(fixed : 2); osp;
                 either string('/.........',1); osp;
                        write('/', fixed : 2, '/'); osp;
                        string('/.........',1)
                 orelse write('/',month : 2, '/')
                 endeither;
                 osp;
                 writeln(fixed : 4); osp; nl
           until eof
      end.

      input:-                  output:-

           21 January1977           21/1/1977
      32/3/    1984                 32/3/1984
```

3. IMPLEMENTATION

Programs written in the phrase system are preprocessed into valid Pascal by a program written in that system. The following translation of the 'either' on lines A11 - A14 takes place

```
either        ------------>  [ begin eitherentry(1);
                             [ case nextalternative of
                             [     1:begin
    i:=i*10+digit; -->                  i:=i*10+digit;
    goto 99       -------               goto 99

orelse        ------------>  [          end;
                             [     2:begin
      fixed:=i   ----->                      fixed:=i
                             [              end;
                             [
endeither     ------------>  [     3:backtrack
                             [   end;
                             [   eitherexit
                             [   end
```

Note, the overall enclosing 'begin ... end' is simply to

make the result into a single Pascal (compound) statement.

'eitherentry', nextalternative', 'backtrack', and 'eitherexit' are system provided routines, that maintain a second 'system stack' (as distinct from the Pascal stack), which in fact exists in the Pascal heap. This system stack consists of a doubly linked list of blocks that contain 5 information words; the list is initially 'nil'.

'eitherentry' links a new block onto the top of the 'system stack' if necessary, and uses the top to store

(i) its own return link (see later 4),

(ii) the activation record base (ARB) pointer (to the Pascal stack) of the routine that contains the 'either' (see later 4),

(iii) the number of alternatives already tried in this 'either' i.e. zero,

(iv) the current input position, 'chpoint'

(v) its integer parameter (here 1) which is the textual level of 'either' nesting.

'eitherexit' simply pops one block of information from the system stack. The blocks remain doubly linked so that space is not lost to the program, but can be reused later.

'nextalternative' extracts the number of alternatives tried already in the 'either' (at this activation), adds one to it, and returns this as its result, i.e. 1,2,3,... .

'backtrack' pops one block of information, and then calls 'reject', thus failing the entire 'either' and rejecting the alternative at the level above.

Alternatives are enclosed in 'begin ... end' because the syntax of the Pascal 'case' only permits single statements as elements. Each converted alternative is preceded by its number in the order, as a 'case' label. Note that 'case' labels are not ordinary labels in Pascal.

'reject' firstly resets the input 'chpoint', as saved above in (iv). It then resets the Pascal stack pointers to their state prior to entering the 'either', by means of (ii) above. Lastly it exits to the return link saved in (i), which leads to reentry of the 'case', where 'nextalternative' selects its name-sake. The Pascal stack may have to be retracted as described because 'reject' may only be invoked after several levels of routine calls.

Total failure.

If the outermost 'either' fails then all the user's alternatives have failed and the input must be invalid. This can be detected as an attempt to 'backtrack' past the bottom ('nil') of the system stack and diagnostics can be printed and the run terminated. Calling reject from outside

any 'either' is also equivalent to total failure and the same termination occurs.

goto's

Jumps are not allowed into an 'either' and this is "checked" by the Pascal compiler as a similar restriction applies to the 'case'. However jumps out of structures - 'either's, procedures, etc. - are allowed. Just as the Pascal system may have to recover its own stack, so the system stack may need recovery, after such a jump.

After Housden [1], all user labels (not 'case' labels) are converted from, say

'99:' into '99:phrasetidy(n);'

where 'n' is the textual nesting of 'either's at the label. However here, the information saved by 'eitherentry' is quite sufficient to enable 'phrasetidy' to recover the system stack, and there is no need to distinguish phrases (procedures which may 'reject') from simple procedures to enable the former to (automatically) save auxiliary information.

Since a jump may bypass several normal procedure returns, the system stack may then contain blocks with saved ARB's that point too high in the Pascal stack, and these blocks can be detected by 'phrasetidy', since Pascal must have tidied its own stack on jumping to the label in question. There may also be blocks that have correct ARB pointers but that correspond to too deeply nested eithers; these can be detected by comparison with 'phrasetidy's' parameter.

Program structure.

In addition to the conversions described, the preprocessor must insert the few system routines and global declarations at the outermost level of the user's program. The routines can be inserted either physically or as precompiled externals. These names are included in the Appendix.

4. IMPLEMENTATION DEPENDENCE

Most of the "behind the scenes" routines are written in standard Pascal. For example, 'eitherentry' calls two 6-instruction assembler functions to produce its own return link and its calling routine's ARB. Being in Pascal, 'eitherentry' can use the heap, in particular.

Only 'reject' is written in assembler, as it must follow the dynamic record links in the Pascal stack, reset machine registers and perform a non-standard return. On the CYBER this requires 14 assembly instructions.

Even 'phrasetidy' is written in Pascal, as the system stack being in the heap, can be manipulated within the language. 'phrasetidy' also calls one the the assembler functions mentioned above.

The input machinery is written in Pascal, although coding it in Fortran or assembler could be desirable from the efficiency point of view.

Although written almost exclusively in Pascal, the phrase system does rely on Pascal-implementation features that are strongly implied, but not made obligatory, by the Pascal standard [7], e.g. that the stack grows upwards, and is "classically" organised. Thus probably only the 26 lines of assembler would need to be rewritten for a new machine and/or Pascal compiler - but this is not necessarily the case. However the fact that the rest is written in Pascal greatly adds to the clarity of its working, and would simplify any rewrite.

The preprocessor of course relies only on the system routines, regardless of their own implementation.

The implementation described here is under Release 2 of the Zurich Pascal compiler [9] on a CDC CYBER 73.

5. PASCAL AS A SYNTAX LANGUAGE

The well advertised absence of certain features from Pascal, for valid reasons [7], makes for slight difficulties in its use as a syntax language.

String quantities, in particular string routine parameters, must be of a constant length, which makes for slight inelegance in that the routine 'string' (§2) can only check fixed length (10 character) strings or portions of them, and requires a second parameter to indicate the significant portion. It would be a trivial matter, but contrary to the spirit of standard Pascal, to make the preprocessor convert arbitrarily long strings that are to be matched either into a series of calls on the existing 'string' routine, or into a table and a call on a hidden table driven routine. The latter possibility would be greatly eased by a table (vector) initialisation statement, which is available in certain dialects of Pascal.

The lack of an end-of-line character as such (instead there is an eoln predicate), makes it difficult to write syntax that may backtrack over end-of-lines, unless a "valid" character is sacrificed to represent end-of-line, the conversion taking place in the input routine. Care must be taken as this character will not be printed as an end-of-line by the Pascal output routines. This is a restriction caused by hardware considerations [7].

The Pascal heap, or non-garbage-collected free store, proved ideal for the phrase system stack. Initially this stack is 'nil'. If the stack grows "taller" than ever before, a new block is added "on top". Conceivably one particularly deep series of (possible recursive) 'either's could tie up a lot of storage, never to be released or reused, but this seems unlikely to occur in practice.

A syntax tree is not constructed automatically as the user's phrase structures are executed. However if routines

were provided for the user to explicitly construct one during parsing then the heap would be the ideal place to put the tree.

The absence of garbage collection means that there are no large hidden overheads in this sort of use of the heap.

6. CONCLUSION

Initially it looked as if the absence of label routine parameters would make the addition of phrases to Pascal virtually impossible. In fact it proved straightforward. There are a number of advantages to the method. The use of the '*case*' structure means that no hidden labels are generated (case labels are not ordinary labels), and jumps into an '*either*' are automatically disallowed. It proved unnecessary to distinguish between *procedures* (or *functions*) and *phrases*, thus minimising on the concepts added by the extension.

A similar technique should be applicable to any stack based language with a '*case*' or similar structure, such as Algol 68 or BCPL ('switchon'), although care would have to be taken over where to keep the system stack. In the absence of a '*case*' it should be possible to do the same thing with a 'computed *goto*' (say), but then hidden labels must be generated.

The extension "adds on" to Pascal very neatly. This is felt to be very important with regard to teaching, as top-down "automated BNF" systems such as BCL, reveal what is going on very transparently, and much of this value would be lost by a messy addition.

Firstly to be used to illustrate top down parsing, the general efficiency, availability, and style of Pascal mean that this tool should be suitable for realistic applications.

REFERENCES

[1] R.J.W. Housden, *An implementation of phrase structure in Algol*, Software Practice and Experience, vol. 2, 231, 1972.

[2] R.J.W. Housden, *Phrase structures in Fortran*, Computer Journal 14, 224-228, 1971.

[3] D.F. Hendry and B. Mohan, *A BCL Manual*, University of London Institute of Computer Science, 1968.

[4] R.B.E. Napper, *An introduction to the compiler compiler*, University of Manchester, Department of Computer Science.

[5] D. Gries, *Compiler construction for digital computers*, Wiley 1971.

[6] D.E. Knuth, *Top down syntax analysis*, Acta Informatica, 79, 1971.

[7] N. Wirth and K. Jensen, Pascal users manual and report, Springer Verlag, 1975.

[8] P. Naur (ed), Revised report on the algorithmic language Algol 60, IFIP, 1962.

[9] Wirth et al., Release 2 of the ETHZ Pascal 6000-3.4 system, ETHZ Zurich.

APPENDIX

```
procedure  eitherentry(n:integer)         30 lines Pascal
function   nextalternative:integer         5 lines Pascal
procedure  eitherexit                      6 lines Pascal
procedure  backtract                       4 lines Pascal
procedure  phrasetidy(n:integer)          14 lines Pascal

procedure  reject                          4 lines Pascal
procedure  rest; extern                   14 lines assembler

function   outarb:integer; extern          6 lines assembler
function   mylink:integer; extern          6 lines assembler

function   inch:char                      10 lines Pascal
function   peep(n:integer):char            7 lines Pascal
procedure  read next part                 17 lines Pascal
procedure  string(a:alfa; n:integer)       8 lines Pascal
```

system variables/constants

```
startofinput,  nchars, chpoint:integer
buffersize = 200 (*say*)
stackfront, freeframelist:stackrecord
stringlength = 10 (*of alfa*)
eolncode = 43 (*of '$' say*)
```

4

Data Transformations and Program Transformations

Chris J. Barter

1. INTRODUCTION

The background of ideas about programming and programs for this paper is the recent work of a number of people. Rather than a survey, we present a set of beliefs which underlies this work, and which (hopefully) captures the spirit of modern programming:

(a) Hierarchical structure is fundamentally important to the construction, transformation and understanding of programs which specify complex mechanisms or processes. It is fundamentally important because it gives representation to the (human?) process of abstraction - the co-ordination of "parts" into "wholes" and the separation of "how" from "what" - in such a way as to support indefinitely many levels of abstraction. (See Dijkstra [4], Dahl [3], Hoare [6], Wirth [12], and Stanton [10].)

(b) The characteristics of computing systems influence program design, and sometimes in a way which is difficult to integrate with the characteristics of the problem to be programmed. Thus it may be wise to factor these (conflicting?) requirements to firstly design the simplest, clearest, correct program we can, and then attempt to transform that into a final product which satisfies the remaining requirements (such as efficiency). Thus we may initially ignore major characteristics of a computing system (e.g. there is only one processor; central memory is limited; data transfer is costly), providing that we can later find a transformation to surmount the ignorance. (See Wirth [12], Stanton [10], Jackson [7].)

(c) There are interesting relationships between data structure and control structure in many programs, which is especially striking in programs which act (minimally) as recognisers for some given fixed data. The language Pascal is designed such that these two aspects are neatly juxtaposed in program texts; more dramatically the recursive descent specification of the Pascal compiler precisely mirrors the syntactic structure of the programs to be compiled. (See Wirth [14].) Also, Hoare [6] has identified the correspondences between data and control abstractions. However, programs are often expected to do more than recognise fixed data, and indeed may be involved with many different data structures, perhaps by transforming one into another, or by generating new ones. As a consequence, it may not be possible to identify a single structure which characterises the computing process over its progress.

(d) When attempting a hierarchical program design from the top down, the immediate problem is to determine the initial decomposition, which decides the dominant structural relationship at the "top". Sometimes the choice is difficult because too many structures are promoted by the problems, (see (c) above) but more seriously, a good decomposition may be untried because of a premature concern with realities (see (b) above).

(e) Some programming problems are not given a clear and explicit specification. This is a particular lack in many non-numeric and so-called data processing problems, which do not have access to the well established theories and notations of problems in the numerical domain. Often, the only explicit specification of a problem is to be found in the program which solves it, and there the specification may be far from clear. (What, exactly, was Fortran II?). This paper considers those programs which have complex inputs and outputs, and examines the use of syntax graphs to provide structural descriptions of such situations.

2. SYNTAX GRAPHS AND DATA STRUCTURE

We consider that data which is represented in a linear storage medium is, apart from ordering, intrinsically amorphous: data structure is that which is imposed by programs which recognise, generate or otherwise operate on the data. Data descriptions, however, usually de-emphasize these procedural aspects, and emphasize constituent structure through ordering or labelling relations. For example, the graph D below is a data description based on ordering relationships between the constituents A, B and C (which are described elsewhere, perhaps by other graphs).

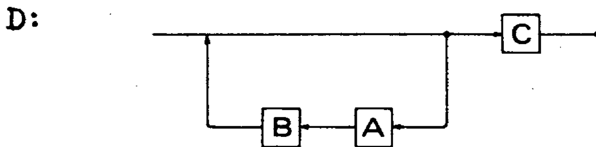


fig. 1

The graph may be given a procedural connotation, as it can be seen to display the structure of a recogniser for objects of datatype D. By adding certain semantics to the graph (and notation), it becomes a flowchart schema

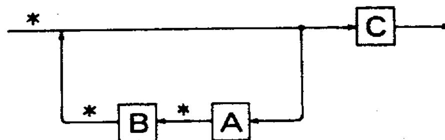


fig. 2

where * stands for an input operation, assigning a value to some "current symbol"

branch points are selectional tests on the value of "current symbol"

objects in boxes are to be tested against "current symbol"

Note that we could have considered

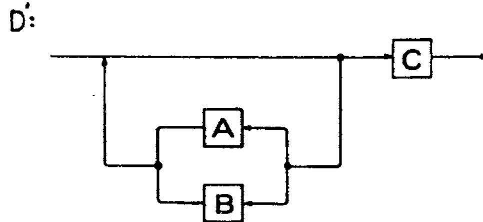


fig. 3

and designed a recogniser based on D', but such a recogniser would accept some data which D would reject since D⊃D'. (Similarly a generator of type D may be constructed.)

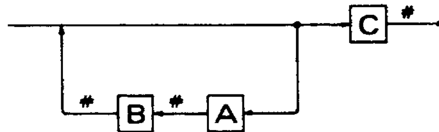


fig. 4

where # is an output operation, the above is a flowchart schema for a generator of objects of type D.

We may now consider schemas which combine recognitive and generative operations - schemas which map between data types. Simple transformations of deletion and insertion are obviously effected by the omission or insertion of output operations. (Note that more than one input source (or output destination) may be considered.) In general we may wish to attach other semantics making use of local and global variables.

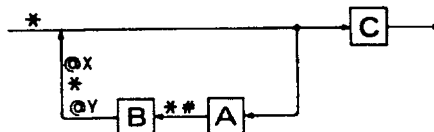


fig. 5

* : "read(f_1,sym)"

\# : "write(f_2,sym)"

@X : "<u>if</u> sym ≠ c <u>then</u> write(f_2,space)"

@Y : "<u>if</u> today = saturday <u>then</u> write(f_2,sym)"

(Using Pascal notation for operations on sequential files.)

The structure of the resulting data f_2 is that abstracted from the generative aspect of the flowchart schema

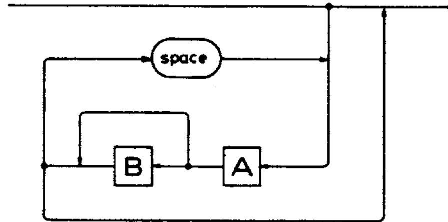


fig. 6

illustrating a transformation D $\Longrightarrow$ F_2, involving:

a deletion (of C)

an insertion (of "space")

a copy (of A)

and two structure changes wrought by the semantics of two selection operations.

We also observe that the structure F_2 is not readily described by a regular expression or any extension of BNF, and accordingly admit the analog of "<u>goto</u>" in program schemas (see Knuth and Floyd [9]).

We list the following advantages which may accrue from the use of syntax graphs to describe data and program schemas. Where program designs involve a single static data structure:

(a) a clear and explicit data description is available

(b) the structure of a recogniser is given, and error detection and recovery techniques are known

(c) the structure provides a framework for the specification of problem semantics leading to a flowchart scheme.

Further, where program designs involve (sequences of) data transformations, syntax graphs also offer

(d) a means of describing data transformations

(e) a means of specifying "milestones" (in the heuristic search sense) in the search for a problem decomposition

(f) a means for comparing programs and a guide to program transformations.

We shall illustrate these advantages in the following examples.

3. EXAMPLE PROGRAMS

3.1 Example 1

By way of introduction to the creative use of syntax graphs, we discuss the "oddword reversal problem", originally presented by Dijkstra [4], and subsequently treated by Knuth [8] and Stanton [10]. Dijkstra's input text may be described by:

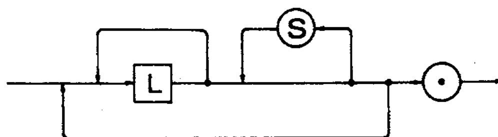


fig. 7

Much of the reasoning given in [4] p.68-71 concerns the design of the above structure; notice that the question of how "reading" is to be done (discussed on p.68) is readily decided from the graph. This structure is allowed to dominate all design aspects, and no conflict arises because reading may be co-ordinated with reversal and printing at the word level.

Note: The above structure is nondeterministic. However, it does represent a "word" as a sequence of letters, and is a regular structure, i.e. maps directly into the regular expression $(L^+S^*)^+$. Compare with:

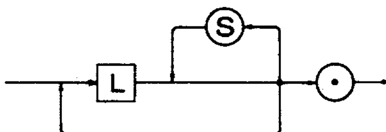


fig. 8

which is deterministic, regular but not word-structured. Or compare with:

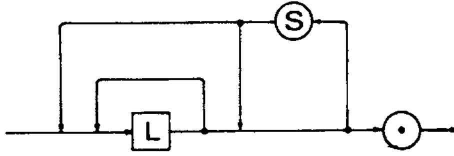


fig. 9

which is deterministic, word-structured but not regular.

Knuth [8] views the input text differently:

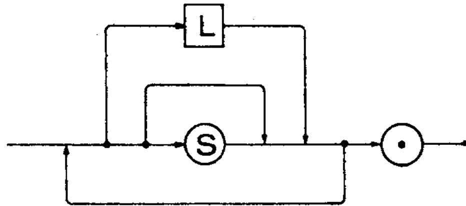


fig. 10

which is nondeterministic, not word-structured, and also changes the problem to a more general one which allows leading spaces and the null text. The required program semantics are easily attached, with some cleverness required to decide that because no state in the automata has a sensible "word" interpretation, an extra state variable is required (in fact Knuth does not introduce a special variable, but uses a wordlength counter, k, in two ways).

The input text assumed by Knuth may be given a word-structure:

INTEXT:

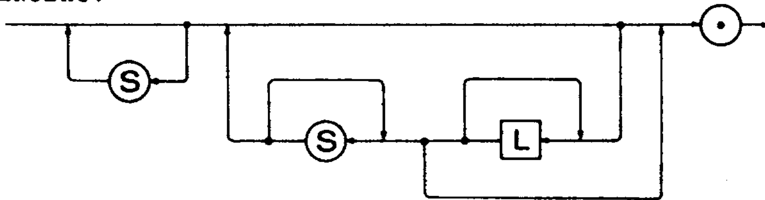


fig. 11

and for the first time, we present a description of the required output:

OUTTEXT:

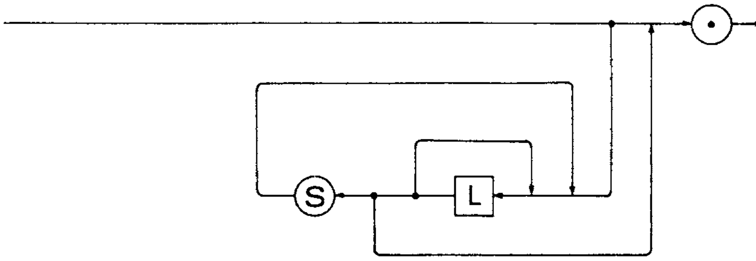


fig. 12

We may now attach semantics to a recogniser for INTEXT to create a generator for OUTTEXT (following Dijkstra or Knuth), but instead we shall examine the alternate solution of Stanton [10]. Stanton assumes the input data structure INTEXT but does not choose this as the dominant structure of his program; his design is a concatenation of three processes which perform reading, reversal transformation and then printing.

viz.

INTEXT ⟹ readitems(q1) q1 ⟹ transform(q1,q2) q2 ⟹ printitem(q2) OUTTEXT

showing a progressive transformation of the data through q1, an intermediate structure (a sequence of words, without spaces) and thence q2 (a sequence of words with oddwords reversed) and finally to OUTTEXT. At this stage, Stanton presents a direct programming language specification of the three processes. We shall give a structural description of the data transformations involved.

The program "readitems" could be designed on the recognitive structure of INTEXT, but Stanton chooses not to, and decouples his design from the specific structure INTEXT by assuming that it is possible to design a scanning procedure "readitem" which will give the input text the following structure (INTEXT'):

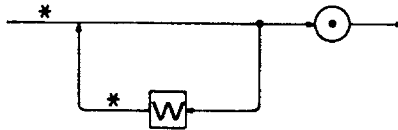


fig. 13

where W is a word.

If "*" stands for a call to "readitem", then the above may guide the design of "readitems".

Adopting a labelling convention where

* will prefix an input operation

will prefix an output operation

@ will prefix other operations

and using the same labels as Stanton (T1, T2, P1, etc.), then the complete design may be presented as:

INTEXT':

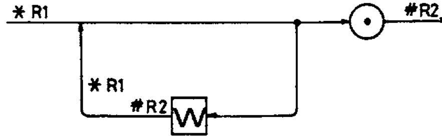


fig. 14

*R1: x:= readitem

#R2: join(q1,x)

which is the design of readitems(q1) which generates the structure:

q1:

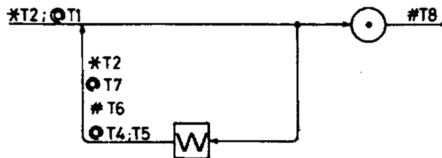


fig. 15

*T2: x:=head(q1)

#T6: join(q2,x)

#T8: join(q2,x)

@T1: i:=0

@T7: i:=i+1

@T4;T5: <u>if</u> odd (i) <u>then</u> reverse (x)

which is the design of transform (q1,q2) which generates the structure:

q2:

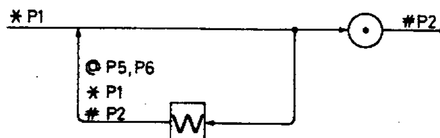


fig. 16

```
*P1:     x:=head(q2)
#P2:     printitem(x)
@P5P6:   if x≠'.' then print (' ')
```

which is the design of printitems (q2) which generates the structure:

OUTTEXT':

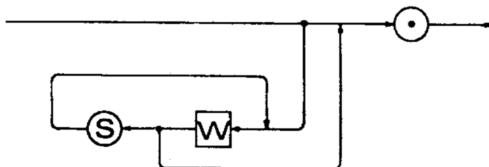


fig. 17

which is the same as OUTTEXT providing "printitem" is correctly implemented. Note that "printitem" and "reverse" need to know the representation given to "words" by the scanner "readitem".

What does the above design tell us about the virtues of (clear and explicit) data description? Not much. This is because the most complex structural transformation INTEXT ⟹ INTEXT' is hidden in the design of the scanner "readitem" - we shall explore the nature of specialised I/O procedures later. Secondly, the intermediate data structures are very simple (and all the same), and adequately defined by the notion <u>type</u> queue = <u>file</u> <u>of</u> word. Thus in this example, Stanton's direct code for the three processes is as clear as the annotated syntax graphs given. On the other hand, when it comes to attempting the program transformations which integrate the separate programs into a single sequential program, the syntax diagram presentations do have advantage because:

(a) They are diagrammatic and have the virtues found by Stanton in his flowcharts.

(b) If we know that it is possible to synchronise pairs of read/write operations between processes (in this case at the word level), and use this knowledge in an attempt to perform code integration of the kind done by Stanton, then the structure diagrams give an indication of the difficulties. In this case there are no structural impediments to integration.

<u>Note</u>: In the design above, considerations of structure lead to the design of a set of "semantic fragments" (such as "x:=readitem" and "i:=i+1"). It is interesting to speculate upon the utility of such a set in a structural transformation of a program under the condition of semantic invariance.

3.2 Example 2 : The Telegram Problem

The following programming problem has been presented as structurally interesting by Henderson and Snowden [5] and Jackson [7]. A stream of telegrams is available as a segmented file of words and spaces. The segments are of fixed length due to a physical access buffer. Words comprise letters and digits, terminated by either a space or an end-of-segment mark (eos).

INFILE:

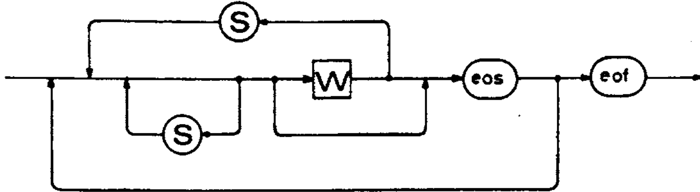


fig. 18

note that the graph allows the null segment. We could specify a segment of at least a word or a space at the price of extra syntactic complexity. In any case, we cannot ensure syntactically that the segment is the correct size, and so a counter is required.

The data has a second organisation, that of telegrams

TELEGRAMS:

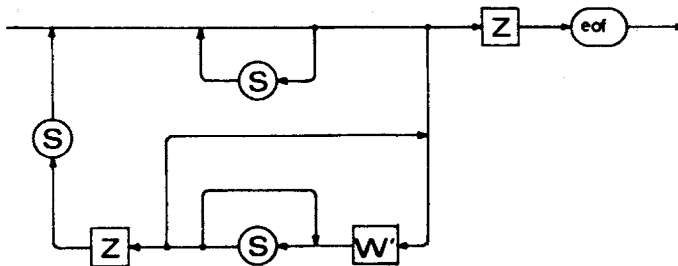


fig. 19

Z is the telegram terminator, being the special word "ZZZZ'
W' are all words except "ZZZZ"

The null telegram, comprising zero or more spaces and the telegram terminator "ZZZZ", terminates the telegram file.

TELEGRAM:

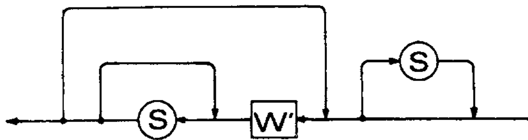


fig. 20

The required output is a "neat listing" of telegrams, each accompanied by a word count and an oversize word occurrence message.

A neat listing will have the general organisation of the input file; we first describe a roughlisting

ROUGHLISTING:

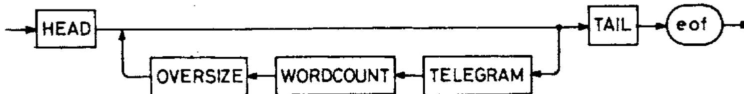


fig. 21

HEAD:	report heading
TAIL:	report tail
TELEGRAM:	given previously

WORDCOUNT:

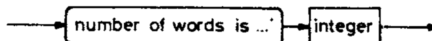


fig. 22

OVERSIZE:

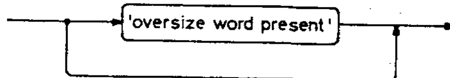
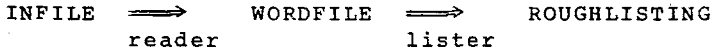


fig. 23

The difficult aspect of the problem is that the segmentation of the INFILE is not related to a TELEGRAM; the only correspondence between INFILE and TELEGRAMS is that each W in INFILE corresponds to either a W' or a Z in TELEGRAMS. (Also, an interword "eos" in infile is to become a space in the telegram roughlisting.) Consequently, the dominant program organisation is:



giving the following design:

"reader" is a program based on a recogniser for

INFILE:

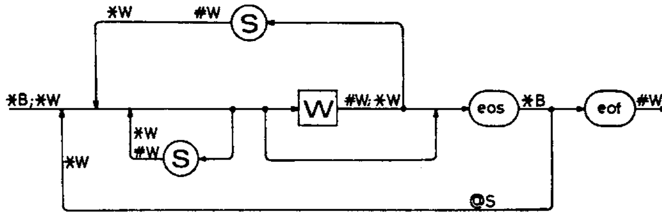


fig. 24

*B : read segment and fill seg buffer.

*W : read word {W or S or eos} from buffer.

#W : write word to wordfile {W, S, eof}.

@S : write space to wordfile.

We maintain the integrity of spaces and treat "eos" after Henderson and Snowden, to generate

WORDFILE:

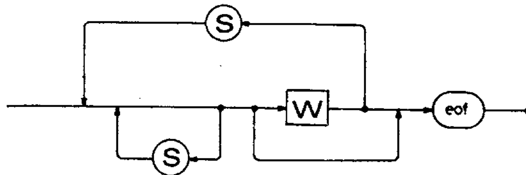


fig. 25

note that the description admits a file without words, yet we know that at least the terminator "ZZZZ" for the null telegram must be present. In fact, we know that WORDFILE also has the structure TELEGRAMS given earlier, where

$$\text{TELEGRAMS} \subset \text{WORDFILE}$$

Thus we adopt the structure of TELEGRAMS as the recognitive structure of

LISTER:

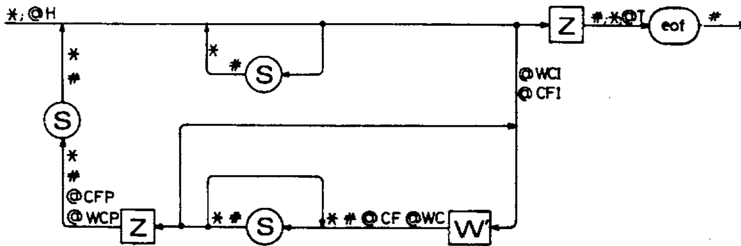


fig. 26

*	:	read word {W', S, Z or eof} from wordfile
@H	:	write "head", the report header
#	:	write word {W', S, Z or eof} to roughlisting
@T	:	write "tail"
@WCJ	:	initialise word counter : integer WC:=0
@WC	:	increment word counter : WC:=WC+1
@WCP	:	write "wordcount"
@CFI	:	initialise oversize check : boolean CF:=false
@CF	:	set oversize check : CF:=true
@CFP	:	oversize : if oversize then write "oversize"

To complete this first design phase, "buffer" and "word" must be represented, and the instructions *B and #W designed. We shall ignore the data transformation roughlisting ⟹ neatlisting (see next example).

The opportunity for program improvement arises, as before, by elimination of the intermediate file and multi-pass processing. Because we can show a 1:1 correspondence between "writeword" instructions in "reader" {i.e. #W and @S} and "read word" instructions in "lister" {i.e. *}, the two programs may be synchronised in some way:

(a) If coroutine control is available, then directly as a pair of co-operating coroutines, communicating at the word level.

(b) If such control mechanisms are not available, they may be simulated (e.g. Jackson simulates an "own" state vector including a "local instruction counter" in Cobol).

(c) Attempt a program merge transformation after Stanton [10].

The last-mentioned transformation is not as simple as in example 1, because it is hard to find a structural basis for

the 1:1 correspondence; in example 1, there was a common structure to all programs being merged (with only minor rearrangement required, of the kind $S^+ \equiv SS^*$). However, in line with the footnote to example 1, we know from the first design phase what the semantic fragments are, so we might try some other structure in place of TELEGRAMS. Let us consider WORDFILE which is structurally compatible with INFILE or even more directly, INFILE itself. We now have the same difficulty in a different form, in that the semantics which were derived for the program "lister" assume a TELEGRAMS structure, but INFILE is not organised by telegrams at all, (again, the structure clash). Specifically, there are no states in (a recogniser based on) INFILE corresponding to "end of (non-null) telegram" and "end of null telegram".

Now, additional states may be introduced in two ways:

(a) by staying within the "scan and test next symbol" recogniser model, introducing new decision nodes and paths.

e.g. replace

fig. 27

(b) by introducing auxiliary variables (flags) outside of the "scan and test" framework, as we have done already with counter variables.

Thus we may specify the recognitive structure:

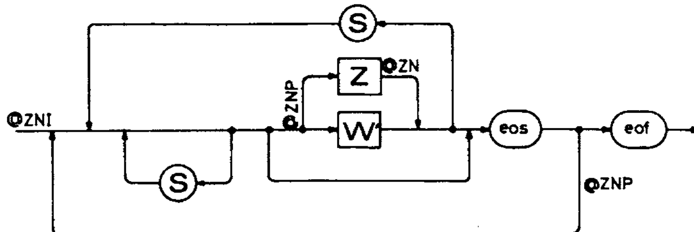


fig. 28

where @ZNI : boolean ZN:=false {state variable for "null telegram just read"}

@ZN : if WC=0 then ZN:=true
else begin ZN:=false; @WCP; @WCI; @CFP; @CFI end

```
@ZNP  :  if ZN=true then     write("null telegram
                                   termination error")
```

The remaining semantic fragments {*B,*W,@H,#,@T and @WCI, @WC,@CFI,@CF,} are attached in a straightforward way giving a single sequential program design.

Another design might attempt to use the structure of telegram as the dominant structure (directly or transformationally) as did Henderson and Snowden. This is difficult because provision for "eos" has to be made and recovered from in so many places. If the integrity of interword spaces can be abandoned (by changing the problem), we may proceed as for example 1, based on a telegrams structure and a "next word" scanner.

3.3 Example 3 : A Text Editor

An example to illustrate "milestones", more complex program transformations and the design of I/O procedures. The problem is to edit some input text

INTEXT:

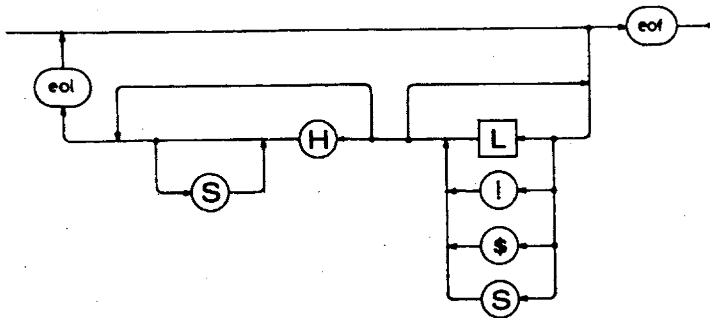


fig. 29

which is a sequence of lines and spaces (S), letters (L) which group into words, and edit marks (H, ! and $) having the following meaning:

$: Delete all characters from (and including) the "$" back to the start of the line.

! : Delete all characters from (and including) the "!" back to (and including) the first letter of the word preceeding the "!". Where there is no preceeding word, treat "!" as "$".

H : A hyphen, to indicate that the last word of that line is incomplete, and that its spelling is continued in the next word of text if any (what if none?).

Deletions take precedence over hyphenation.

The required output is

OUTTEXT:

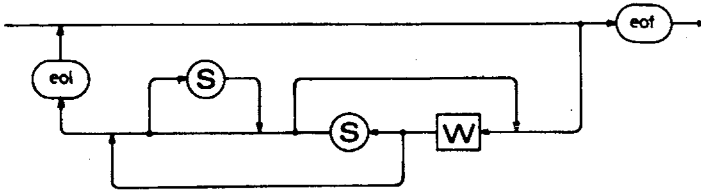


fig. 30

which is a sequence of (fixed length) lines of edited text, the only spaces being word separators and end-of-line padding. No hyphenation permitted.

If we decide upon a sequence of data transformations as the structure of the initial program, we should aim to distribute the problem semantics clearly over the intermediate programs, and ensure that the structure change is gradual over the sequence. This may be a problem in its own right - a search problem to find a suitable sequence in the space of all possible. Working backwards is obviously as useful as working forwards, and so too is the identification of crucial intermediate structures (milestones).

Such is the case when we guess that a useful intermediate structure is one where the intra-line editing has been done (deletions performed and redundant spaces removed), but where the line organisation and hyphens are still intact. Word structure must exist at this stage for the above editing to have been done.

LINEH: {lines with hyphenation}

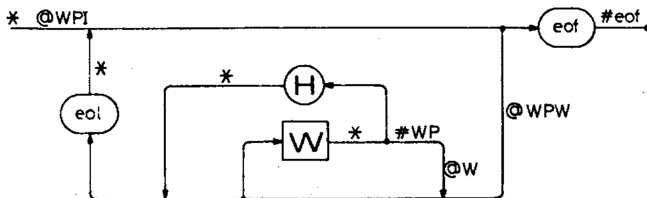


fig. 31

*	:	read symbol {W, H, eol or eof}
@WPI	:	initialise prefix word WP to empty
@WPW	:	<u>if</u> current symbol is a word <u>then</u> concatenate into WP
@W	:	<u>if</u> current symbol is a word <u>then</u> overwrite onto WP

#WP : write WP to output.

LINEH is a milestone. The point to note is that the structure of LINEH directly suggests the elimination of hyphenated lines, and indirectly, forces us to decide precisely on the semantics of hyphenation, which were not completely specified. The generated structure is

PURETEXT:

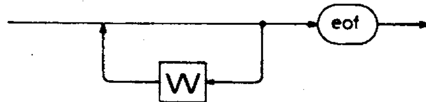


fig. 32

We now work backwards to design the predecessor structure to LINEH. We hope to specify "$" and "!" edit operations in this transition, and to optimise our chances for a clear and simple specification we decide to allow the predecessor structure the minimum complication necessary - viz. words and edit marks only, no spaces and line organisation.

LINEHED: {lines with hyphen, excl and dollar}

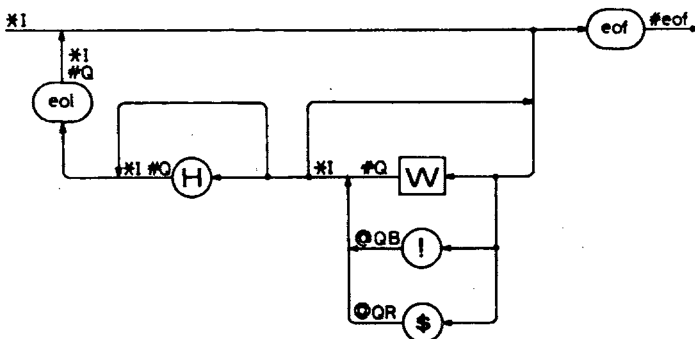


fig. 33

*I : read next symbol {W, H, !, $, eol or eof}

#Q : write word to sequence Q

@QR : remove line from sequence Q

@QB : if current line of Q not empty, then remove the last word (of line) from Q (back from of Hoare [6]).

This design step is vindicated if we can perform the data transformation LINEHED ⟹ LINEH without any further mediating data structures, viz. the "Q" of LINEHED and the "line" of LINEH must be the same at some level of representation where the operations Q#, @QR, @QB of LINEHED and * of LINEH all apply. See section 5 on representation.

The remaining data transformations are obvious, giving the full program sequence as

$$\text{INTEXT} \underset{T_0}{\Longrightarrow} \text{LINEHED} \underset{T_1}{\Longrightarrow} \text{LINEH} \underset{T_2}{\Longrightarrow} \text{PURETEXT} \underset{T_3}{\Longrightarrow} \text{OUTTEXT}$$

We now attempt some program transformations towards a design more suited to a single sequential machine.

(a) The step T_0 may be replaced by redefining *I in LINEHED to be a call to an input procedure (as in example 1).

(b) T_3 (which has not yet been designed) may be replaced by a simple structure (sequence of words) using an output procedure "writef(W)", a write procedure which attends to format. In fact, we now see PURETEXT merely as a crutch to understanding in our initial design, for "writef( )" may be employed direction in place of #WP in LINEH, thereby elimination T_3.

(c) We have now eliminated all data transformation except T_1 and T_2, which we attempt to merge. We now do this by synchronising the editing (T_1) and de-hyphenation (T_2) at the level of "line" by establishing a communications area of the type Q, and the hyphen prefix store WP inherited from the program T_2.

The result of the merge of T_1 and T_2 is

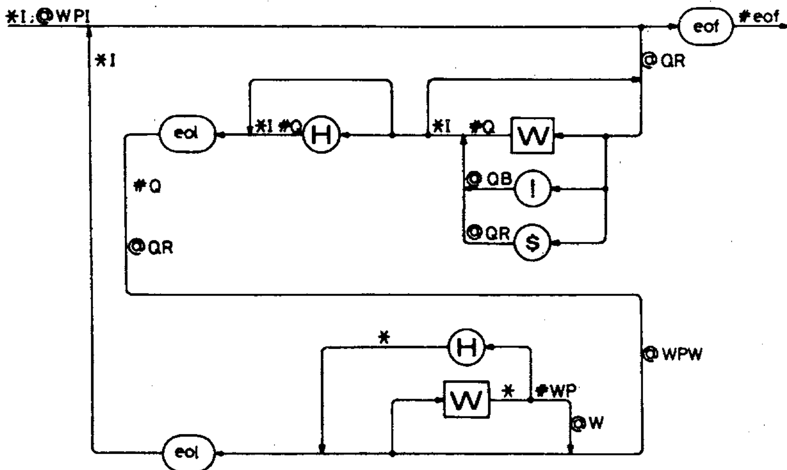


fig. 34

The minor changes to instruction semantics as a result of the various merges are:

@QR between what was T_1 and T_2 above must be a nondestructive reset of Q

*I   is now a call to an input procedure

#WP uses the output procedure "writef"

*   is now "read from Q"

4. INPUT AND OUTPUT PROCEDURES

Input and output mechanisms as introduced in the examples are best represented as coroutines (Conway [1], Dahl and Hoare [3]) but if this is not possible then we may attempt to represent them as procedures.

Input procedures which perform minor editing, symbol construction and classification are well known as "scanners". Such procedures may be directly designed from an input data structure specification (e.g. INTEXT, INFILE and INTEXT of examples 1, 2 and 3 respectively). In all but the simplest cases, case switching on an "own" variable is required to simulate the reactivation of the underlying process at the correct resumption point.

Output procedures are more difficult to design in that there is not such a close relationship to the required output structure. The relationship can be made closer if we allow additional buffering local to the output procedure (also an "own" variable).

The main cause of the difficulty is that the calling/called relationship with output procedures is the reverse of that with input procedures, with respect to the direction of data transformation.

5. DATA REPRESENTATION

The syntax graph notation is familiar enough to see the relation between iterative structures in a graph and sequence data types (a relation which was exploited in the previous examples). If we allow labelled graphs and multi-level graphical descriptions, then other graph-datatype relations may be established, e.g.

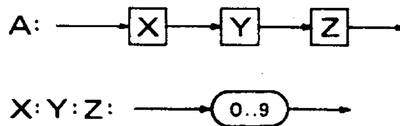


fig. 35

where A is a concatenation of objects X, Y and Z, corresponding to a cartesian product data type, expressed in Pascal, say:

A : <u>record</u> X, Y, Z : 0..9 <u>end</u>

Similarly, a set of alternates in a graph may be considered as a union data type. (The subrange 0..9 is a union of

integers, 0..9, and could have been graphed as a set of alternate branches.)

We note that structural descriptions based on ordering relations between constituents lends a bias towards ordered representation in linear memory, and the use of "next symbol" operations. On the other hand, labelled descriptions often anticipate selective reference and update operations. The STRUCT mode in EL1 [11] provides an interesting combination of both aspects, since constituent modes of a STRUCT (cartesian product) may be selected either by label or by sequential index.

Syntax graphs are separated from representational considerations in another way: in example 3, the data structure Q was introduced at an abstract level, as were the operations on Q (viz, #Q, @QR, @QB). Such data abstractions are valuable, as noted by Hoare [6, p.95]: "The use of abstractions in data structuring may help to postpone some of the decisions on data representation until more is known about the behaviour of the program and characteristics of the data...". Conveniently, the concept "Q" (being data + operations) is left open to a variety of possible refinements and decisions about representation. The standard repertoire of programming devices and language features suggest a range of ideas for the further refinement of "Q": sequential file, queue, deque, pushdown list, indexed stack, array, linked lists, etc.

Example 3 was programmed in Pascal for a compiler with a compile time macro facility, and the definition of Q and its operations were abstracted from the program text and collected together in a macrodefinition section of text. This proved to be convenient when trying various ideas for "Q", but more importantly, maintained a closer relationship between the flowchart schema and the program text and collected together in a macrodefinition section of text.

6. CONCLUSIONS

Syntax diagrams have been used to support the design of programs which perform input-output data transformations on sequence-type data. The graphs are useful because they provide a non-procedural data description as well as a basis for flowchart schemas (in particular, schemas for recognisers and generators which are essential to any data transformer).

An unusual characteristic of program design based on the design of a recogniser is that most of the purposive problem semantics are designed after the control structure and principal datatypes, and are added as macro-like insertions.

Finally, we may see in the examples, the clarity which can be given to problems which have had a not-so-clear flavour in the past: in particular we may draw attention again to "structure clashes", "milestones" and program "merge" transformations.

ACKNOWLEDGEMENT

I wish to acknowledge helpful discussions with Barry Dwyer and Chris Marlin, and Young Choi, who translated many schemas into working programs.

REFERENCES

[1] M.E. Conway, Design of a Separable Transition-Diagram Compiler, Comm. ACM 6, No.7 (July 1963).

[2] O-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Structured Programming, Academic Press, 1972.

[3] O-J. Dahl and C.A.R. Hoare, Hierarchical Program Structures (in reference 2).

[4] E.W. Dijkstra, Notes on Structured Programming (in reference 2).

[5] P. Henderson and R. Snowden, An Experiment in Structured Programming. BIT 12, 1972, pp.38-53.

[6] C.A.R. Hoare, Notes on Data Structuring (in reference 2).

[7] M.A. Jackson, Principles of Program Design (APIC Studies in Data Processing No.12), Academic Press, 1975.

[8] D.E. Knuth, A Review of "Structured Programming", Stanford Computer Science Department report STAN-CS-73-371, 1973.

[9] D.E. Knuth and R.W. Floyd, Notes on avoiding "goto" Statements, Inf. Processing Letters 1, No.1, 1971.

[10] R.B. Stanton, Complexity in Program Structure in "The Complexity of Computational Problem Solving", R.S. Anderson and R.P. Brent (eds.) Queensland University Press, 1976.

[11] B. Wegbreit, The treatment of Data Types in EL1, Comm. ACM 17, No.5 (May 1974).

[12] N. Wirth, Program Development by Stepwise Refinement, Comm. ACM 14, No.4 (March 1971).

[13] N. Wirth, On the Composition of Well-Structured Programs, Computing Surveys 6, No.4, 1974.

[14] N. Wirth, Algorithm + Data Structure = Program, Prentice-Hall, 1976.

5

Lessons from Automatic Preprocessing of Symbolic Computing Problems

John A. Campbell

1. INTRODUCTION

The history of symbolic computing (SC) is different in at least two senses from the history of numerical computation. As most of the work in the field loosely described as "programming language systems" is either stimulated or menaced (according to one's point of view) by the idea that there is a user somewhere who will ultimately be relying on one's results in order to compute numbers more efficiently, problems arising from the immediate needs of SC users have received little attention. Nevertheless, these problems are worth examination in their own right.

The first of the significant peculiarities of SC which distinguish it from numerical computing is that it is not clear in advance what is the best representation of data for the quantities in the statement of a calculation. Because SC makes extensive use of list processing, it is very demanding on primary storage. Many potential users of SC systems have given up their computations because their incautious experiments with SC have quickly filled all available storage and produced no results. This is normally only an indication that the representation of data is not the best, and is using the storage inefficiently. For example, the expression

$$\frac{5}{2h^{7/2}} \left(\frac{dh}{dz} \right)^2 \frac{d^2h}{dz^2} \tag{1}$$

translates into LISP prefix notation as

```
(TIMES(QUOTIENT 5 2)(EXPT H(QUOTIENT -7 2))(EXPT (DF H Z 1)
2)(DF H Z 2))
```
(2)

and requires at least 25 words of storage, while (2) may contain many redundancies in an actual computation. (This example will be discussed further in Section 3). A system of SC programs, e.g. REDUCE, [4] may have a more compact canonical form for (2), but this form has been designed for relative efficiency over a large number of different computations, and has no means of taking advantage of special novelties in individaul computations. For (1), if it is additionally known that h is the only symbolic function and z is the only variable to occur in a differential operator, SC languages (e.g. LISP) and systems (e.g. REDUCE) can make no automatic use of that information, but it is evident that a representation which replaces (2) by

((5 . 2)(-7 . 2)(1 . 2)(2 . 1) (3)

and occupies only 8 words in LISP, then preserves all the essentials of (2) and is a "better" representation for SC. Obviously, an SC sub-system which can make transformations of data such as that from (2) to (3), and transform existing programs or subroutines for standard operations on (2) accordingly, must have considerable value in this field. Section 3 below examines the systematic preprocessing of data to achieve a compact representation in storage.

The second peculiar property of SC is that the statements of many "classical" computations are acceptable programs in themselves, e.g.

P(0) = 1;   P(1) = X; (4)

FOR N ε [2,10] WRITE P(N)=((2*N-1)*X*P(N-1)-(N-1)*P(N-2))/N.

for the Legendre polynomials in x. Of course, (4) is now almost respectable for numerical computing too, given a value of x, but the understanding of good numerical programming usage has taken a long time to evolve, while SC language and system designers have been spared most of the consequent distractions. Moreover, a much larger fraction of SC than numerical computations may be expressed in a simple form like (4).

This simplicity has probably led to premature satisfaction with the range of facilities available in languages for the control of symbolic computations, since the consideration of one important kind of manipulation which is just as legitimate as (4) has been largely excluded. SC systems at present accept only programs, while the user with a problem may find that the "program" stage is separated by a large amount of tedious equation-solving or formula-manipulation in general from the stage which he finds it most convenient to reach by hand. The largest SC systems, e.g. MACSYMA [3] and SCRATCHPAD [5], contain sections which a user may instruct to perform basic steps of formula-manipulation, but these systems are too large to be available in the average computing centre. Further, the sequence of steps needed to obtain a satisfactory program may not be obvious at the outset, so that even a large conventional system is not always appropriate. Ideally, what is needed is a special-purpose system which is fairly compact and which can fill in as many steps of formula-manipulation as possible.

The two aims in the design of a special system contradict one another unless a new approach is tried. The necessary approach seems to be "automatic deduction of information", and may be dignified by the title of "automatic programming" when a systematic basis for this type of construction has been shown to work on a good range of SC problems. At present, the basis is not systematic. It is described in Section 4 below, partly as a record of work in progress, and partly because it makes use of several ideas whose usefulness has already been established in the treatment of compact representations of data (Section 3). One additional

hope for the future is that closer studies of the framework necessary to support this variety of automatic programming will indicate a need for extensions to SC languages (references 3-5 in effect describe both systems and languages for SC) having much richer semantic as well as syntactic content than those that are used currently.

Section 2 outlines the keys or properties of symbolic expressions and data which are examined during both types of automatic preprocessing. The special-purpose system of programs to carry out the preprocessing is written in LISP.

2. PROPERTIES USEFUL IN AUTOMATIC ANALYSIS

2.1 General

Each of the two types of analysis may be regarded as a minimization. A search for a representation of data which overcomes as far as possible the profligate behaviour of SC in primary storage is directed towards the minimization of the number of pointers in the basic expressions which standard SC must eventually handle. A program for formula-manipulation in restricted storage cannot afford extensive backtracking or much of the traditional apparatus of automatic theorem-proving; it must select and act on a minimum amount of relevant information from the expressions to be manipulated. The headings in the rest of this section summarize the keys which have proved to be the most useful in automatic preprocessing so far.

Initially, all expressions are translated into LISP prefix notation by a modified [1] version of Weissman's [3] infix-to-prefix translator. The structure of a given computation is then examined by LISP programs which detect relevant information and redundancies in the resulting uneconomical list structure.

2.2 Low-order information

In effect, almost all symbolic programming is devoted to the computation of families of results for given problems. Typical members of families are quantities defined by a recurrence relation, or the various orders of partial results in some perturbation theory. In practice, the general statement of a calculation comes complete with specimens of answers from the lowest order up to the highest order which it is convenient for the inventor of the calculation to find by hand. These relatively low-order specimens are often of considerable use in the determination of the general properties of any subsequent symbolic computations. Therefore, provision is made for the declaration of any number of low-order expressions, in the style of (4), which are then used as data for the stage of automatic preprocessing. Some of these expressions can, of course, be recalculated from the bare minimum of particular data, e.g., the starting values needed to accompany a recurrence relation, but to do so, while the problem is still cast in LISP prefix notation, as in (2), is wasteful in space. It should be no hardship for the user to give as much low-order information as he possesses, to increase the efficiency of the task of auto-

matic deduction of further properties.

Examples of the uses of low-order information are presented in Sections 2.4 and 2.5.

It should be remarked that deductions made from a finite number of specimens are not proofs, e.g., if the first 3 members of a family have 1, 4 and 9 terms respectively, the simple deduction that the next member has 16 terms may be contradicted if it is revealed that the length of the nth member of the family is actually $2^n + n-2$. But an automatic preprocessor is not in the business of finding general properties unaided. Each tentative deduction is presented for the user's inspection and possible correction before it is taken up as an item of data for further deductions. Interactive computing is necessary to make the interchange of information between user and preprocessor fully effective.

2.3 Weights

Among the simple keys which are helpful in automatic preprocessing, the weight or asymptotic weight or dimension of a symbolic quantity is significant both for deductions and for the choice of a good representation of data. Provision is therefore made for the declaration of a weight for any such quantity; e.g.,

```
ASYMP W = 3, Y = 5, EPS(N) = N+2                    (5)
```

effectively says that W, Y and ε_n are proportional to x^3, x^5 and x^{n+2} for some x. The weight of an arbitrary expression can then be calculated.

To assist in a choice of representation of data, it is evidently useful to know, for example, that p_n is a polynomial of degree n in x, or that terms of degree greater than n in x should be removed in truncation of Z_n. The simple extension

```
ASYMP 0<P(N)<N,   Z(N)<N                            (6)
```

of (5) allows this information to be declared. (5) and (6) in their present forms express the assumption that weights refer only to one implicit quantity, but this restriction may be removed by syntactic changes in the future.

In the problem of the computation of functions $Z_n(b)$, where $Z_1(b) = t(b)$, and

$$Z_n(b) = t(b) + \sum_{k=2}^{n} k^{-1} Z_{n-1}(kb) + \sum_{j=2}^{n} (j!)^{-1} \left(\sum_{k=1}^{n} k^{-1} Z_{n-1}(kb)\right)^j, \tag{7}$$

the requirement that all terms of the form

$$\prod_{i=1}^{h} t(k_i b), \quad n < \sum_{i=1}^{h} k_i, \quad h > 0$$

are to be dropped from Z_n can be expressed in part through (6). Taken together with other information derived from preprocessing of (7), this reveals that Z_n is a sum of $\sum_{i=1}^{n} \pi(i)$ terms, where $\pi(i)$ is the number of partitions of i, and therefore leads to a most compact representation which needs no pointers to link successive terms of Z_n.

An even simpler problem which has been solved is the computation of repeated convolutions of the uniform distribution $p_1(x)$ which is 1 for $0 \leqslant x \leqslant 1$ and zero elsewhere. Since

$$p_n(x) = \int_{-\infty}^{\infty} p_{n-1}(x-y)p_1(y)dy, \tag{8}$$

the first declaration in (6) is sufficient by itself to set up a pointer-free (n+1)-term representation for use with p_n.

The most important use of weights (in the sense that they provide important information which is hard to find in any other way) is in the automatic deduction of expressions for insertion into programs. In these cases, the weights of the results to be calculated are often known in advance. Therefore a line of deduction can be abandoned as soon as it becomes evident that the result will have the wrong weight. Also, the correct weight can be employed as a guide to the best deductive step to use in a situation where there are no other obvious clues which suggest what to do next. An example of this process is given in Section 4.

2.4 Lengths of Expressions

It is often said that SC is expensive in space because the size of expressions that are computed grows exponentially. Crudely, in a perturbative calculation, this may be so if the $(n+1)^{th}$ member of a family is built up by repetition of the same basic computing step on each term of the nth member, with few or no cancellations or simplifications. If pure exponential growth occurs, it is unlikely that any automatic or manual treatment will lead to significant improvements in representations of data. However, there are enough special and better cases to make an examination of the rate of growth of low-order results attractive.

In the special cases that have occurred in the SC problems which I have received over the last 18 months, each type of growth can be associated with a particular compact representation of data. The preprocessor is therefore set up to match the lengths of the first few low-order results in a problem to general expressions (e.g. polynomial, exponential), to estimate the type of growth. If the match is achieved, the corresponding representation is selected as a basis for storing the data.

Two cases are worth considering as examples. Firstly, for linear growth in the size of expressions (as in (8)), the observation that each of the an + b terms of the nth member of a family contains t distinct pieces of non-redundant information leads simply to an array with (an+b)t entries as the medium of storage. For the polynomials in x in (8), for example, a = 1, b = 0, t = 2 (the numerator and denominator of the numerical coefficient of a term), and the power of x in a term is redundant because this can be recovered from the eventual ordering of the expression for p_n in storage. Detection of non-redundant information is discussed in Section 2.5.

The discussion of (8) is simplified above; in fact, it is easiest to write p_n as a collection of polynomials $p_{nm}(x)$, where $0 \leq m \leq n$ and $p_{nm} = p_n$ when $m - 1 \leq x \leq m$. Therefore the problem admits quadratic growth, and a complete specification of p_n requires space for n(n-1) + 1 terms. The preprocessor can determine this in the general "polynomial" branch which considers linear and quadratic growth as particular cases.

The most interesting special example identified to date is the "partition" example, where the number of terms in order n depends on the number of partitions of some suitably simple function of n. (7) is a rather severe "partition" example, but a simpler problem of this type is the subject of Section 3. Its essentials are already displayed in (1). Even though there may be any number of derivative factors and powers in a term analogous to (1), it is possible to encode the entire derivative part of the term, given the order of the expression to which the term belongs, by a single number N without loss of information. Further, given the order and N, it is possible to reconstruct the entire derivative part. N enforces a canonical ordering of terms, i.e., the position of the representation of a term in an array of terms implies N, so that the derivative structure itself need take up no space in storage.

The eventual aim in making use of the key of length for low-order expressions in SC is to build a collection of categories of growth, to each of which a standard compact representation and algorithms for going backwards and forwards between it and a literal representation such as (2) are attached. In several cases, of which the "partition" case is the most impressive, the key of length alone allows the choice of a most compact representation of data.

2.5 Frequency of Access

As a matter of course, the preprocessor uses the formula or algorithm given to it for the generation of the nth member of a family of expressions in a problem to find at least one new member of the family, in the prefix notation of (2). The point of this exercise is to record what happens to the lower-order member(s) which are input to the algorithm. A LISP pattern which is an abstraction of the form of particular expressions like (2) is produced (e.g., through replacement of numbers by dummy LISP atomic symbols), and a reference-count property is introduced for each atomic symbol in the pattern.

After the computation of a new member of the family of results, the counts are examined. The examination discloses which parts of the data are redundant, and what are the relative frequencies of access of the remaining parts. Redundant data are not provided for in the compact representation which is then set up. Where items of non-redundant data require LISP lists rather than array entries of fixed size, the assigned positions in an appropriate list of items are ordered in decreasing frequency of access, with the most frequently accessed object at the head of the list.

2.6 Further Properties.

The keys discussed above have already shown their work in the preprocessing of SC problems. Work is still in progress on other keys which have had less general use so far, but which have been important in individual problems. Most of them require some preliminary declarations about a problem as input to the preprocessor. For example, because symbolic denominators cause more difficulties with storage than any other component of symbolic expressions, any declaration that particular forms of denominator which may occur as by-products of a computation can be renamed and transferred to numerators (e.g., "for all n let 1/(an+b) = d(n)") should lead to a substantial improvement in the representation of data. Another example: given that a user may know simplification rules which apply to his problem, it is probable that the rules themselves contain information on how to optimise the representation of data for the problem. Moreover, he may be trying to write a program which consists of the application of the rules in some particular sequence, and a preprocessor may be asked to find that sequence. Therefore the question of the most effective methods of extraction of information from such rules in still under study. One particular case is mentioned in passing in Section 4.

3. COMPACT REPRESENTATION OF DATA FOR A FORMAL-DERIVATIVE PROBLEM

The problem is to compute f_n, given that

$$f_n = \frac{1}{h^{\frac{1}{2}}} \frac{d}{dz} f_{n-1}$$

and that

$$f_o = \frac{5}{4h^3} \left(\frac{dh}{dz} \right)^2 - \frac{1}{h^2} \frac{d^2h}{dz^2} \qquad (9)$$

Elements of this problem have been presented above. A possible general term (1) is displayed in LISP prefix notation in (2), and in a somewhat more economical list representation in (3). The transition between (2) and (3) follows simply from the preprocessing step described in Section 2.5.

Although (2) is a list of dotted pairs of numbers, not all parts of the list have the same significance. The "abstract pattern" mentioned in Section 2.5 shows here that

the first two dotted pairs, which carry the numerical coefficients and powers of h in the terms of (9), always have the same structure and meaning, while the remainder of the list is an expression of variable length containing derivative information. The contents of the first two pairs are therefore marked as belonging together in a pointer-less block of our items, while the available specimens of derivative expressions are scanned for keys based on lengths or weights.

If the weight of $d^m h/dz^m$ is declared to be m, the preprocessor can find that the overall weight of any term of f_n is n + 2. The composition of a constant weight W from variable-length expressions is a standard clue suggesting that the number of such expressions is a function $\Pi(W)$. By this route, therefore, the generation of the hypothesis that f_n has $\Pi(n+2)$ terms is immediate. However, in the more likely event that the user makes no declaration of weights for (9), the same result is suggested by the measurement of the lengths of $f_o, f_1, f_2, \ldots$. Neither of these methods of obtaining a result is a proof; if used interactively, the preprocessor is designed to report the conclusion as tentative and to go ahead only after approval of the report.

As mentioned in Section 2.4, expressions which grow with n like the number of partitions of n have a simple pointer-free representation [6], whose description is now included in the preprocessor. The consequence for (9) is that the compact representation of f_n in storage is found to be an array with $4\Pi(n+2)$ elements, each term occupying four locations. The entire derivative structure of a term is implicit in its position in the array. Reference 6 describes the representation in detail.

Originally the preprocessor had no special method for handling partition-like growth. However, by calling attention to the fact (in this and one similar problem) that the "partition" aspect of the problems was the only one from which pointers and lists could not be eliminated, it stimulated the work which led to the algorithm in [6].
In future, one valuable aspect of preprocessing may thus be to isolate the sub-problems in transformation of data which need further human attention.

4. AN EXAMPLE OF A TRANSFORMATION OF A PROBLEM

In cases where the most convenient point for a manual calculation to stop is still well short of what is needed for acceptable input to a conventional SC system, the prospect of using some form of SC to bridge the gap is attractive. The abilities of a preprocessor (and its limitations in design) are well illustrated by one realistic problem [7].

It is required to find functions A_{2n} and B_{2n-2}, given the function Z_{2n}, which depends on a set of symbolic quantities

$$\varepsilon_o, \varepsilon_1, \ldots, \varepsilon_{2n-4}, \varepsilon_{2n-2} \; . \tag{10}$$

The functions are related through the integral identity

$$\oint \frac{\partial F}{\partial E} (Z_{2n}-A_{2n}) + \left(\frac{\partial Z_{2n}}{\partial E} - \frac{1}{2}\frac{\partial \varepsilon_o}{\partial E} B_{2n-2}\right) d\xi = 0 \tag{11}$$

where integration is round a closed contour in the complex ξ-plane. The operator identity

$$\frac{\partial}{\partial E} D = D \frac{\partial}{\partial E} - \frac{\partial F}{\partial E} D \tag{12}$$

is also given, with $D = d/d\xi$. It is known that $\frac{\partial F}{\partial E}$ and $\frac{\partial \varepsilon_m}{\partial E}$ for any m are unsimplifiable (13), that

$$D\varepsilon_m = \varepsilon_{n+1} \quad , \tag{14}$$

and that ε_m has weight m + 2 (15). Thus Z_{2n} has weight 2n.

Clearly A_{2n} begins with Z_{2n}, and gains extra terms through repeated integration by parts in (11), with the help of (12). A general-purpose SC system could be instructed to compute in this way, but a system general enough to deal with a wide range of problems would be too large for most computers. For a small preprocessor, one can only ask that equations for A_{2n} and B_{2n-2} be found which reproduce the given low-order functions for particular values of n (e.g. $Z_4 = A_4 = -\varepsilon_o^2/8, B_2 = -\varepsilon_o/2$.

The actual preprocessor behaviour is summarised as follows:

1) Simplify (expand) all parts of the main relation ((11)) which are not explicitly named as unsimplifiable (13), given, or wanted, i.e. expand $\partial Z_{2n}/\partial E$.

2) From (10) and (13), the general form of that result is

$$\frac{\partial Z_{2n}}{\partial E} = \sum_{k=o}^{n-2} T_{nk} \frac{\partial \varepsilon_k}{\partial E} \quad , \tag{16}$$

   where T_{nk} has weight 2n - k - 2. A_{2n} and B_{2n-2} must be derivable from T_{nk} and the ε symbols.

3) No combinations of T_{nk} and the ε symbols composing terms of weight 2n can be made to match the given low-order values of A_{2n}. Therefore additional basic quantities with different weights must be generated by means of weight-changing operators. From (14) and (15), the only such operator is D. D acting on ε produces no new family of symbols, but D acting on T_{nk} gives the new family

$$T_{nk}^{(j)} = D^j T_{nk} \qquad (T_{nk} = T_{nk}^{(0)} \tag{17}$$

   with weights 2n + j - k - 2.

4) The simplest combination of symbols with weight 2n is now

$$T_{nk}^{(j)}\ \varepsilon_{k-j}.$$

Matches to low-order specimens of A_{2n} and B_{2n-2} suggest forms for coefficients, thus giving complete suggestions

$$A_{2n} = Z_{2n} + \sum_{k=1}^{n-2} \sum_{j=0}^{k-1} (-1)^{j+1} T_{nk}^{(j)} \varepsilon_{k-j}\ ,$$

$$B_{2n-2} = 2 \sum_{k=0}^{n-2} (-1)^{k} T_{nk}^{(k)} \qquad . \qquad (18)$$

Equations (16) through (18) express a transformation of the original problem into a form suitable for conventional SC. They can be checked by substitution into (11).

The lesson in this study is that the process of "suggestion" of correct algorithmic results for problems requires much less than the full apparatus of mathematical techniques present in large [3], [5] SC systems, provided only that the types of clues or keys which the process uses are chosen carefully. Section 2 describes a preliminary selection of keys. Further work may be expected to lead to an improved selection, and to more formal methods of description of useful preprocessing techniques.

REFERENCES

[1] J.A. Campbell, J.G. Kent and R.J. Moore, B.I.T. 16, 241, 1976.

[2] C. Weissman, LISP 1.5 Primer, Dickenson Publishing Co., Belmont, California, 1967, Ch. 20.3.

[3] W.A. Martin and R.J. Fateman, Proc. Second Symposium on Symbolic and Algebraic Manipulation, Association for Computing Machinery, New York, 1971, 59.

[4] A.C. Hearn, Comm. A.C.M. 14, 511, 1971.

[5] J.H. Griesmer and R.D. Jenks, Proc. Second Symposium on Symbolic and Algebraic Manipulation, Association for Computing Machinery, New York, 1971, 42.

[6] J.A. Campbell, SIGSAM Bull. A.C.M. 10(40), 46, 1976.

[7] P.O. Fröman, Annals of Physics, 88, 621, 1974.

6
An Implementation of Janus
K. Robert Elz & Peter C. Poole

1. INTRODUCTION

In the early 60's, proposals were put forward for a universal intermediate language which could serve as the common target for all high-level language compilers and as the common source for translators to machine code. The language was called UNCOL and the objective was to reduce the amount of work required to develop compilers to translate a variety of high-level languages for a number of different machines, the so called "m x n problem", i.e. m languages and n machines would require m x n compilers. Although considerable effort was expended on the design of the language, and a description of the first version published [9], the work did not come to fruition and the project was terminated. The reasons why the project was unsuccessful were never published, but it is suspected that it was perhaps too ambitious for the time, and the then state of knowledge of the structure of language, the structure of machines and the processes of translation. In 1974, a paper was published describing a language called Janus [1] which was intended to serve much the same purpose as UNCOL. It represented the culmination of a number of years of research into techniques for producing portable software, i.e. software which could be readily moved from one machine to another.

The function of Janus is to enable the compilation process to be split into two separate and distinct phases. The first phase is machine-independent and involves the translation of the high-level language into Janus. It consists of an analyser to recognise the constructs of the input language and transform them into a simpler sequence of operations more akin to those available on actual computers, and a code generator which takes this output and transforms it into Janus code. The second phase, which is machine-dependent, then involves the translation of Janus into the assembly code for the target machine. Currently, this process is carried out via the STAGE2 macro processor [10] and a set of macro definitions which relate the Janus abstract machine to the real machine. However, other translation processors could be used: e.g., simple translators expressed in a high-level language, or standard macro assemblers.

Once a program has been expressed in Janus, it could be, in principle, transported to any new machine and, since STAGE2 itself is portable, implemented via a full bootstrap technique which does not require any reference back to the originators of the software. However, Janus is not intended as a language for transporting all software. For example, there is no suggestion that it should be used by human programmers for, as will be seen from the description below, it would not be a particularly convenient language to use in

this manner. Instead, it is intended that programs will still be written in conventional high-level languages and that the portability of these programs will, in the main, rely on the portability of the compilers. Thus, the key question to study and answer is to what extent Janus can be used to port such compilers.

In an experiment to answer this question, a compiler for the high-level language PASCAL has been designed and constructed by the Software Engineering Group at the University of Colorado [7]. Early releases of the compiler have been distributed to various sites throughout the world so that implementations can be attempted, in order to evaluate the feasibility of the approach; in particular, answers are required to such questions as how much effort is required to implement the system, how difficult is the process and how efficient is the resulting product. The Department of Computer Science at the University of Melbourne is one such test site and an implementation of the PASCAL compiler on an Interdata 8/32 has been carried out. It is the purpose of this paper to summarise the results of this exercise and attempt an evaluation of the Janus project at this current point in time.

2. THE PASCAL COMPILER

The PASCAL-to-JANUS compiler, (hereafter called PASCALJ), was written in PASCAL itself and developed on a CDC 6400 at the University of Colorado. PASCALJ is designed to compile the complete standard PASCAL as defined by Jensen and Wirth [4], with a few extensions. A complete syntax of the language accepted by the compiler is given in the report by Ravenel [7]. The basic design of the compiler is very modular and it incorporates a parser automatically generated by a Syntax Analyser Generater [6].

The distributed software comprises the following programs.

(i) The PASCALJ compiler written in PASCAL (7000 source code lines).

(ii) The PASCALJ compiler expressed in Janus (30,000 source code lines).

In addition, there are two test programs written in Janus whose function is to aid the implementor in checking his realisation of the Janus abstract machine.

The basic steps required to produce a fully operational PASCAL compiler on a new machine are as follows:-

1. Map the Janus abstract machine on to the target computer and create a set of STAGE2 macros which will translate symbolic Janus assembly code into the assembly code of the real machine. Alternatively, other macro assemblers or specialized translators may be utilised.

2. Use the macros to translate the test programs into assembly code. Compile and execute them to check the implementation of the macros.

3. Using these macros, translate the Janus version of the compiler into assembly language.

4. Combine this program with an environment containing input-output routines, storage management routines, conversion routines, etc. and assemble to produce an operational PASCALJ compiler.

5. Modify the code-generating routines in the source of the PASCALJ compiler so that they produce machine code instead of Janus.

6. Translate this modified compiler via the PASCALJ compiler to produce a version expressed in Janus.

7. Translate this Janus version via STAGE2 macros (or whatever process is used in step 3) to produce an operational PASCAL compiler which generates code directly for the target machine.

Variations of this implementation sequence are possible. For example, a PASCAL compiler generating machine code could be produced by adding to the compiler (produced at Step 4) routines which translate Janus to machine code directly. These routines could also be written in PASCAL, translated by the compiler available at Step 4 and then added to the Janus version of this compiler.

3. THE JANUS ABSTRACT MACHINE

The Janus language is fully defined in a report by Waite, and Haddon [12] which contains the complete syntactic definition of the language as well as a description of the semantics. The following summary is based on this report but describes the language in terms of the underlying abstract machine rather than by formal methods of syntax definition. Although this relies on the intuitive understanding of the reader, it does enable the flavour of the language to be conveyed more readily.

The Janus abstract machine consists of a memory and a processor. The latter contains three registers - a base register for indirect access to memory, an index register and a condition code register which may be set to the values: LT, EQ, or GT. In addition, the processor has access to an operand stack of unspecified size. It is the source of operands and the destination of results for most instructions.

The repertoire of executable instructions available in Janus may be divided into four groups.

(a) Transmission: These are used to move information around within the Janus machine. Instructions exist to transfer information from one part of the memory to another and between memory and the operand stack. Values can also be loaded into the base and index registers. A special TEST instruction provides a mechanism for transferring the condition code onto the operand stack as a boolean value.

(b) Computational: These involve the arithmetic, logical

and conversion operations, which take some number of operands from the stack, process them, and return some number of results to the stack. This, implies that the instructions are "addressless" and indeed such a form is allowed. However, if an instruction specifies an operand explicitly, then this is moved onto the stack before the processing commences. One instruction which compares its operands arithmetically is available to set the condition code.

(c) Control: These cause control to be transferred to a specified location. Transfers may be direct or indirect, conditional or un-conditional, but in all cases the target must be an executable instruction. Case selection is provided, a particular statement being selected on the basis of a match between an index value and a constant associated with the statement. Procedure entry and exit also fall into this class. A complete procedure call is the sequence of statements specifying both the transfer of control and the calculation of the arguments to be passed to the procedure. When control is returned to the caller from a procedure, a result may be returned on the operand stack.

(d) Allocation: The memory of the abstract machine may be allocated statically at compile time or dynamically at run time. The instructions in this group are used to manage the latter operation. Storage may be claimed (or released) from two areas, the stack (not to be confused with the operand stack) and the heap. Stack storage is released in the reverse order to which it was claimed, whilst there is no particular relationship between the allocation and release of heap storage.

The memory of the Janus abstract machine may be regarded as a collection of named operands comprising constants and variables. The latter are grouped into several different categories which reflect the storage allocation strategy and the access mechanism. The operands may be either elementary or composite. An elementary operand is one which has no structure discernible to Janus. A composite operand, on the other hand, is one which is constructed from other operands according to formation rules. Its components may be either elementary or composite, but ultimately it can be described in terms of the elementary operands.

Each operand has a mode and a value. Modes may be primitive, corresponding to the elementary operands, or composite. Janus recognizes six pre-defined primitive modes:

(i) ADDR which specifies a location in the data or code space of the program.

(ii) BOOL which may take the value either true or false.

(iii) CHAR which is a processor representation of a single character.

(iv) INT which is an exact representation of an integer value.

(v) REAL which is a processor approximation to the

value of a real number.

(vi) PROC which specifies the entry point of a procedure and the environment within which it was declared.

Additional primitive modes may be defined for particular applications but these would constitute an extension to Janus.

Composite modes, which must be defined explicitly, are constructed from other modes by applying formation rules. Two such rules exist for Janus, defining the data aggregates, array and record. The definition of an array mode specifies the number and mode of the elements in an array of this class and carries an indication of how the elements are to be aligned. This enables the program to indicate to the translator the importance of storage economy as opposed to access time; i.e., whether array is to be packed or not. The definition of a record mode establishes the displacement of each field from the beginning of the record and allows the determination of the size and alignment of the whole record.

Access to a variable in memory is made via a reference which in its most general form specifies a base, an index and a displacement. The base is just the address of the variable in the Janus memory: the index and the displacement are respectively used to select an element of an array variable and a field of a record variable. The interpretation of the base depends upon the category of the variable which qualifies the reference. Janus supports the following categories of storage:

(i) STATIC: the base is interpreted as the absolute address in the Janus memory.

(ii) DISPn: n is an integer specifying the procedure level. When a procedure at level m is activated, a set of activation records for procedures at levels 0 to m-1 is accessible. If a reference in a procedure at level m is qualified by a DISPn category then the base is interpreted as the location of the variable within the activation record at level n (0<=n<m).

(iii) LOCAL: a special case of DISPn for variables within the activation record of the current procedure. This is provided to assist optimization since these are likely to be accessed very frequently.

(iv) PARAM: base is interpreted as the location of an argument to the current non-recursive procedure.

(v) BASED: base is interpreted as a relative address in the area addressed by the Janus base register.

(vi) TEMP: base is taken to be the name of a temporary known only to the referencing procedure. The variable must be of primitive mode, and might be known in only part of the procedure. This is provided (as a distinct type for LOCAL) to allow the implementor to use a register, or other fast access storage.

The reference index may contain a static component known at translate time and a dynamic component computed during execution. The static component consists of an integer expression associated with a mode identifier. The value of the expression gives the integral number of units of the mode specified.

The symbolic format of an instruction in JANUS is an operator followed by (optionally) a modifier and (optionally) a specification of an operand. The operator defines a basic operation and the modifier enables a particular variant of this operation to be selected. Thus, in the instruction

ADD INT.

ADD is the basic operation while INT is the modifier which specifies that the mode of the operands is integer. There may also be flags attached to the operator to indicate variations from the normal operation, e.g. the "N" flag causes the return of results to the stack to be suppressed. If no operand is specified, then, in most cases, it is assumed to be on the operand stack. An explicit operand, on the other hand, may specify a constant, a variable or an address.

The structure of a Janus instruction is illustrated in fig. 1. The instruction adds a real variable to the top

ADD REAL DISPO A3 (2)X1* B1

BASE INDEX DISPLACEMENT

OPERATOR MODIFIER CATEGORY REFERENCE

OPERATION OPERAND

IMPERATIVE STATEMENT

Fig. 1: Structure of a Janus Instruction

element of the operand stack from the B1 field of a record of mode X1 which is the third element of an array A3 of such records in the activation record of the main procedure.

The record mode definition of X1 is given by:

```
RECORD X1.
FIELD INT ALIGN A1.
FIELD REAL ALIGN B1.
RECEND X1.
```

for a class of records containing one integer and one real variable which are to be aligned for rapid access. The mode of A3 would have been defined by:

```
ARRAY X1 ALIGN M3(10).
```

and space reserved for it by the declaration:

```
SPACE M3 DISPO A3.
```

All code in a Janus program is contained in procedures which may be recursive or non-recursive; one of these is the main program at which execution commences after control is transferred from the operating system. It contains declarations of static storage which is allocated at compile time and may be initialized. During activation, each procedure has access to local variables and parameters which are passed to it by value. Further, it operates within an addressing environment which allows it to access certain non-local objects. Each procedure has an activation record which comprises the following components:

(a) linkage information to effect return of control to the caller

(b) parameter values and local storage whose size is known at compile time.

(c) composite parameter values and local variables whose size must be determined and space allocated each time that procedure is entered.

All procedures must be declared. Each declaration consists of a procedure heading followed by a procedure body. The former contains the name and class of the procedure together with a description of the formal parameters. The body comprises declarations of local variables, constant definitions, label declarations and executable statements. Fig. 2 which is based on an example from the Janus report shows the equivalence between a procedure in Pascal and its translation in Janus.

4. IMPLEMENTATION

The design of Janus has been strongly influenced by the requirement that it be easily mappable on to the majority of real computers and this has proved to be the case here. No great difficulties were encountered in realising the abstract machine on the 8/32. Nevertheless, as will be seen

Pascal	Janus
function ex(a:integer):real;	BEGIN REAL DISP1 P1. (declare procedure P1 at level 1 returning real result)
	PARAM INT DISP1 A1. (specify integer parameter A1)
var	PAREND REAL DISP1 P1. (terminate parameter specs.)
i:integer;	SPACE INT DISP1 I3. (declare local integer variable I3)
begin	LOAD INT LOCAL A1. (value of A1 to operand stack)
i := a+2;	ADD INT AINT 2. (add integer constant 2)
	STORE(N) INT LOCAL I3. (store result in local variable I3)
ex := i;	REAL INT LOCAL I3. (convert integer variable I3 to real variable on stack)
	RETURN REAL DISP1 P1. (exit from procedure)
end;	END REAL DISP1 P1. (end procedure declaration)

Fig. 2: A Pascal Procedure and its Janus Equivalent

from the discussion in a later section, the implementation proved to be far from easy, and somewhat time-consuming.

4.1 The Target Machine

The Interdata 8/32 is a midi computer almost identical in structure to the IBM/360. It is byte oriented with additionally full and half word operations on 32 and 16 bit operands respectively. There are 16 general purpose registers for indexing, fixed point arithmetic and logical operations etc., and a further eight registers for floating point operations.

The major difference between the 360 series and the Interdata is that, on the latter machine, all storage is directly addressable (up to one megabyte). No base register is required as it is on the 360. To achieve this, while still retaining reasonable instruction lengths, the assembler

selects one of the three possible instruction formats available for each reference - index register plus displacement, index register plus signed offset relative to the program counter or two index registers plus signed offset. The latter is a 48 bit instruction compared to 32 bits for the others. The availability of these formats and instructions such as a short jump relative to the program counter means that optimization of the assembly code is possible. This is carried out by requesting optional extra passes during assembly.

The repertoire of instructions includes most of those available on the 360 although there is no "move-character" operation. Extra facilities available include instructions for manipulating bit arrays and circular lists. The latter may be used to organize a push down stack in memory. There is also a writeable control store of 512 words which may be entered from a user program. This enables frequently used sequences of assembly instructions to be encoded and obeyed at the micro-code level for increased efficiency.

4.2 Mapping the Abstract Machine

In mapping Janus on to the Interdata, the first decisions that have to be made concern how the "hardware" of the abstract machine can be realised on the actual machine.

The Janus operand stack, which may be of indefinite size, is placed in memory since there is no direct hardware equivalent. As mentioned above, there are hardware instructions which enable a section of memory to be organized as a push down stack. However, no use was made of these instructions, since the simulation of such a stack at run-time would have been far too inefficient. This is because the Janus machine permits arithmetic operations on stack elements, as well as LOAD and STORE, whereas the Interdata does not. Thus the operation ADD without an operand would need to be translated into two instructions to transfer the top cells of the stack to registers, an instruction to add the registers, followed by one to store the result back on the stack. The overhead produced by such a sequence would have been unacceptable, even with the stack housekeeping being performed at the micro-code level. Further, the lack of a finite depth to the stack would have involved, in many cases, the addition of instructions to test for stack overflow, thereby increasing the overhead even more.

The mechanism actually used relies upon simulating the stack operations at translation time, a process made possible by the specification in Janus that the part of the stack local to a procedure is void at any label within that procedure. Thus, within any procedure, it is always possible to determine the maximum extra number of cells of the stack that will be added by the procedure. Hence, by reserving space in the activation record of each procedure for the maximum operand stack size of that procedure, references to entries on the stack may be considered to be references to local variables, and no address computations (other than hardware indexing to add the address of the base of the activation record) are necessary at run time. The translation

of the addressless ADD operation now becomes LOAD, ADD, and STORE, a much faster sequence. Further, no testing for stack overflow is required.

Obvious savings can also be made by retaining the top element of the stack in a register wherever possible. Two registers are reserved for this purpose (two are required to allow for PROC mode objects which consist of the address of the code location identified by the object and a pointer to the environment (activation record) in which it was created, and to facilitate multiplication/division). Whenever a value is loaded onto the Janus operand stack, it is retained in a register until removed (in which case it will never have been stored in memory) or until the register is required to load another value. The ADD operation may now, in suitable cases, reduce to a single Interdata ADD instruction.

Because of the hardware distinction between fixed and floating point arithmetic, another register must be used for REAL operands. It then becomes unnecessary to transfer to memory the value currently on the stack if it is of mode REAL, and the new element is of another mode, or vice versa. To simplify this process at translation time, the operand stack in the activation record has been split into two distinct sections, one for reals and the other for the remaining modes.

The Janus registers (BASE and INDEX) are each allocated to a fixed hardware register. The Janus specification states (in more formal terms) that almost any instruction will void the contents of both of these registers, causing compilers to generate code to reload either or both registers more often than is needed for the Interdata. These registers have been defined to be volatile, to allow the implementor the greatest possible flexibility in his implementation technique, and in fact, this feature has been used in a small way. When Janus specifies that the INDEX register contains a number which is to be interpreted as a multiple of some data structure size, it is necessary to multiply the contents of the register by that size to obtain the correct integer for address modification. This can be done in situ rather than by copying the INDEX register to a temporary since the contents do not need to be preserved.

The Janus condition code is realised by the hardware condition code, with each of the Janus values being represented by one of the possible values on the Interdata. While it has been possible to use this method in the current implementation, a more efficient use of storage in future implementations might produce complications. Currently, characters are stored as half words (see below) and any comparison is an arithmetic one which sets the actual condition code to the value expected by Janus. If the characters were stored as bytes, then a compare logical instruction would have to be used and the condition code settings would no longer correspond to those required for Janus. Some mechanism to reset the condition code would therefore be required so that the full range of conditional jumps permitted by Janus could be implemented correctly.

Once the registers have been mapped, the next question concerns storage. The Janus memory is a named set of operands and nothing is said in the definition about their inter-relationships. They can therefore be mapped in a straightforward manner onto the linearly addressed memory of the Interdata. To simplify the translation and economise on storage for the initial implementation, all data types are mapped on to half words with the exception of ADDR and PROC mode objects which require 4 and 8 bytes respectively. No provision has been made as yet for REAL since the Pascal compiler does not make use of this mode itself. Further, no distinction has been made between packed and aligned structures. Access is simplified but space is wasted. The fact that BOOL and CHAR modes are the same as INT allows a uniform treatment of LOAD, STORE and COMPARE.

Another important decision that has to be made in the mapping concerns the way in which the activation records for procedures are organized. Janus specifies that a stack be maintained for the activation records of procedures active at any instant. The activation record contains system control and linkage information, parameters passed by value, local variables and the operand stack. The layout is as shown in fig. 3.

The first section contains the dynamic link (to the previous activation record), the return address of the procedure and the contents of two registers which must not be destroyed by calls. The static nesting level is also included. Space has been reserved for the dynamic nesting level and a pointer to the procedure being called (the owner of this A.R.) which will be useful in the final system to provide traceback information in case of error. As yet, these are not used. Following this is a display [13] which contains all static links. The concept of a display rather than a static chain as used in some other Pascal implementations [3] was chosen primarily because the Interdata does not have a simple, elegant way to chase back along a static chain. The code required at each reference to a non-local variable would more than offset the saving gained by eliminating the display. One improvement which has been added, is to duplicate the last two display entries in registers for quicker access. In PASCALJ itself, this is remarkably effective, since the display never contains more than two entries.

The greatest problem to overcome in defining the mapping of Janus onto the Interdata was to invent a suitable procedure calling sequence. Janus is very vague in this area, and there was no standard on the Interdata worthy of consideration. The solution which has been adopted is not entirely satisfactory, and is still giving cause for concern. However, nothing demonstrably better has yet been invented. All Janus parameters are passed by value, and must be placed in the activation record of the called procedure, since it refers to its parameters as local variables. As parameters are often computed, it seemed best to place the value directly into the new activation record, rather than saving it in some temporary location and later copying it across to the A.R. However, to do this, the activation record must exist, and only the procedure being called knows what size A.R. it requires. To cope with situations like:- y:=f(i,g(x),j)

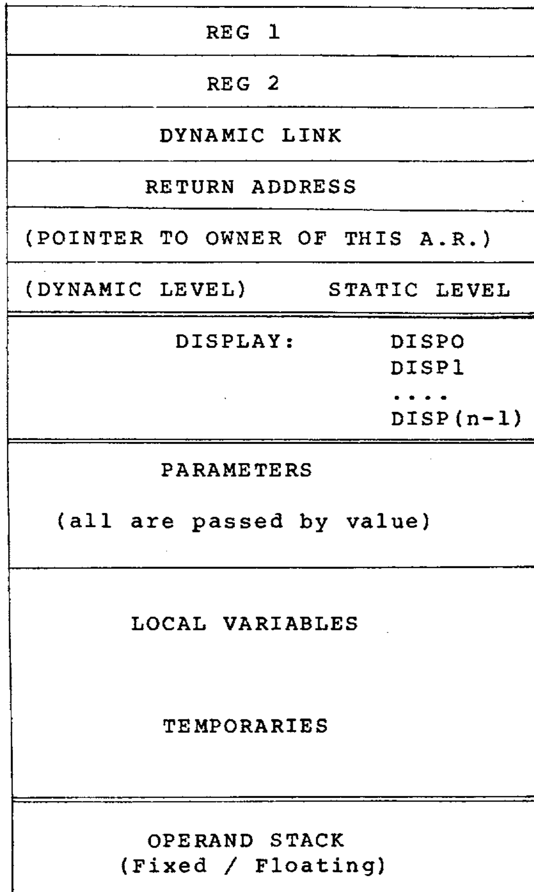


Fig. 3: Activation Record Layout

it could not be assumed that the new activation record would immediately follow the current one (or the value of x would overwrite that of i). Rather than dynamically maintain a top of stack pointer, and increment it after each parameter has been added, it was decided to enter the procedure before computing any arguments. The procedure then obtains its A.R. and returns the address at which it requires it parameters. (This also neatly copes with the problem of non-recursive procedures, which do not have A.R.'s). The calling procedure then computes the arguments, and stores them, in order, beginning at the address given. When all parameters have been transferred, the called procedure is re-entered whereupon it calls a standard routine to update its display, etc. and then executes normally.

Procedure termination is much simpler. Functions load their value into the "top of operand stack" register, then both procedures and functions exit to a system routine which

restores the environment to the state that existed at the call (except for the function result). Procedure entry and exit is illustrated in fig. 4.

Statement in procedure j7	Effect at run-time
RCALL BOOL DISP1 R J71.	Boolean function J71 is entered and obtains space for its A.R. The two registers which must be saved are, and a return is made to J7 (the caller) with a register containing the address for parameters.
RARGIS(N) INT AINT 0.	The first argument of J71 is to be the integer zero. This is stored (by J7) in the new A.R. of J71.
RARGIS(N) REAL LOCAL V72.	The second argument is the REAL number contained in local variable V72. This is also stored in the new A.R.
RECEND BOOL R J71.	There are no more arguments. J71 is reentered and calls a system routine to save its return address, the A.R. dynamic link, and to update its display. It then continues as the current procedure. When finished, it loads its result (a BOOL value) into the "top of stack" register, then branches to a system routine. This routine restores the environment of J7, which may then
STORE(N) BOOL LOCAL V73	Store the result, or otherwise process it as desired.

Fig. 4: Example of Janus Procedure Call (on Interdata)

4.3 Translation

Once the mapping has been effected, then the next task is to encode the macros which will enable STAGE2 to translate Janus to CAL, the assembly language on the Interdata. This turned out to be a difficult task which made very heavy demands on memory. To help alleviate this situation, a decision was made to employ an almost totally context-free translation from Janus which has resulted in certain inefficiencies, particularly in regard to register usage. Further, a multi-pass system was required. The first pass consists of one in which the characteristics of the target machine are embedded in the Janus code. This involves the replacement of the references to manifest constants with references to their values (and the elimination of the now unnecessary definitions), the computation of the sizes of the composite mode objects and the replacement of references to the mode with references to the size.

The second pass performs the actual Janus translation and requires about 1,200 lines of STAGE2 macros. The translation run uses about 170 K-bytes of memory to produce a CAL version of the Pascal compiler which consists of about 38,000 lines of code. The time needed for this pass is approximately 40 mins. (a proportion of this is due to inefficiencies in the operating system rather than to the processing speed of STAGE2). Even the CAL assembly run requires a minimum of 40 mins., if unoptimized code is requested. If a considerable degree of optimization is required, then this time can rise to 5 to 6 hours. The resultant program occupies 110 K-bytes of storage excluding the run time stack and heap.

4.4 Run Time Environment

Once a machine language version of the compiler has been produced, it must be linked with the run time environment. This comprises various utility subroutines as well as a number of routines which are specified by the distributors of Janus. An example of the first type of routine is one where a particular Janus statement is performed by a subroutine call rather than by in-line code. Typically Janus statements requiring this treatment would be storage management (GRAB STACK, FREE HEAP), conversions from REAL to INTEGER mode and vice versa, operations on composite mode objects (CMPM and MOVE). Routines are also likely to be needed for interaction with the operating system, for effecting program entry and/or exit and possibly to maintain system status at procedure call and return.

The second class of routines may contain anything that is needed to enable the software to execute and which has not been distributed in Janus. Calls to routines of this type are treated just like calls to Janus procedures except that the modifier in the CALL statement specifies that a system routine is involved. An example of such a routine is one providing input and output since Janus specifies no other mechanism for communication with the outside world. Other possible routines are dependent upon the characteristics of the software being implemented. Typically, one might include mathematical routines (square root, logarithm, circular and hyperbolic functions etc.) and routines for interaction with the operating system (file handling etc.). It should be noted that many of these routines can themselves be written in Janus and, ultimately, it is planned to distribute as many as possible with the software.

For the implementation of PASCALJ under discussion, nine routines of the first type and eight of the second have been implemented. The latter are all concerned with input/output and represent the minimum number of the operations in the compiler itself. Of the former, five routines are required for storage management, two for housekeeping at procedure entry/exit and two for operations on composite mode objects.

4.5 Operational Characteristics

The PASCALJ compiler based on the February 1976 release is currently operational on the 8/32 and has been validated to the point that it can compile itself. This process

requires 190 K-bytes and takes 16 mins. of elapsed time. If generation of output is suppressed, then the process requires 6 mins., i.e. 1150 lines of PASCAL per minute. This is to be compared with the Interdata FORTRAN which translates at the rate of 2500 lines per minute and the CAL Assembler which processes about 1000 lines per minute.

Although it is difficult to compare storage requirements of the same program for machines of different word lengths, the figure given by Ravenal (1975) for the CDC machine indicate that 40 K decimal 60-bit words are required by PASCALJ as translated by PASCAL2 to compile itself. The Interdata 8/32 implementation requires 48 K decimal 32-bit words. Another bootstrap of the compiler has been carried out by Ravenal [7] on a Xerox Sigma 3 computer. He reported that he was able to create a working system in 48 K decimal 16-bit words using a simple overlay structure of a root and two segments. He also noted that the first release of the compiler produced code which was less than optimal and suggested that a ratio of 3 to 1 over equivalent hand-written code was a reasonable estimate of its inefficiency. We are in substantial agreement with this figure.

5. EVALUATION

The results from work described in this paper seem to indicate that PASCALJ is not very portable, at least, at the present time. This conclusion is supported by other groups who have attempted an implementation and there is little doubt that the system is not yet ready for general distribution. The implementation is time consuming with the current tools and is probably outside the reach of most people. The effort required to implement Janus is still considerably greater then the goal of a few man months. Nevertheless, working systems have been produced.

One approach to improving the portability of Janus involves developing a hierarchy of abstract machines, each slightly simpler than its predecessor, as proposed by Poole [5]. Waite has already commenced work on this approach and is defining a number of levels J1, J2, J3, ... each of which can be derived in a machine independent manner from its predecessor. For example, the J1 abstraction [11] lies just below Janus in such a hierarchy. It makes certain assumptions about the structure of computer memories and then makes use of these in the implementation of the Janus named operands. Whereas the Janus memory is regarded merely as a collection of named operands which possess a tree like structure, J1 imposes additional structure corresponding to the properties of physical storage. For example, the J1 memory is assumed to consist of an array of words, a word being the smallest addressable unit of the memory, and containing a number of bits or a number of bytes.

The lowest level of abstraction will be one which eases the task of the implementor and allows him to transfer the system with the minimum expenditure of effort. If there is a good match between the abstract machine at this level and the real machine, then code of reasonable efficiency should result. If not, more effort will be required to implement

one of the higher level.

In another experiment to explore the portability of Janus, we are attempting to define a representation for Janus which will allow it to be interpreted. The major effort in transporting the system will then merely be to write the interpreter. This approach has been tried with other portable systems [8] and the usual result is that the transported software is very inefficient. However, if the form of the interpretive code is compact enough, it can serve as the basis for a hybrid [2]. The optimisation then required after the interpretive form has been implemented involves generating such a hybrid. Care has been taken during the implementation of Janus on the 8/32 to ensure that identical storage layouts have been used in both cases so that the hybrid is possible.

Another criticism of the Janus approach made by a number of implementors is that syntax of the language is too strongly biased towards STAGE2 processing. There is little doubt in our minds that STAGE2 is not the best tool for implementing Janus directly as has been done in this exercise. It may still be the appropriate tool for one of the lower levels of abstraction, particularly in view of its own portability. However, at the top level, a faster method of translation is required. The next release of the system to be made in February 1977 is expected to incorporate changes to the grammer of Janus so that the basic symbols can be extracted by a simple lexical analyzer.

ACKNOWLEDGEMENTS

Our thanks are due to the Software Engineering Group at the University of Colorado who supply the software and the supporting documentation, also to Mr. C.B. Mason of the Australian Atomic Energy Commission, one of the developers of the compiler.

This work is supported by the Australian Research Grants Committee, Grant No. F76/15583.

REFERENCES

[1] S.S. Coleman, P.C. Poole and W.M. Waite, The Mobile Programming System: Janus. Software, Practice and Experience, 1, 1974, 5.

[2] R.J. Dakin, P.C. Poole, A Mixed Code Approach. Computer J, 16, 1973, 219.

[3] C.O. Gross-Lindemann, H.N. Nagel, Postlude to a Pascal Compiler Bootstrap on a DECSystem10. Software, Practice and Experience, 6, 1976, 29.

[4] K. Jensen, N. Wirth, Pascal User Manual and Report. Springer Verlag, Berlin, 1974.

[5] P.C. Poole, Hierarchical Abstract Machines. Proc Culham Symposium on Software Engineering, HMSO, London, 1971.

[6] B.W. Ravenel, SAG: An LL(1) Parser Generator. Technical Report SEG-75-2, Software Engineering Group, Department of Electrical Engineering, University of Colorado, Boulder, Colorado, 1974.

[7] B.W. Ravenel, PASCALJ User's Guide and Implementation Notes. Technical Report SEG-75-3, Software Engineering Group, Department of Electrical Engineering, University of Colorado, Boulder, Colorado, 1975.

[8] M. Richards The Portability of the BCPL Compiler. Software, Practice and Experience, 1, 1971, 135.

[9] T.B. Steel, A First Version of UNCOL. Proc AFIPS WJCC, 19, 1961, 371.

[10] W.M. Waite, The Mobile Programming System: STAGE2. CACM 13, 7, 1970, 415.

[11] W.M. Waite, Janus Memory Mapping: The J1 Abstraction. Technical Report SEG-76-1, Software Engineering Group, Department of Electrical Engineering, University of Colorado, Boulder, Colorado, 1976.

[12] W.M. Waite, B.K. Haddon, A Preliminary Definition of Janus. Technical Report SEG-75-1, Software Engineering Group, Department of Electrical Engineering, University of Colorado, Boulder, Colorado, 1975.

[13] N. Wirth, The Design of a Pascal Compiler. Software Practice and Experience, 1, 1971, 309.

7
Data Abstraction Facilities
Jan B. Hext

1. INTRODUCTION

The basic idea of data abstraction is to separate the conceptual aspects of data processing from the concrete details of data representation.

The name "data abstraction" may sound somewhat obscure and forbidding, but in fact the basic concept is nothing new. On the contrary, mathematicians have seldom dealt with anything else. The proposition 2 + 2 = 4, for example, is an abstraction: its validity does not depend in any way on the number two being represented in decimal or binary or anything else. Indeed, the whole number system is an abstraction. Its essential features are that we can do various things with it, such as counting, addition and multiplication. But the means by which we do them in practice are, for the most part, irrelevant. As long as we obtain the right answers, we do not normally concern ourselves with the techniques which are used in the process.

Of course we still have to define what we mean by counting, addition and multiplication. Fortunately there is a general consensus on these matters which spares us the need for giving detailed definitions. But if someone were niggling enough to insist on a rigorous, formal definition, we could refer him to *Principia Mathematica* [18]. The axioms and theorems listed there should hopefully keep him out of our hair for quite a while. He could even amuse himself by proving that the addition algorithm used by his pocket calculator satisfied these axioms.

Thus data abstraction is familiar enough to all of us and it was only with the advent of computers that people became seriously embroiled in problems of representation. The trouble was that the pendulum then swung over to the other extreme and the conceptual processes tended to disappear from view. This was especially the case when programming in the primitive machine codes and it is still a danger for anyone working in assembler. But, roughly speaking, as we ascend to higher-level languages the trend is away from matters of representation and more towards abstraction. For example, when we declare

```
integer i
```

in Algol 60, we are committed to an abstraction of the type *integer*. The only thing we can do with i, besides copying it, is to apply the standard arithmetic operations to it. We cannot manipulate its representation in any way or otherwise take advantage of it. For example, we cannot use a left shift to multiply i by two. In languages such as BCPL, this

is possible. But it would obscure our real intentions and, since it requires certain assumptions about the representation being used, it would jeopardize the correctness and portability of the program. Such tricks are therefore frowned upon and, if the language prevents them from being used, so much the better.

At the other end of the complexity scale, we may consider a data base. This may be accessible either through a data manipulation language or through a query/update language. The former allows the user to manipulate the pointers, indexes, etc. which are used in the representation, thus allowing him to wreak havoc with the system. In contrast, the latter gives the user an abstract view of his data, devoid of representational details. He can ask for data to be changed or retrieved, but the mechanisms for doing this are hidden from his sight and could, in principle, alter from day to day. This has two major advantages: it makes life simpler for the user and safer for the data base.

These two examples show how a programmer may have abstract data types thrust upon him. But what if he wants to set up his own? For example, supposing he wishes to construct and manipulate binary trees. In Pascal he might define a type for the purpose:

```
type tree = ↑node ;
     node = record val          : nodetype ;
                   left, right : tree
            end ;
```

and then declare the variable

```
var t : tree ;
```

The declaration looks quite convincing. But in fact there is no guarantee that t will always be a binary tree. One moment it might be the tree of Figure 1 (a) and the next it might be the graph of Figure 1 (b).

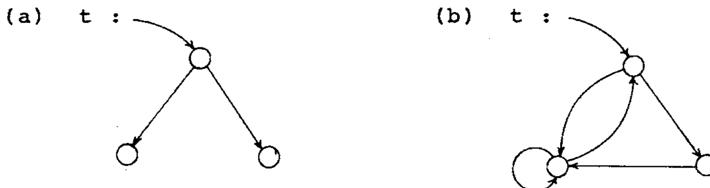


Fig. 1: The Type 'tree'

If the programmer is to guarantee that t remains a tree, he must restrict the range of operations that can be applied to it. For top-down purposes he might allow new nodes to be added to the leaves. For bottom-up purposes he might allow two existing trees to be linked to a new root. But uncontrol-

led access to the links must be prevented. This done, he can be sure that objects of type tree really will be trees.

We are thus led to look for a language which allows a programmer to define his own data types in this way. Four main features are required for the purpose:

(i) a convenient notation for naming a new data type and the operations to be available on it.

(ii) the ability to specify a particular representation for that data type, and to define the operations in terms of it.

(iii) the ability to create objects of the new type.

(iv) a guarantee that objects of the new type can only be accessed by means of the defined operations.

Parts (i) and (ii) are normally combined into a single definitional facility. The following table lists some of the languages which do this and the names that they give to the resulting module:

Language	Module name	Reference
Simula 67	Class	Dahl et al. [6]
Clu	Cluster	Liskov and Zilles [13]
Model	Space	Johnson and Morris [11]
Alphard	Form	Wulf [20]
Concurrent Pascal	Monitor	Brinch Hansen [3]
Modula	Module	Wirth [19]

Our purpose in the following sections is to review the facilities for data abstraction which these languages offer and to discuss some of the issues involved.

2. EXAMPLE

Of the above languages, the only one that is at all widespread is Simula 67. Fortunately, despite its age, Simula is so advanced in its abstraction facilities that we may take it as a standard by which to judge the others. We shall therefore use it at this stage to give a simple illustration of data abstraction, taking the well-worn example of a stack.

Let us suppose, then, that we are developing a program which at some stage makes use of a symbol stack. We shall assume, for convenience, that the symbols are represented by type <u>char</u> and that a stack of at most 30 of them will be sufficient. But what exactly <u>is</u> a stack? The answer given by an abstractionist is that in essence it is an object to which we can apply certain operations:

push - for loading an element on the stack

pop - for removing it

```
top    -   for copying the top element

empty  -   for testing if the stack is empty
```

There is room for minor variations here, but these four will do for our purposes. We shall also need the ability to create a stack and, since storage is probably limited, we must impose an upper limit on its size - in this case, 30.

To handle this notion, Simula uses a class declaration. The following is the appropriate skeleton:

```
class symbolstack (size) ; integer size ;

begin

        <declaration of required structures> ;

        procedure push(x) ; char x ;

            <body of push> ;

        procedure pop ;

            <body of pop> ;

        char procedure top ;

            <body of top> ;

        boolean procedure empty ;

            <body of empty> ;

        <initialization>

end symbolstack ;
```

This states that an object of type symbolstack has a certain size and that the procedure 'push', 'pop', 'top' and 'empty' can be applied to it. Its representation by means of a specific structure remains to be filled in, along with the bodies of the four procedures and the initialization section.

This, we may say, captures the essence of what is meant by a symbol stack. In fact, though, it does nothing of the sort unless we know what is meant by 'push', 'pop', 'top' and 'empty'. Fortunately most of us probably do know what they are all about. But if we wanted to be rigorous, we would have a supply a Principia for stacks, laying down the axioms which they should satisfy. We shall return to this point later. Meanwhile, assuming that the operations are intuitively understood, we can proceed to the second stage of our definition, namely their implementation.

Here we have a choice of techniques, so naturally we aim for the one which gives the best performance for our particular application. An obvious possibility is to use a vector, V say, for holding the elements, and a pointer, p say,

to the top position filled. Accordingly, we can fill in the representation as follows:

```
char array V[1:size] ;

integer p ;
```

These are called the attributes of the class. We may note in passing that the parameter size is also an attribute, being treated as though it were a local variable just like p (as in Algol's call-by-value). With this decision made, the bodies of the four procedures can be coded. For example, the body of 'push(x)' becomes:

```
begin p := p + 1 ;

    if p > size then report ("stack overflow ...") ;

    V[p] := x ;

end push ;
```

We shall not pursue the effect of 'report' at this stage, but simply note the need to handle the stack overflow exception. It raises problems similar to those of arithmetic overflow.

Following the procedure declarations comes the initialization section. This is executed whenever a symbolstack object is created. Since the stack will presumably start off empty, the only action required is to set p to zero. In fact, Simula will do this by default, so this section can be left blank.

In order to create a symbolstack object, we invoke the operator new. Thus the expression

```
new symbolstack (30)
```

will create and initialize a new symbol stack of size 30. The expression yields a pointer to the object as its result and this pointer may be assigned to a variable of type ref (symbolstack). Thus we could declare

```
ref (symbolstack) S ;
```

and then assign a stack to it using the operator ':-':

```
S :- new symbolstack (30) ;
```

The symbol stack can now be referenced through S, using the dot notation. For example, we can write statements such as

```
S . push ('A') ;

x := S . top ;

if not S . empty then S . pop ;
```

The notation may seem a little unnatural, since it would be

more conventional to write the procedure calls as 'empty(S)', 'pop(S)', etc., treating S as an ordinary parameter. But since the definitions do not have a corresponding formal parameter, S has to be supplied in this special way outside them. An alternative approach is illustrated in the next section.

The Simula class facility has several other features. But before discussing them, we shall consider some of the variations on these basic concepts.

3. VARIATIONS

We have already noted the great variety of names which other languages give to their abstraction modules. The notations used within them are equally varied. In some cases, the differences are purely syntactic; but in others they represent significant differences in meaning. As an example, we may take the Clu definition of a symbol stack:

```
symbolstack : cluster (size : integer)

    is push, pop, top, empty ;

    rep (bound : integer)

            = (p : integer ;

               V : array [1 : bound] of char) ;

    create

        s : rep (size) ;

        s.p := 0 ;

        return s ;

        end

    push : operation (s : rep, x : char) ;

        s.p. := s.p. + 1 ;

        if s.p. > s.bound then error ;

        s.V[s.p] := x ;

        return ;

        end

    pop :  ....

            ....

end symbolstack
```

This has obvious counterparts to the components of the Simula definition. The heading summarizes the abstract view

of the type. The rep part gives its representation and the create part its initialization. Then follows the definition of the four operations.

Leaving aside the trivial syntactic differences, we may begin by noting the introduction of the attribute bound. This is used to record the value of size, since the parameters of a cluster are not otherwise preserved. This makes their preservation optional, though it is debatable whether the benefits justify the greater complexity.

The symbol rep introduces the representation part. However, it is also used subsequently, in the specification of the parameter s, as though it were a type. In fact it denotes what is sometimes called a concrete type. The point is that, strictly speaking, if s were specified as the abstract type symbolstack, we could not then access its components. The use of a concrete type avoids this difficulty. Of course, the corresponding actual parameter must have type symbolstack; so, when it is passed to the procedure 'push', say, there is an implied conversion from the abstract to the concrete. The language Model recognizes this explicitly: the formal parameter is given the abstract type and, whenever its representation is referred to, it has to be prefixed by the "concretion" operator '#'.

In contrast to the Simula version, the parameter lists for the operations include the parameter s for the symbol stack which is to be operated on. This means that we can now call 'empty(S)', 'pop(S)', etc. in the more conventional way. On the other hand, all occurrences of 'p' and 'V' within the definitions have to be prefixed by 's.', which is rather tedious. Occasionally, of course, we wish to refer to s as a whole, rather than to its components. In this case the Clu version is simpler; Simula has to use the expression 'this symbolstack' for the purpose.

Another point worth noting concerns the creation of objects of the defined type. We have seen how this is done in Simula, where the operator new creates the required object and returns a pointer to it as a result. In Clu, the objects are created by declaring variables of the specified type. Thus the declaration

```
S : symbolstack (30)
```

creates a symbol stack of size 30 which is permanently bound to S. This avoids the use of pointers, with all their attendant hazards. On the other hand, pointers can sometimes be used to good advantage, so it is not clear whether they should be allowed or not. A discussion of the issues is given by Berry, Erlich and Lucena [1], who argue that pointers should not be allowed except within type definitions. Whatever the conclusion, though, Simula is certainly open to criticism for requiring pointers in every case.

4. EXTENSIONS

Besides these variations on the basic theme, we may list the following four extensions to it which have been made by

some of the languages mentioned.

(i) The language Model allows operators to be defined, such as '+' and '/', so that they can be applied to the new type. This is a familiar concept in the earlier work on extensible languages.

(ii) Alphard enables iterations to be defined over the new type. For example, if the type is a list, we can define a construct for iterating over its elements. As usual, the abstract form of this construct is independent of its implementation.

(iii) Simula allows the initialization section to be extended into a coroutine. This turns the object into an ongoing process which can be suspended (by 'detach' or 'resume') and resumed (by another process doing 'resume') within a quasi-parallel system. This has particular relevance to simulations.

(iv) In Concurrent Pascal, a limitation is put on the monitor procedures that only one of them may be active at any one time. For example, we cannot have one process doing 'S.pop' while another is doing 'S.push': their mutual exclusion is guaranteed by the system. The monitor also has a queueing facility for delaying a procedure call; for example, if S is empty, a call of 'S.top' can be delayed until after a 'push'. These facilities are geared for use in operating systems.

These last two examples show that a data type need not be thought of as a purely passive structure whose operations are completely controlled by the outside world. On the contrary, its objects may have a life of their own and may control the way in which they are operated on. Different applications require different facilities in this respect. But the choice of the most appropriate ones is still a matter of great debate.

5. PROTECTION

As remarked earlier, if the integrity (or "consistency") of a defined type is to be guaranteed, its representation must not be accessible except through the operations provided. In other words, it must be protected from unauthorized access and, in particular, its fields must be invisible to the environment.

As an example we may take the Simula classes head and link. These are predefined for the user in order to give him facilities for storing a set of objects in a two-way list. Each object is given pointers Succ and Prede for the purpose, and there are standard operations such as 'out', 'into', 'follow' and 'precede' which make use of them. The programmer can obtain copies of these pointers by means of the functions 'suc' and 'pred'. However, the pointers

themselves are invisible; that is to say, the programmer cannot refer to them by their names. They are thereby protected from injudicious manipulation.

On the other hand, no such protection is afforded when the programmer himself defines a class. For example, with the symbol stack S defined above, there is nothing to stop someone from writing an assignment

```
S . p := ...
```

thereby setting the stack pointer of S to any value he likes. He may claim that there are times when this is very useful. For instance, he may argue that it is more efficient to empty the stack by the assignment

```
S . p := 0
```

than by the loop

<u>while</u> <u>not</u> S.empty <u>do</u> S.pop

But to this we reply that it is better still to define an operation 'clear' for the purpose. Allowing people to change p may have advantages on the small scale, but it is courting trouble on the large. For this reason, Simula on the PDP-10 has facilities for limiting the scope of an attribute. The specification PROTECTED makes it invisible to the general environment, and the specification HIDDEN makes it invisible to modules which the class prefixes (see below). Thus the combination HIDDEN PROTECTED limits its visibility strictly to the class that it belongs to [16].

In contrast to standard Simula, languages such as Clu and Model make all attributes totally invisible. They are local to the definition and cannot be referred to from outside. This provides the full protection which the ideals of data abstraction require.

Koster [12] has argued that visibility and (conversely) protection are the key issues in data abstraction. After all, what is new about defining data structures and procedures? We can define symbol stacks in Fortran, Algol or any other language, along with their associated procedures. The ability to wrap them all up in a single package called a class or a cluster is nice, but it is not essential. So the only novel feature that remains is the protection that may (or may not) be provided. And here, he argues, nearly all languages are seriously deficient. They do not offer protection facilities except in a very limited, inflexible way. The real need is to have generalized facilities for controlling the visibility of types, variables, procedures, etc. between program modules. This would offer many benefits, the ability to do data abstraction being just one of them. He has developed the language Slan based on these principles.

The language Modula illustrates the same philosophy (Wirth, [19], p.6):

> A module is a collection of constant-, type-, variable-, and procedure declarations. They are called <u>objects</u>

> of the module and come into existence when control enters the procedure to which the module is local. The module should be thought of as a fence around its objects. The essential property of the module construct is that it allows the precise determination of this fence's transparency. In its heading, a module contains two list of identifiers. The _define-list_ mentions all module objects that are to be accessible (visible) outside the module. The _use-list_ mentions all objects declared outside the module that are to be visible inside. This facility provides an effective means to make available selectively those objects that represent an intended abstraction, and to hide those objects that are considered as details of implementation. A module encapsulates those parts that are non-essential to the remainder of a program, or that are even to be protected from inadvertent access.

With these facilities, an abstract type can be defined by a module. The define-list will contain the name of the type and its associated procedures, but the structural details will be kept hidden. Thus simple data abstraction is a by-product of the more general notion of defining modules with carefully controlled interfaces.

6. CORRECTNESS

A type module is _correct_ if it satisfies certain conditions required of it. As we have seen, a strict proof of its correctness requires some sort of _Principia_ which gives a formal statement of these conditions by means of definitions and axioms. This axiomatic method is outlined by Hoare [10] and has since been developed by Standish [17] and several others. Liskov and Zilles [14] give a review of the various techniques, showing that there is still much to be done before satisfactory proof procedures are achieved. The following is their example of an algebraic specification for integer stacks:

Functionality :

```
CREATE :                     → STACK

PUSH   : STACK x INTEGER → STACK

TOP    :             STACK → INTEGER ∪ INTEGERERROR

POP    :             STACK → STACK ∪ STACKERROR
```

Axioms :

```
1.  TOP (PUSH (S, I)) = I

2.  TOP (CREATE) = INTEGERERROR

3.  POP (PUSH (S, I)) = S

4.  POP (CREATE) = STACKERROR
```

The first part defines the domain and range of the abstract operations and the second defines the relations which the operations must satisfy. Among the expressions generated by the algebra are those whose values are of the defined type (in this case STACK). We may say that these represent objects of the defined type, the understanding being that expressions which are equivalent under the axioms represent the same object and those which are not equivalent represent different ones. A primitive object is one that can be constructed without using other objects of the defined type (the only example here being the result of CREATE).

With this approach, we may say that an implementation is correct if "it defines an isomorphic image of the presented algebra". In the simplest case, this means that there is a one-to-one correspondence between abstract objects and concrete states. The concrete operations must then satisfy the corresponding functionality and axioms. In practice, though, this may need modifying in two ways. Firstly, not all the concrete states need represent an object (for example, those with a negative stack pointer). Those that do are distinguished by satisfying some condition, or set of conditions, called an invariant. The functionality conditions must therefore take the concrete type to be limited by this invariant. Secondly, some objects may have several possible representations. For example, a set of integers can be stored in many different sequences. The various possibilities are said to be equivalent, and the '=' in the axioms, when referring to concrete states, must be interpreted in this sense.

With this approach the correctness proof for a data abstraction involves four stages:

(i) establishing distinct representations for the primitive objects;

(ii) stating the invariant (if any);

(iii) defining the equivalence relation (if the mapping is not one-one);

(iv) showing that the axioms are satisfied.

Note that it is not necessary to do (i) for all the objects, since a correct representation of the other objects will follow by induction. Despite this, to give a complete, formal proof is a laborious matter for even the simplest cases. So most proofs are more or less informal.

The above algebra is in many ways the most natural formalization of the abstraction, but even so it has several drawbacks:

(i) The axioms assume that PUSH is a function with a stack result, rather than a routine with a side effect.

(ii) The technique of handling errors does not necessarily reflect our real intention. In

some contexts, it is good practice to return an error result, but in others we may prefer to trap the condition.

(iii) Once again, we need to put some limit on the size of the stack. This sounds simple, but in fact it more than doubles the length of the specification. This is a high price to pay for something that adds nothing significant to the concept of a stack.

One technique that comes in useful at this point is to restrict the sequence of operations on an object by means of a path expression [4]. For our stack, the relevant restriction is

$$0 \leq \#(\text{push}) - \#(\text{pop}) \leq \text{size}$$

which is formally written as the path expression

$$\underline{\text{path}}\ (\text{push} - \text{pop})^{\text{size}}\ \underline{\text{end}}$$

If this restriction is included as part of the type definition, the compiler can generate code to ensure that it is always satisfied. It will then trap both overflow and underflow of the stack. It may be remarked that we can make it trap these conditions by simpler means. In Pascal, for example, we could limit p to the subrange 0..size. But this, of course is part of the implementation. In contrast, the path expression is part of the abstraction and therefore superior as a specification technique.

Path expressions can be applied to a variety of problems, including those of concurrency [7]. However, it is not clear what are the best primitives for them. Regular expressions are inadequate even for the above example, so some extensions are obviously needed. Habermann [9] gives some possibilities, but unfortunately they make use of the representation. An alternative version [5] is more promising in this respect and has been formalized in terms of Petri Nets. But recent work suggests that these are inadequate even for the simple readers and writers problem [8]. So there is scope here for further improvement and originality.

7. TYPE PARAMETERS

A notable weakness of our definition of symbolstack is that it is tied to the type char. If, as in the previous section, we required an integer stack, a complete new class would have to be defined. Yet the concept of a stack is clearly independent of the type of its elements and we would like to be able to define it in that way.

To be fair to Simula it should be pointed out that in some cases this strategy is possible. For example, the classes head and link which, as already mentioned, implement a two-way list structure, are independent of the type of the list elements. If we wish to define a class barber, say, whose objects can be stored in a two-way list, we define it

with the class link as a prefix:

```
link class barber ...
```

This provides all barbers with the links Succ and Prede, and allows us to apply the operations 'out', 'into', etc. to them.

This technique has two drawbacks. The first is that, although we may start a queue of barbers by declaring

```
ref (head) barberqueue
```

there is no guarantee that only barbers will ever join it. We may have another class

```
link class butcher ...
```

and it is quite possible for butchers to join the barber queue as well. So, if we take the first member of the queue and expect him to perform a haircut, we have to check that he is a barber rather than a butcher. This is done be referring to him as

```
barberqueue.first qua barber
```

the effect of the qualification 'qua barber' being to insert a run-time check on his type.

The second weakness is that the strategy does not work for stacks and other non-linked structures. What we need for this is the ability to specify the type of the elements as a parameter of the stack definition. An obvious notation would be the following:

```
class stack (t, size) ; type t ; integer size ;
```

We could then create objects such as

```
S1 :- new stack (char, 30) ;

S2 :- new stack (integer, 100) ;
```

according to taste. However, this would once again land us in type-checking problems, since it is not clear what should be the types of S1 and S2. The type ref (stack) would raise the same problems as the type ref (head).

Clu and Model both allow types as parameters, so it is interesting to see how they handle this problem. In Clu, the stack cluster would begin

```
stack : cluster (t : type, size : integer) ...
```

and stacks could then be created by suitable declarations:

```
S1 : stack (char, 30) ;

S2 : stack (integer, 100) ;
```

Both the element-type and the size are treated as part of the type. Hence, if a stack is passed as a parameter to a procedure, the formal type must be identical. The only exception, of course, is when the procedure is one of the operations defined for that type. The philosophy of Model is much the same, except that the parameters of a type definition _must_ be types. Hence _size_, for example, would have to be subrange such as [1...30] or [1...100]. This can be a rather annoying restriction.

If a type definition is to be pre-compiled, independently of any environment, the presence of type parameters will call for considerable type-checking at run-time. To avoid this we must (a) abandon total pre-compilation and (b) ensure that the values of all actual parameters of type definitions are known at compile-time. This means, for example, that the size of each stack must be known at compile-time. This is similar to Pascal's philosophy for arrays. To recover the more flexible approach of Algol, we would have to treat type parameters differently from non-type parameters.

Assuming, then, that the types are fully determined at compile-time, the simplest thing for the compiler to do is to compile the type definition separately for each different type parameter that is used. This may be a small price to pay for maximum run-time efficiency. However, it raises a further problem concerning the operators used inside the definition. As an example, suppose that we wanted to define a procedure 'plus' which would add the top two elements of a stack. This would involve the use of '+' at some point, which might compile as one thing for _integer_ elements and another for _real_. But if we tried compiling it for _char_, it would cause a compile-time report, even though the program might never use it on type _char_. This is so unacceptable that Model defers the type-checking until the first call of the operator at run-time using that particular type. An equivalent, and rather simpler strategy, would be to compile '+' on type _char_ so that it would produce a run-time report. Hopefully the code would never then be executed.

Thus type parameters produce some interesting problems for the compiler writer. Organick [15] has made the further comment that, when used at all extensively, they make programs difficult to understand. Be that as it may, the facility appears to be based on sound principles. So maybe the challenge is to find better ways of expressing and implementing it.

8. HIERARCHIES

The abstraction of a type _stack_ is all very well, but it does not get us very far along the road towards producing large and useful programs. However, if we view the final program itself as an abstraction, we see that the overall technique is of great potential. Using the hierarchic approach, we decompose the main program into several smaller operations, each of which in turn is decomposed further, and so on. Thus we have a series of levels, each level defining its abstractions in terms of the level below it.

Simula provides two facilities for doing precisely this - the prefixed class and the prefixed block. The prefixed class allows the definition of a class C, say, to be prefixed by a previously defined class, P say. The objects of class C are then automatically endowed with all the attributes and operations of P. We saw an example of this in the definition

```
link class barber ...
```

which gave a barber the attributes necessary for belonging to a two-way list. The prefixed block allows us to define a class which is itself a collection of structures and classes, and then to make this whole collection available to a block. For example, Simula defines both head and link in terms of the class linkage, and then combines them into a single class simset:

```
class simset ;

  begin class linkage ... ;

        linkage class head ... ;

        linkage class link ... ;

  end ;
```

We can now use this class to prefix a block, thus making the facilities of head and link available in that block:

```
simset begin

         ref (head) barberqueue ... ;

         link class barber ... ;

         ...

       end ;
```

This in turn is used in the definition of the class simulation, which is basically a set of procedures for scheduling, activating, suspending and terminating processes:

```
simset class simulation ;

         begin link class process ... ;

               link class eventnotice ... ;

               ref (head) sequencing set ... ;
               ....
         end ;
```

To set up a simulation of, say, a barbers shop, we can now use simulation as a prefix:

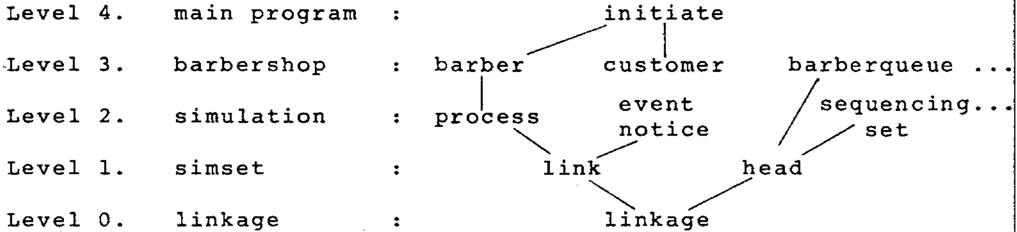
```
simulation class barbershop (nbarbs) ; integer nbarbs ;

            begin .... end ;
```

Finally, using the prefixed block, we can make the facilities of a particular barber's shop available to the main program:

barbershop (5) <u>begin</u> .... <u>end</u> ;

Thus we have the following hierarchy:



The book "Simula Begin" [2] contains some beautiful examples of other programs structured in this way.

Despite these attractions, Simula still leaves room for criticism. For one thing the concept of a prefixed block is rather confusing. For another, as noted earlier, the protection facilities are inadequate. And thirdly, it only provides a "weak" hierarchy; that is to say, the operations at one level are available at all higher levels and cannot be restricted to just the next level. Nevertheless, the basic structuring concepts are very appealing, and it is hard to believe that they were all being offered by Simula more than ten years ago!

By contrast, most other languages are disappointing in this respect. Clu and Model offer nothing beyond normal block structuring. Slan and Modula, as we have seen, are more flexible in their mechanisms and avoid some of the deficiencies of Simula. But none of them have anything to match the prefixing concept in terms of its brevity and power.

9. CONCLUSION

In conclusion, it is as well to summarize the potential benefits of good abstraction facilities. Basically, data abstraction can do for data what procedures did for programs. It provides clearly defined modules which encapsulate an essential feature of what is going on. These modules can be units for

- program design and development
- specification and correctness proofs
- optimization and maintenance
- separate compilation.

In these days it is not necessary to extol the merits of such objectives. The only need is to work towards achieving them. We may then look forward to the day when system libraries hold

collections of such modules, readily available to the programmer; when the abstract type will be offered with a choice of representations; when the systems itself, maybe, will choose the one most suited to a particular application; when it may even change to another one during execution, on the basis of experience.

This is the long-term prospect. In the short term we may expect journals to publish type definitions in the same way that they publish algorithms. Given a suitable language, it is an obvious thing to do. Surprisingly, though, no journal has yet done it. If the Australian Computer Journal needs more material, maybe it should take the opportunity to pioneer this enterprise. Meanwhile, as already indicated, the field of data abstraction still offers great scope for continuing development and research.

REFERENCES

Note: the reference DADS refers to the Proceedings of the ACM SIGPLAN/SIGMOD Conference on Data Abstraction, Definition and Structure, Salt Lake City, published in SIGPLAN Notices, Vol. 11, Special issue, 1976.

[1] D.M. Berry, Z. Erlich and C.J. Lucena, Correctness of Data Representation: Pointers in High Level Languages, DADS, 115-119, 1976.

[2] G.M. Birtwhistle, O-J. Dahl, B. Myhrhaug and K. Nygaard, Simula Begin, Auerbach, Philadelphia, 1973.

[3] P. Brinch Hansen, The Programming Language Concurrent Pascal, IEEE Trans: Software Engineering, Vol. 1, No. 2, 1975.

[4] R.H. Campbell and A.N. Habermann, The Specification of Process Synchronization by Path Expressions, Lecture Notes in Computer Science, Vol. 16, Springer Verlag, Berlin, 1974.

[5] R.H. Campbell and P.E. Lauer, Formal Semantics of a Class of High-level Primitives for Coordinating Concurrent Processes, Acta Informatica, Vol. 5, 297-332, 1975.

[6] O-J. Dahl, B. Myhrhaug and K. Nygaard, The Simula 67 Common Base Language, Norwegian Computing Centre, Oslo, 1970.

[7] L. Flon and A.N. Habermann, Towards the Construction of Verifiable Software Systems, DADS, 141-148, 1976.

[8] A.J. Gerber and C. Morgan, Private communication, Department of Computer Science, University of Sydney, 1977.

[9] A.N. Habermann, Path Expressions, Department of Computer Science, Carnegie-Mellon University, Pittsburg, 1975.

[10] C.A.R. Hoare, Proof of Correctness of Data Representations, Acta Informatica, Vol. 1, 271-281, 1972.

[11] R.T. Johnson and J.B. Morris, Abstract Data Types in the MODEL Programming Language, DADS, 36-46, 1976.

[12] C.H.A. Koster, Visibility and Types, DADS, 179-190, 1976.

[13] B.H. Liskov and S. Zilles, Programming with Abstract Data Types, Proceedings of SIGPLAN Symposium on Very High Level Languages, Santa Monica, published in SIGPLAN Notices, Vol. 9, No. 4, 50-59, 1974.

[14] B.H. Liskov and S. Zilles, Specification Techniques for Data Abstractions, Proceedings of International Conference on Reliable Software, Los Angeles, published in SIGPLAN Notices, Vol. 10, No. 6, 72-87, 1975.

[15] E.L. Organick, Comment in Discussion Session from the NCC, quoted in SIGPLAN Notices, Vol. 10, No. 7, p.31, 1975.

[16] J. Palme, New Feature for Module Protection in Simula, SIGPLAN Notices, Vol. 11, No. 5, 59-62, 1976.

[17] T. Standish, Data Structures - An Axiomatic Approach, Bolt, Beranek and Newman, Cambridge, Mass., 1975.

[18] A.N. Whitehead and B. Russell, Principia Mathematica, Cambridge, 1910.

[19] N. Wirth, MODULA : A Language for Modular Multiprogramming, Institut fur Informatik, ETH, Zurich, 1976.

[20] W.A. Wulf, ALPHARD : Toward a Language to Support Structured Programs, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa, 1974.

8

A Review of Proposals for Introducing Dynamic Arrays into Pascal

Barbara P. Kidman

ABSTRACT

Proposals for introducing arrays with dynamic bounds into Pascal are reviewed and the implications discussed. It is concluded that the simple extension designed in the Zurich school, which allows dynamic array parameters, would considerably enhance the usefulness of the language.

1. INTRODUCTION

Pascal has been criticised often [2,4,7,8,9,11,12,14] for the lack of "dynamic arrays", and a number of remedies have been suggested. It is the purpose of this report to describe and discuss suggested changes to Pascal in relation to their effect on the language, its implementation and application.

The phrase 'dynamic arrays' is customarily used to describe arrays whose bounds change in some way during execution even though this can occur in rather different contexts. (Compare [4] and [8].) In Algol 60, for example, the bounds of an array variable declared at a block head are expressions evaluated on block entry. Even with constant bounds in the above context, formal arrays are considered to have dynamic bounds if differently bounded actual parameter arrays are acceptable. The terms 'static variable' and 'dynamic variable' are also applied [6,10] but in a different sense.

All array bounds in Pascal are constants. Four different contexts can be distinguished in which arrays might have dynamically determined bounds:

(a) Array parameters

(b) Explicitly declared array variables

(c) Anonymous array variables

(d) Array variables with bounds flexible in the Algol 68 sense.

Fortran allows adjustable or dynamic bounds in (a), Algol 60 has dynamic bounds in (a) and (b), and Algol 68 in all of these contexts.

To clarify terminology and as a background for later discussion a brief overview of pertinent characteristics of Pascal is given.

2. ARRAYS IN PASCAL

2.1 Explicitly declared array variables

Pascal is a block structured language in exactly the same sense as Algol 60 except that blocks exist only at procedure level. Declared variables denoted by identifiers come into existence on procedure entry and cease to exist on procedure exit. Although the manual [6] terms such variables static, they should not, of course, be confused with variables in other programming languages which are allocated to static storage areas. A simple dynamic storage allocation scheme, using the well known run time stack is used to implement the block structure of Pascal and Algol 60 [3,13], a data area for local data being pushed on to the stack on block activation, and popped on block exit. In Pascal, as in Algol 60, arrays declared locally in parallel procedures will be overlaid in memory. However, a fundamental difference between the runtime systems in the two languages results from the restriction of array bounds to constants in Pascal.

If memory address is expressed as the pair

(base address of data area, offset)

then in Pascal, the offset, but not the base address, is static, whereas in Algol 60 the offset may have to be evaluated at run time. Partly as a consequence, run time stack management can be more efficient in Pascal than in Algol 60; on the CDC 6400, the execution of calls to a dummy recursive procedure with one value parameter is almost twenty times faster in Pascal than in Algol 60.

2.2 Anonymous array variables

The term 'dynamic variable' [6,10] describes indirectly accessed unnamed variables created at any stage of program execution - in contrast to explicitly declared variables where existence is related to the activation of the block in which they are declared. In Pascal such anonymous variables, as they are termed here, are accessed via pointers and created dynamically by calls to the procedure new, with their storage being allocated on the heap. There is some provision for dynamic determination of the amount of storage allocated when creating anonymous variables of type record.

2.3 The concept of type

A data type in Pascal essentially defines the set of values which may be assumed by a variable of that type. Assignment of a value must be to a variable of the same type or subrange thereof, and formal and actual parameter types must correspond. With the exception of files, and the records mentioned in 2.2 above, data values of one type occupy a fixed number of storage locations. Hence, in most cases, static type checking ensures the validity of the well defined operations which are applicable to values of each type.

The concept of array type in Pascal encompasses the bounds, and so array variables are of the same type only if their

dimensionality, bounds and component types are identical. Record structures may have such arrays as fields, and assignments may be made to entire arrays and records. For example, following the declaration

```
var A,B : array [1..10,1..10] of real;
```

the validity of the assignment A:=B

can be largely assured by type checking at translation time; this also applies when A and B are formal.

Any extension to the language should preserve as far as possible the simplicity and generality of this approach and the consequent security given by successful compilation. There is already a potential insecurity in assignment to anonymous record variables with a variant part when such variables are created by calls to new which include the tag as parameter, thus dynamically controlling the case of the variant and the size of the record, for example, assignment to p↑ after the statement:

```
new (p,tag).
```

While the Pascal manual [6] states that such assignments are not allowed, enforcement would be difficult to implement.

In Algol 68, on the other hand, the mode of an array does not include bounds. Static mode checking then does little to assure security in the case of assignment to entire array or record variables. An additional semantic rule (see[16] 5.2.1.2) and a corresponding run time check are required to ensure matching of the bounds of array values assigned to array variables.

2.4 Advantages and disadvantages in static bounds for array parameters

Type constraints in Pascal force actual parameter arrays to have precisely the same bounds as the corresponding formal parameters. Thus it is impossible to write, for example, a general purpose matrix transpose procedure suitable for a source code library; in the one program multiple copies of such a procedure are required to handle arrays of different sizes.

However, it is very simple to simultaneously adjust array bounds and loop limits in Pascal source code by resetting constant identifiers. Also a reasonable degree of generality can be obtained by declaring array bounds larger than required so that data is stored in part of the array. Working with such subarrays is natural and straightforward in Pascal, but cumbersome and error prone in Fortran. In Pascal with static array bounds no run time descriptors are necessary, and type checking ensures that array mapping functions are consistently applied to actual parameter arrays. In Fortran, consistency in this regard is dependent on programmer discipline and understanding, which is a major difficulty in learning the language as well as a common source of errors.

In implementations, such as that on the CDC 6400, which allow linking to external procedures, it is possible, though far from satisfactory, to access specially coded but independently compiled procedures to which array bounds are passed as parameters as in Fortran. With this approach the security of parameter type checking and subscript bounds checking must be foregone.

3. PROPOSED EXTENSIONS TO ARRAY PARAMETERS

3.1 Type identifiers with parameters

Wirth [14] is assessing the language Pascal hinted at a solution to the array parameter problem without giving a detailed description. The proposal involves a restricted form of type parameterization illustrated in the example below, which is taken from [14].

```
type table(m,n)=array [m..n] of integer;

var tl:table (1,100);  t2:table (0,999);

function sum(t:table;u,v.:integer);
```

The actual parameters in the type of the actual array parameter are restricted to constants as shown. Subsequent calls made to sum would take the forms:

```
sl:=sum(tl,1,100);

s2:=sum(t2,0,999);
```

Presumably it was intended that this facility be applicable to multi-dimensional arrays, although further generalisation was specifically not recommended [14].

3.2 Type and constant parameters

Tennent [12] has described features of a Pascal-like language, Pasqual, which among design aims, purports to solve the array parameter problem, but again does not allow arrays with dynamic bounds in other contexts. Several kinds of parameter are introduced, including _type_ parameters, as illustrated in the following example:

```
procedure transpose(type ind; var t:array[ind,ind] of real);
```

Given the declaration

```
A : array [1..10,1..10] of real
```

then the procedure can be called as transpose(1..10, A). The actual type parameter 1..10 is bound to the formal ind at compile time, when the substitution in the second parameter is made.

Types are classified according to the operations which can be performed on them, partly to ensure that the actual parameter type corresponding to ind is an appropriate index

type. Consequently index is substituted for type in the formal parameter list:

```
procedure transpose (index ind; var t:array [ind,ind] of real);
```

Other more complex features of Pasqual are not relevant to the present discussion. However, another approach described to the array parameter problem is illustrated in the following example:

```
procedure transpose (const m,n:integer;
                     var t:array[m..n,m..n] of real);
```

The actual parameters corresponding to const parameters must be integer expressions computable and again substituted in the second parameter at compile time. Tennent suggests duplication of the procedure code for each call of such procedures with distinctive values of m and n; in other words the compiler would merely do what the Pascal programmer must now do. It is necessary to impose restrictions on const parameters for recursive calls and elsewhere.

3.3 Formal types

The notion of formal types has been proposed by Pokrovsky [9] as a means of 'uniformly introducing arrays with dynamic bounds' into Pascal. A formal type is a type pattern showing the dimensionality or shape of the array but not its bounds, much like the mode concept of Algol 68. Pokrovsky gives a formal syntactic definition of his proposed extensions in BNF, but here an example is used to indicate the basic syntax:

```
type matrix = array [*integer,*integer] of real;

var A : matrix [1..10,1..10];

procedure transpose (var t:matrix);

{ top and bottom functions available }
```

In declarations using formal types, termed instantiations, the same type or a subrange thereof is substituted for the formal indextype. * is suggested as an abbreviation for *integer. The formal type notion is also used to introduce anonymous variables with dynamic bounds as described below, but fixed bounds are retained for explicitly declared arrays.

3.4 Variable aggregates

A second array like entity called an aggregate is suggested in a note by Steensgaard-Madsen [11] with declarations presumably as follows:

```
var A [m..n,m..n]: real;

procedure transpose (var t [integer, integer]: real);
```

The example in [11] has an expression as a bound in the first

declaration, but subsequently the bound pair was described as an indextype. Both variable aggregates and array type variables can be passed as actual parameters. Further details are not given.

3.5 Dynamic array parameters

Jacobi's proposed extension [5] has been published in the form of updates to the Pascal manual [6], and its implementation is reported to be in progress [1]. Essentially only one new construct is introduced, the dynamic array type, which has the form of a Pascal array declaration with the bounds positions occupied by type identifiers, for example:

```
var A:array [1..10,1..10] of real;

procedure transpose (var t:array [integer, integer] of real);

{functions low and high yield bounds}
```

The subranges used as indices in the type of the actual parameter are substituted for those in the dynamic array type of the formal parameter. After the declaration above, a call to the procedure has the usual form:

```
transpose(A).
```

The introduction of dynamic array parameters makes necessary some additional restrictions in Pascal. For example, assignment to or from dynamic array parameters is not allowed, and the relational operators cannot be applied to dynamic array parameters which are of the type packed array of char. Further possible restrictions are packing in the last dimension only of dynamic array parameters and disallowance of dynamic arrays as parameters to formal procedures.

4. PROPOSED EXTENSIONS TO ANONYMOUS ARRAY TYPES

4.1 Formal types

Formal types are also applied by Pokrovsky [9] to anonymous array variables. Adding the following declarations to the example in 3.3 above

```
type refmatrix = ↑matrix;

var AA:refmatrix;
```

allows the creation of an array with dynamic bounds by calling the extended procedure new with an additional parameter, whose bounds are dynamically evaluated expressions:

```
new(AA, [1..n+1,1..n+1])
```

Given the previous definition, the procedure transpose may be invoked either with AA↑ as an argument or with an explicitly declared array (with fixed bounds) of type matrix as argument (see 3.3):

```
transpose(AA↑);    transpose(A);
```

4.2 Extension of the record variant

MacLennan [8] points out that, given the declarations

```
type vrec = record
        case j:1..3 of
        1:(A1:array[1..1] of char);
        2:(A2:array[1..2] of char);
        3:(A3:array[1..3] of char)
        end;
        var p:↑vrec;
```

calls such as new(p, n) can be used to create a record variable whose variant field consists of an array with upper bound n ($1 \leqslant n \leqslant 3$). On this basis he suggests a more general form of the record variant like

```
    type string = record
                        case lngth:0..1000 of
                         varying S:array[1..1000] of char
                        end;
    var p:↑string;
```

A call such as new(p, 25) will then create a variable of type string, with a tag of value 25 and with variant field S an array of bounds 1..25.

4.3 The dynamic row

As a response to [8], Wirth [15] has suggested an extension which effectively allows one index of an array-like structure to be dynamic. A new construct is introduced in conjunction with the definition of pointer types only, as illustrated in the following example:

```
var T : ↑row of T0;
```

An extension to new will allow dynamic specification of the length n in the usual way when a variable of this type is created:

```
new (T,n)
```

A length function will be available to return the length of the row, and a row can be indexed, for example:

```
T↑[i]  with  1 ≤ i ≤ n,
```

Conradi [2] in comparing Pascal and MARY also suggests

the introduction of a structure somewhat resembling the row without giving a specific indication of how the construct could be accommodated in Pascal.

5. DISCUSSION

5.1 Array parameters with dynamic bounds

Clearly all of the proposals for introducing dynamic bounds into Pascal array parameters have much in common. The ideas from Tennent's paper [12] are interesting, but, as he recognizes, go beyond what can be regarded as an extension to Pascal.

The syntactic constructs proposed first by Wirth [14]

```
type table(m,n) = array [m..n] of integer
```

and by Pokrovsky [9]

```
type table = array [*] of integer
```

both effectively introduce parameterized type identifiers, but restrict the parameterization to array bounds. This device ensures technical type matching between formal and actual array parameter types. A somewhat similar notion, in less elegant form, also underlies the aggregated variables suggested by Steensgard-Madsen [11].

On the other hand, in Pasqual [12] and in Jacobi's extension [5], the "parameterized index type" occurs only as a type pattern in the formal parameter list itself, thus avoiding the introduction of a new syntactic form for type identifiers.

Pokrovsky's understanding [9] (based on at least one implementation) that the definitions below define three different types is at variance with the action of the current CDC 6400 compiler.

```
type T1 = array [1..10,1..10] of real;

     T2 = array [1..10,1..10] of real;

var V1:T1; V2:T2;

     V3:array [1..10,1..10] of real;
```

Using the definition of the formal type matrix given in 3.3, then Pokrovsky appears to interpret the following as declaring two arrays of different type,

```
A : matrix [1..10,1..10];

B : array [1..10,1..10] of real;
```

and hence the assignment A:=B

would fail to pass compilation type checking. On the other hand with

```
        C : matrix [1..20,1..20];
```

the following assignment would be trivially legal A:=C.

Jacobi's extension [5], in omitting the explicitly parameterized type identifiers of the earlier proposal [14], avoids the above anomaly. However, despite type checking, assignment to or from array parameter variables with dynamic bounds is a dangerous operation unless array bounds are subjected to a run time checking. Such assignments are disallowed by Jacobi.

From the above discussion it appears that clarification is needed on the type equivalence relation in Pascal. With the declarations of variables V1, V2, and V3 given above, the current CDC 6400 compiler allows any of these variables to be actual parameters corresponding to a formal parameter specified as V:T1. Presumably the three variables would also be acceptable actual parameters corresponding to the formal dynamic array parameter V:<u>array</u> [integer, integer] <u>of</u> real.

Jacobi's restriction of disallowing use of relational operators with any dynamic array parameters appears reasonable. The inclusion of functions like low and high as in [5] which yield dynamic array parameter bounds is essential; a second parameter, as in [1] and [9], will be required to indicate dimension.

It is to be particularly noted that type checking in the schemes under discussion ensures that the correct actual parameter array bounds information is passed to the procedure, and hence the array mapping used by the procedure will be consistent with that in the calling program. In other words the difficulties encountered with Fortran do not occur.

Some comments in the implementation of the proposed extensions follow. Tennents' suggestion [10] of duplicating the procedure code for all distinct bound values of the actual parameters offers only minor advantage to the programmer, and in effect would serve only to hide the real array parameter problem from the programmer.

If array bounds are dynamic it is necessary to maintain a run time array descriptor such as a dope vector in which is stored the bounds information needed for the array mapping function and for bounds checking. This applies therefore, to any form of array parameter with dynamic bounds. The bounds information can be entered in the descriptor by code generated for each textual procedure call.

For call by value dynamic array parameters, the size of the local data block into which the actual parameter array is copied and hence its relative address (offset) will both be dynamic. This must cause some complication in stack management.

5.2 <u>Explicitly declared arrays with dynamic bounds</u>

To allow dynamic bounds in declared arrays requires a

simple and obvious extension to the syntax, of course, but the effects on language concepts, on efficiency and on implementation would be considerable. The absence of proposed extensions along these lines, other than the criticism of Habermann [4], suggests general agreement that the disadvantages outweigh the gain [7,14].

5.3 Anonymous array variables with dynamic bounds

The comments in 5.1 above relating to type checking and to the implementation of array parameters with dynamic bounds also apply to anonymous array variables with dynamic bounds. The consequent loss of security from simple static type checking imply still further language "exceptions" such as restrictions on the use of operators, or alternatively run time checking.

With regard to the implementation of Pokrovsky's formal types, note that a run time descriptor will be required for the array AA↑ of type matrix (see 3.2 and 4.1 above) but not for the array A which is also of type matrix. Moreover, in contrast to the case for array parameters, the bounds information to be maintained in the descriptor is itself dynamic. Assessment of practicality without some attempt at implementation is difficult.

The 'compromise' suggested by Wirth [15], namely an indirectly accessed dynamically bounded <u>row</u>, avoids much of the complication. Disallowance of assignment to entire <u>row</u> variables is necessary for reasons already discussed. MacLennan's proposal [8] for introducing dynamic arrays via the record variant depends on knowledge of the implementation. On the other hand, the high level dynamic <u>row</u> facility is transparent to the programmer.

6. CONCLUSION

Pascal is a carefully designed, defined and implemented language, which has been extremely successful. Any extensions or changes to such a language deserve the same care.

There is general agreement that introducing array parameters with adjustable or dynamic bounds would considerably enhance the usefulness of Pascal. The elegant and simple extension proposed by Jacobi [6], from the Zurich school, appears to meet requirements admirably; implementation is reported to be in progress. However, as introduction of dynamic bounds in any context implies further restrictions or exceptions in the language as well as loss of efficiency, further extension along these lines is not attractive. Implementation of the high level dynamic <u>row</u> structure [15] may be appropriate in special circumstances.

REFERENCES

[1] U. Ammann. Letter published in Pascal Newsletter, No. 5, 27, 1976.

[2] R. Conradi, <u>Further critical comments on Pascal, particularly as a systems programming language</u>, SIGPLAN Notices, <u>11</u>, No. 11, 8-25.

[3] D. Gries, Compiler construction for digital computers, Wiley, New York, 1971.

[4] A.N. Habermann, Critical comments on the programming language Pascal, Acta Informatica, 3, 47-57, 1973.

[5] Ch. Jacobi, Dynamic array parameters, Pascal Newsletter, No. 5, 23-27, 1976.

[6] K. Jensen and N. Wirth, Pascal user manual and report, Springer-Verlag, New York, 1975.

[7] O. Lecarme and P. Desjardins, More comments on the programming language Pascal, Acta Informatica, 4, 231-243, 1975.

[8] B.J. MacLennan, A note on dynamic arrays in Pascal, SIGPLAN Notices, 10, No. 9, 39-40, 1975.

[9] S. Pokrovsky, Formal types and their application to dynamic arrays in Pascal, SIGPLAN Notices, 11, No. 10, 36-42, 1976.

[10] M. Shave, Data structures, McGraw-Hill, London, 1975.

[11] J. Steensgaard-Madsen, More on dynamic arrays in Pascal, SIGPLAN Notices, 11, No. 5, 63-64, 1976.

[12] R.D. Tennent, Parameters, declarations and parameterized declarations in a Pascal-like language, Personal communication, 1976.

[13] N. Wirth, The design of a Pascal compiler, Software-Practice and Experience, 1, 309-333, 1971.

[14] N. Wirth, An assessment of the programming language Pascal, Proc. Intl. Conf. on Reliable Software in SIGPLAN Notices, 10, No. 6, 23-30, 1975.

[15] N. Wirth, Comment on a note on dynamic arrays in Pascal, SIGPLAN Notices, 11, No. 1, 37-38, 1976.

[16] A. Van Wijngaarden et al. Revised report on the algorithmic language Algol 68, Acta Informatica, 5, 1-236, 1975.

9

Operating System Command Languages – Some Recent Developments

Cliff B. Mason

ABSTRACT

An operating system command language is that part of a programming language system that allows a computer user to interface with an operating system and to control the execution of his job. A detailed history of the study and development of operating system command languages is given together with examples of some of the different research philosophies. The paper concludes with a resume of the International Federation for Information Processing Working Conference on Command Languages and prospects for future progress.

1. INTRODUCTION

To many computer users, the prospect of learning a job control language is more terrifying than learning a new programming language. The latter is generally related to a user's application, and through it a user expresses his ideas and the algorithmic form of his problem. There is little in a programming language that is unrelated to the solution of the problem at hand. Unfortunately, this cannot be said for many job control languages. In this paper the more generic term Operating System Command Language (OSCL) is used instead of job control language. As the term implies, an OSCL is generally a reflection of the underlying operating system.

We may ask the question - why should an applications programmer concern himself with the architecture of a computer, the structure of its operating system, and the flow of control within that system, when submitting his inventory program or engineering problem? After all, the computation could be equally well performed on any computer. The operating system is only a means to automate and streamline the flow of jobs through the computer, to simplify the life of a computer operator, and to schedule the allocation of hardware and software resources between competing jobs. None of these functions are of direct interest to the programmer, except for their effect on job turnaround, and they certainly bear no relationship to the problem he is solving. Yet each day we find applications programmers trying to fathom the mysteries of an OSCL and its interaction with the host system.

This aspect of man-machine communication must be considered when examining any programming language system. To a computer user, a Programming Language System can be no more and no less than that set of facilities which is essential to the successful solution of his problem. The facilities will include a suitable programming language, system software packages such as compilers and loaders, perhaps utilities and last but not least, the OSCL that alone will control the passage of a job through the computer. Until we learn to combine the functions

of these separate parts in a complete integrated system, we have a lot of work ahead of us and users will continue to waste millions of dollars each year on OSCL errors. It has been estimated that 1,450 million dollars was wasted in 1975 on command language errors [39] (p.390).

With figures of this magnitude and the countless frustrations each of us has encountered with OSCLs, few will argue against a massive effort to solve the problem. However, few will agree on the correct approach to the problem, a fact that is painfully obvious when looking at past and present research into OSCLs. For this reason, a fairly complete description of the history and study on OSCLs (Section 2) is given in the paper.

Then some of the different research philosophies are examined, highlighting them with examples of current research projects (Section 3). In conclusion in Section 4, the paper summarises the results of the International Federation for Information Processing (IFIP) Working Conference on Command Languages [39], when research workers had their first opportunity to debate the relative merits of their different research philosophies. Future opportunities for research, including areas so far neglected, are mentioned.

As one frustrated worker put it at a recent SHARE Command Languages meeting, "chaos again"; this paper obviously does not purport to solve all OSCL problems, but to draw attention to the many avenues of research and to aid anyone contemplating work in this field, particularly in Australia.

2. THE HISTORY AND STUDY OF OSCLs

Details of the evolution of OSCLs may be found in a paper by Barron and Jackson [3]. More recent information, particularly that concerning the activities of standardisation groups and professional societies, is taken in part from a paper by Enslow [17] and from personal correspondence with those concerned.

2.1 Early Developments

In the early 1950s, there were no assemblers or language translators; each program defined explicit machine operations and ran in dedicated mode. As software packages were developed to reduce the semi-clerical task of coding in machine language, the complexity of a given program run increased. This prompted the development in the early 1960s of "monitors" to automate the different phases of program execution and to ease inter-job transitions.

Typical early systems included the Fortran Monitor System (FMS), the Bell Laboratories system (BESYS) and the IBM 7090/7094 system (IBSYS). The user interface to these monitors contained simple control cards to identify the user and specify language translation requirements. Generally speaking, the control cards were scattered throughout the program deck and its data.

An early alternative to the control card approach was that

used in the Atlas paper tape system and, subsequently, in the ICL GEORGE 3 system. Here, all job description details were concentrated together at the front of the job, removing the need for meta symbols to distinguish control statements and simplifying the transition from batch to interactive operations [32].

When the IBM OS/360 Job Control Language (JCL) was introduced, the complexity of the user-interface increased dramatically, as did the number of problems and wasted computer runs. Users found that debugging the JCL statements was as large a task as debugging the program. For example, IBM's JCL reference manual consists of 361 pages of quite difficult reading; this compares with only 160 pages to describe the Fortran language - a sorry reflection on a control language that is meant to simplify the interface between a Fortran user and the operating system. The subject was important enough to become the sole topic of text books, e.g. [10] and [5].

The upsurge of interest in OSCLs prompted their organised study and at least five groups have been addressing themselves to the question of standardisation. Professional societies, user groups and substantial users have, from time to time, studied OSCLs and a summary of all this work is given below.

2.2 Standardisation Groups

(i) The American National Standards Institute (ANSI) set up the first group to formally study OSCLs. The initial proposal was presented in June, 1967 by Mr. Millard H. Perstein and an ad hoc committee was formed a year later under his chairmanship (USASI X3.4.2F, later to become ANSI/X3/SPARC/OSCL). The committee concentrated on the technical aspects of developing a standard OSCL with little thought about feasibility or economics. An extensive survey was conducted of nine operating systems and their control languages (GECOS III, EXEC VIII, Honeywell Mod 4 OS, MULTICS, PDP-10 Monitor, 360/OS(TOS), 360/DOS, 360/TSS and Sigma 5/7 BTM). Details of the survey were published in 1971 in a report to SPARC (Standards Planning and Requirements Committee) and provided the basis for a general classification applicable to the systems studies. "In general the surveys and (function) matrix pointed out that there are a finite number of functions for which an operating system can provide control regardless of its complexity. However, within the industry today there is an almost infinite variety of ways of combining and implementing these functions from a language standpoint." (OSCL 71 [26], para 2.5). Their recommendations were:

- The need for a standard is pressing and its attainment should be possible.

- Several features should not be included as part of the OSCL: editing, formating and control of output, and data base management.

- None of the OSCLs surveyed is a suitable candidate for the standard.

- There should be only a single standard OSCL.

- Piecemeal standardisation should be avoided.

The suggested program has made little progress since 1971 owing to the abovementioned deficiencies in the study and the loss of their Chairman in 1972. P.H. Enslow took over this role until his transfer to Europe and Dr. E.H. Sibley of the University of Maryland was appointed Chairman in 1974. Interested persons were invited to participate in the work of the committee [33]. A preliminary paper on the economic justification of a standard OSCL has since been published [34].

(ii) A Dutch JCL Committee under the Chairmanship of H.J. Weegenaar was established in September, 1971 with a brief "to research (the) basic functions of job control, required by the data processing user and expressed in a job control language." Existing systems were studied in the light of a pre-defined OS "function-matrix", however, the limitations of their function-matrix soon became obvious. They then devised a new framework of classification based on "binding classes of job control description":

- job structure binding;

- job resource binding, and

- job interface binding.

Basic job control functions were divided between the three binding classes and separate working groups established to look at each. Preliminary reports on their progress are available [40] and the Dutch group remains very active.

(iii) CODASYL: the organisation responsible for the preparation and maintenance of the Cobol standard, established an OSCL task group within its Systems Committee in 1973. The group's work was intended to be highly user-oriented, complementary to the ANSI effort, and directed towards the development of an OSCL that might be a candidate for standardisation. The latter objective was deferred in the group's December 1973 report [9] which listed its tasks as:

> "1. Investigate functional requirements for communication between the user, the functional programs and the hardware. In present day systems all such communication is channelled through the Operating System and the form of the communication is known as the Operating System Control (or Command) Language."
>
> "2. To determine the functions necessary to define a standard OSCL interface and what problems such a standard interface implies in the building of an operating system."

The report describes potential scenarios for the use of an OSCL and goes on to describe the development of operating system models. The first is a 'functional model' developed

as an analysis tool to determine operating system functions necessary to define a standard OSCL interface (task 2 above). The second model is 'hierarchic' and pictures an operating system as a set of interconnected code complexes driven by a high-level recursive language. The extent to which the OSCL is part of this language may be examined, i.e. the extent to which an OSCL is a specialised programming language. Much work remains, particularly with the hierarchic model, but 1974 plans called for validation of the models on existing and theoretical systems, development of a model of users, examination of the role of defaults and, ultimately, the commencement of work on a working OSCL.

(iv) In 1970/71, the U.S. Department of Defense commissioned Code Inc. to specify and evaluate a Standard Job Control Language (SJCL) and to draw up detailed user information about SJCL. Attention centred on the syntax definition of SJCL rather than the general functional requirements of an OSCL. Detailed reports were prepared [36] but SJCL was never implemented. Code Inc. is no longer in business and there seems little likelihood of further work on SJCL.

(v) The U.S. Federal Information Processing Standards (FIPS) people have kept a watching brief on the ANSI/SPARC/OSCL and CODASYL work and, if this does not lead to an early standards proposal, then the U.S. Government may well establish a task force to prepare such a standard, in a similar fashion to their action on the Cobol standard.

2.3 Professional Societies and User Groups

(i) The IEEE Computer Society and the Association for Computing Machinery (ACM) have not been actively involved in OSCL research and standardisation. The IEEE Technical Committee on Operating Systems (TCOS) undertook the development of a dictionary of operating systems terminology [16]. The ACM Special Interest Group on Operating Systems (SIGOPS) and on Programming Languages (SIGPLAN) held an 'Interface Meeting' in 1973 which was primarily concerned with the effects of the user-operating system interface on the ability to transfer programs. Little came out of the meeting that would affect OSCL design. Since then, the SIGOPS newsletter (Operating Systems Review) has provided a forum for articles on OSCL research.

(ii) In July 1974, the International Federation for Information Processing (IFIP), through its Technical Committee 2 (TC-2 Programming), sponsored a 'Working Conference on Command Languages' [39]. This successful meeting was the first significant contribution of IFIP to the OSCL scene and brought together for the first time a majority of the groups working on OSCLs. Further details of this conference are discussed below. One outcome of the meeting was a recommendation to establish an IFIP Working Group on Command Languages. A study committee was appointed to draw up terms of reference for the group. Their proposals, endorsed by TC-2, were formally approved by the IFIP General Assembly in late 1975. The group is known as WG2-7 (Command Languages) and the Chairman is Mr. F. Hertwick, Institute of Plasma Physics, Garching, W. Germany.

(iii) On 24th January, 1974, the British Computer Society (BCS) held a one day symposium entitled 'Job Control Language - Past, Present and Future' [35]. This meeting, while not being able to progress very far, was significant in being the first open meeting to devote itself entirely to OSCLs and generated considerable interest in the subject. As a result, the BCS Advanced Programming Group formed a specialist sub-group on JCLs which began work in March 1974. A possible program was drawn up (Appendix A) of which items 3, 5 and 6 were thought to be most useful for the group to explore, i.e. a 'common JCL' for existing systems, its spheres of influence and its form. Major consideration would be given to defining user requirements. A report on the first year's activity, published in the Computer Bulletin (June, 1975) dealt with their assessment of other work in the field and the development of a framework within which the orderly development of a machine independent JCL could proceed.

The BCS group is small but its members include most of the OSCL experts in the United Kingdom, i.e. at least those who are actively engaged in the development of machine-independent, high-level OSCLs for their organisations, e.g. [21-25],[11-14].

(iv) SHARE, the IBM scientific users' group, established a command language project in 1970 after hearing a paper on 'Alternatives to JCL' [4]. A preliminary position paper on command languages was published [37] which dealt, in very general terms, with the requirements of future command languages, especially in the areas of program invocation, data types and expressions, statements and labels, procedures, files, resource management, job environment control, error handling, language responsibility, portability and subsetability. In May 1972, IBM provided a lengthy response to the paper [30] urging continuation of project work to produce a "clear set of requirements" and firmer definitions and specifications. At the SHARE XXXIX meeting in August 1972, it was decided to develop a major position paper on Command Language Requirements. It was in final draft form at the following SHARE meeting in March, 1973 but does not appear to have been published; the project was disbanded soon afterwards.

Interest was rekindled, in March, 1975, at a session under the SHARE Languages Division entitled 'Futures: Design Goals-Command Languages'. Very few design goals were agreed upon, as would be expected from a one hour session, but considerable interest in the topic remained. At the next meeting in August, 1975, a new project was formed entitled 'Command Languages', presently managed by Donn Mukensnable of the First National Bank of Chicago. (The projects goals are outlined in [31], p.54.) Most attendees agreed that the topic is both very interesting and vital to the future of the industry, but few could agree on the methodology for handling the problem, let alone devise an eventual standard. The project conducted sessions at both SHARE meetings in 1976 and submitted resolutions to IBM on command language problems in existing IBM software. They also outlined a further white paper on OSCLs. Topics were: Human Factors, Message Services, Editors (which is interesting as most

other groups consider this is a separate issue to OSCL), Command Procedures, Command Functions (Utilities), Global Communications, Command Syntax and Security. A real problem for SHARE is that most attendees are system programmers with more important concerns and little time to spend on what is a major research undertaking.

(v) An object of increasing significance in the field of common control languages is the computer network. The advantages of a common communications or command language to a heterogeneous network of computers are very real, yet moves in this direction have been slow by comparison with other work on OSCLs.

ARPANET, the research network of the Department of Defense Advanced Research Projects Agency and probably the best known network of computers and users, was developed initially to meet the needs of specialists and experts who were capable of overcoming the problem of incompatible interfaces. As ARPANET grew, the problem compounded and the increased user population, now less concerned with the internals of ARPANET, demanded a simple, common network protocol. In 1973, a draft proposal for a network Common Command Language (CCL) was presented to a meeting of the ARPANET Users' Interest Working Group (USING), which was considering network resource sharing. It was not until January, 1974 that proposals for a network CCL received a boost. Mike Padlipsky was appointed Chairman of the CCL Definition Committee, a small group of enthusiasts actively interested in preparing a firm specification for a CCL. Basic user access protocols on ARPANET were a combination of teleprocessing network (TELNET) and remote-job-entry protocols. TELNET supports terminal access to the set of host machines available through the network making each user appear local to the remote host [38]. TELNET maps the local host characters into a network-wide format (defined for a Network Virtual Terminal) and then, at the remote host, maps them into the character set for that host. In addition, TELNET properly supports echoing and attention handling and provides a transcript of the entire communication. Network communication within a host is provided by the Network Control Program (NCP). This handles such areas as data distribution between user processes; support of host/IMP (Interface Message Processor) protocol; support of host/host protocol; and management of the IMP as a device attached to the host.

The latter three functions are seen as being independent of the host operating system and could thus be offloaded to a Front End Processor (FEP). This would lead to reduced software development costs and increased flexibility in meeting user requirements [27]. FEP utilisation also increases the feasibility of developing a Network Operating System (NOS) for a heterogeneous network to support user access to network resources. The NOS would provide a translation mechanism to permit expression of commands in a form natural to the user, as opposed to the present situation where a user must learn the command structure of each host or system accessed. The FEP could provide the necessary command translation at each interface [29]. Padlipsky's CCL Definition Committee have considerable work ahead of them before

the possibilities of a NOS and a FEP can be fully explored, but there is cause for encouragement of the view that a common network command language is a real prospect.

(vi) The Institute for Computer Science and Technology of the National Bureau of Standards has been performing an extensive study of computer networks for the Office of Computing Activities of the National Science Foundation. A report on the standardisation of user access procedures [20] is of direct interest to the study of OSCLs. "This report surveys user access protocols of six representative systems. Functional access requirements are outlined and implementation of access procedures is analysed by means of a common methodology. Qualitative assessment of standardisation possibilities identifies standardisation candidates such as: system and user signals, on-line user entries, and network wide categories of message content". This work is akin to that of Rosenthal.

2.4 Further Important Work

So far, an outline has been given of the status of research work conducted by most of the major computer societies, user groups, government agencies and standards committees. Many other individuals, organisations and software firms have been active in developing OSCL packages that may be used to replace 360/JCL, to translate a common command language in a satellite or network system to various target JCLs, or to replace the OSCL completely with extended high-level programming languages. Particular examples of this work including one of the few Australian efforts are described below.

3. RESEARCH PHILOSOPHIES

To some extent, much of the work on OSCLs has been prompted by the immediate needs of a particular installation or group and this has influenced the approach to the problem. The following are details of some different research philosophies.

3.1 Development of a New OSCL (The Standard)

The development of a new, general purpose machine and operating system independent command language that would become 'the standard OSCL' is, at best, a very long term goal. There is no guarantee or consensus that such a thing is even possible. Yet a number of groups see this as the only solution to the OSCL problem and are actually working toward it.

(i) To repeat some of the recommendations of the ANSI/X3/SPARC/OSCL report [26]:

- "There should be only a single standard OSCL."
- "Piecemeal standardisation should be avoided."

The present ANSI committee is now faced with the task of justifying the feasibility of this approach and the economics of any attempt to implement it. They are specifically charged with:

"making a determination of the need and feasibility of establishing a standard OSCL or family of OSCLs at this time, based on current technological, operational, and economic factors" [34].

Their draft report "Economic Justification of an OSCL/OSRL" (Sibley, 1976b) makes some interesting observations about the whole question of standardisation, and all that it entails - doubts about user and manufacturer acceptance, conformity of implementation, constraints to future innovation, and its cost. The paper argues that once the one time costs of developing a standard OSCL ($3M)* and re-educating users ($190M) are amortized, savings from the introduction of a standard OSCL will amount to about $67M per year. Most of the saving ($62M) comes from a saving in the need to re-educate people to each new OSCL as they move to a different computer. The balance ($5M) is an estimate of the saving in OSCL errors and seems a very small fraction of the previously mentioned OSCL error wastage of $1,450M.

If the development of a new standard OSCL is to have such a marginal effect on OSCL errors (less than half of one per cent), then surely we have to be looking at a much wider field of reference, i.e. looking at the whole programming language system from a user's viewpoint.

Other groups working towards the definition of a standard or common OSCL include the CODASYL OSCL Task Group, the Dutch Job Control Language Committee, the BCS Group 5 Working Party on Job Control Languages, to a lesser extent the SHARE Command Languages Project and, in the past, the completed definition of SJCL by CODE Inc. for the U.S. Defense Department. Each of these groups has drawn back from the mammoth task of actually developing 'the' standard OSCL.

(ii) Enslow reporting [17] on the work of the CODASYL OSCLTG, notes that two basic assumptions were established for guiding their work:

- "A decision has been made to develop a standard OSCL", and
- "The OSCL Task Group will not be able to develop the complete OSCL"

He goes on to suggest that their goal was to obtain information on what a standard OSCL would look like if one was established.

As mentioned in Section 2.2 of this paper, much of their work is based on the development of an operating system 'functional model' and a 'hierarchic model' that pictures the initiation of operating system functions using a high-level recursive language. The models may then be used and refined in a simulation process especially if combined with a model,

---

*All figures in $US.

as yet undefined, of 'a typical user'. On completing the definition of its OSCL, the group's aim is to prove the generality of their language by precompiling it down into the set of existing operating systems. To follow on this task, an eventual decompilation process is envisaged as the flow of information between system and user is bi-directional. The danger inherent in a precompiler approach is that the generalised OSCL may be restricted to a common intersection of the existing target OSCLs (see Section 3.3). The CODASYL group argue that this may be the only feasible way of receiving manufacturers' recognition of their OSCL.

(iii) The Dutch Job Control Language Committee established two working groups to prepare descriptions of two of the three 'binding classes' of job control functions [40]. The first group dealt with job structure; the second (formed later) dealt with job resource binding.

Structure binding defines the relationship between computer programs (not jobs) to be executed and introduces the concept of a 'jobstep structure', being a computer program and its related files. Combination of jobsteps yields job structures. Within these structures there needs to be a controlled selection function, a 'sub job' which is a pre-defined job structure description, and synchronisation functions. The group went on to define complete language semantics for job structure binding.

Resource binding contains those elements necessary to describe resource allocation concerning computer programs. This, coupled with structure binding descriptions, completes what they call the 'job control layer', or environment. The concept of different layers in a job control facility is an extension of Dijkstra's levels of abstraction [15]. The different levels of system hierarchy upon which this system was based made it possible to verify the logical soundness of the design and the correctness of its implementation.

The third binding class called interface binding deals with those language elements that can be used to communicate with other environments or levels. Although little work has been performed in this area at the time of their last report, it did appear that items such as file security, priority and scheduling, job protection, job attributes and system status interrogation belonged to this binding class.

It is the group's hope that "very soon, specifications of high-level job control languages will be developed as a medium of communication with the job control layer".

(iv) The first meeting of the BCS JCL Working Group was held in Queen Mary College, London on 26th March, 1974. The author was fortunate in being able to be a foundation member of this group. Major groups represented by the fifteen member organisations were those in academic research and educational establishments seeking to provide a common interface for their users to the many computing systems to which they have access, and those with projects charged with supporting software packages on many machines. Three

important groups were poorly represented: major manufacturers, manufacturers of mini-computers (often used as satellites to support common JCL interfaces) and ordinary data-processing installations. The convenors, Dr. Newman and Rod Evans were appointed Chairman and Secretary respectively.

The terms of reference of the group were:

(a) To act as a clearing house for ideas (not only receiving but also disseminating).

(b) To determine the user requirements of a job control language and, in particular, a common JCL.

(c) To specify the user interface to map requirements to operating systems.

It was obvious that items (a) and (b) had to be well advanced before item (c) could be considered in detail, and even more obvious that the different categories of OSCL users had to be established. In fact, five separate categories of use, rather than users, were defined:

- Interacting with one of a small number of specific packages (applications run).
- File manipulation and initiation of program runs.
- Program development and debugging.
- Development of application packages which other people will use (tailoring systems for people).
- Development of system programs and system (job control) interfaces for all users (tailoring systems for machine efficiency).

Once the requirements of these various uses are established, then it will be "possible to discuss language formats which would satisfy these requirements and be acceptable to the various real users. Four topics must be studied:

- language data types (variables)
- structures
- facilities
- formal syntax

(from first annual report, "Computer Bulletin, June, 1975).

After the first few meetings, an explicit framework of operation emerged for the group itself. Its main constituents were:

- a documentation scheme for labelling all the

various documents, both internal and external, for ease of reference,

- a journal of development styled on the CODASYL Cobol journal of development (the group's official status report) and

- a classification framework for dividing the work of formulating a command language. Without such a framework, a group of scattered members meeting once a month could not possibly progress very far with the real problem at hand.

During 1975, studies of the requirements of non-specialist, file handling, and compile and go users, i.e. the first two or three of the five user groupings, prompted the following major conclusions:

- A unified filestore can transcend devices on a three 'space' representation; directory space, logical space and physical implementation space. The average user need not then concern himself with such things as: multiple copies of files for backup; different logical files sharing the same physical information; files split across devices; or the actual device type, e.g. cards, tape, disk, etc.

- The editor is only one of a set of program/utilities and not part of the OSCL.

- Messages occurring in response to commands are as important to define as the commands themselves since it is equally important for the user to be able to understand them as it is for him to understand the command language.

- Compile facilities can be standardised. Output of listing, messages, debugging facilities, 'executable' output, can all be controlled by switches, as can the decision whether to continue from compilation to execution.

- A hierarchy of subroutine libraries is related to compilers that are used to fill any remaining unsatisfied references. The names of the libraries are regarded as extensions to the name of the compiler and the libraries could contain main programs as well as subroutines.

From reports received up to January, 1976, it is clear that the BCS Group had managed to steer clear of syntactic questions and had not tackled the needs of more sophisticated users. Its ultimate goal remained the definition of a machine independent JCL. The group has explicitly excluded the possibility of integration of job control into programs, since the aim is to achieve something which will be applicable to existing systems as well as future developments.

(v) The long range goal of the SHARE Command Language Project is "a single, unified command/control language for implementing and running jobs of sessions on IBM equipment. The language is to be developed from currently provided software in an orderly and stepwise, progressive manner". It is significant that a closed community of users sharing a common range of equipment should set themselves a task only minimally reduced from that attempted by the previous four groups mentioned in this Section. The reason becomes obvious when one looks at the variety of command languages available on IBM (current range) equipment, viz. JCL, TSO, operator commands (JES, HASP), CMS, CP, utilities, data base access languages, DSS, SMP, and CICS. One reason for this proliferation of command languages is the clear distinction between batch and interactive use of OS/360 e.g. JCL and TSO respectively. The SHARE project is intended to eliminate this distinction and, if possible, remove the need to use OS/JCL in future batch operations. It is too early to say whether anything definite will come from this work, but all groups working in this area should monitor this project as its members represent a very large and significant body of real users.

The five projects mentioned above involve large numbers of people, many acting as advisers only. Not one of the projects is a commercial venture (which is probably just as well) and their aims are very long-term. To succeed either singly or in concert, they will need to define a whole new body of information and theory relating to the requirements of 'users' and the primitives of 'operating systems'. Even these two terms (quoted) are not well defined.

This is where the individual worker with his own OSCL application or research project will come into his own. Without this specialised research and experience, the large standardisation projects are likely to miss the mark.

3.2 OSCL Translators (One for One)

The following OSCLs were designed and implemented as replacements for a manufacturer's OSCL. They are representative of the many systems that were probably designed to overcome both the deficiencies, and the complexities of a language such as JCL/360.

(i) JOL - Job Organisation Language is particularly significant in that it appears to be the only OSCL project of any size undertaken in Australia. JOL was initially developed by the Shell Company of Australia as a high-level language replacement for JCL/360. It has since been marketed in Australia by Clarke Computer Software, and in the U.K. by CAP Associates. The Standard Oil Co. (Indiana) has also assisted in the development of JOL [7,8].

JOL uses a "much simplified syntax notation based on the free format PL/1 language". High-level JOL procedures are translated into appropriate JCL by a compiler. During execution, a transient JOL Monitor handles all run time instructions, condition code testing, etc. It is interesting

to note that the designers of JOL went beyond providing a straight replacement for JCL/360, and actually took over some of the functions of the underlying Operating System, e.g. all catalogue references to data sets are resolved once at compile time irrespective of the number of times a data set is used; the number of OS steps is reduced by combining the execution of some programs in the one step. This merely emphasises the point that, in any system, the position of the interface between the operating system layer and the job control layer is fairly arbitrary and is certainly open to much more research.

JOL is marketed as a "proven alternative to IBM's JCL". It is claimed to have a powerful macro language for language extension. Early details of JOL's development are not available but its parentage is described as:

"from OS out of PL/1 delivered by common sense".

(ii) RCL - Reduced Control Language, is a simplified job control language for non-professional users of IBM OS/360 [1]. It was designed and implemented at the Uppsala (Sweden) University Data Centre on an IBM System/370 computer and is now used by an estimated 80% of their users.

RCL has a simpler syntax than JCL, shorter notation, defaults for missing parameters according to installation and job spectrum, and reduced redundancy by inferring certain attributes from others. RCL is less powerful than JCL but may be complemented by inclusion of JCL statements within a job. The whole job is pretranslated under HASP and use is made of one-step monitors to improve efficiency during such phases as update, compile, load or link-edit and execute.

Unlike the previous language (JOL), RCL has not been marketed and its language syntax is at a lower level. However, it was designed to solve a problem, namely, the transition to an IBM system for unsophisticated users, and this end has been achieved.

3.3 Machine Independence and Portability

Many computer installations are finding themselves increasingly locked-in to a particular operating system and hardware, through the massive investment in software. The effort involved in a change of system is considerable for both the computer centre and the ordinary user. Other organisations are finding themselves users of various heterogeneous computer networks, the major nodes of which each require a different interface or command language communications path. It is, therefore, little wonder that much hard work is under way to reduce machine dependence and to enhance the portability of computer programs.

(i) This is the prime motive for the development of GCL (General Control Language) at the U.K.A.E.A. Culham Laboratories [12-14]. Dr. Dakin's language and job language translator, JOLT, may be used to achieve OSCL portability between three target systems - ICL Multijob (System 4 comp-

uters), ICL George 3/4 and IBM OS 360/370.

An interesting feature of the GCL project is its emphasis on provision of a generalised control interface to a satellite (network) system. A satellite computer performs the translation from GCL to existing OSCLs dependent on the target computer selected for a job's execution, in this case, either an IBM/370 under MVS at U.K.A.E.A. Harwell, or the ICL 4/70 under Multijob at Culham. Dr. Dakin was therefore immediately faced with two problems.

The first concerned core and resource limitations inherent in the use of a small satellite computer (a CTL Modular One) for job language translation together with the requirement that the process must be completely independent of the target operating systems, i.e. special software in the target systems (after the style of JOL and RCL in Section 3.2) could not be used. JOLT was first implemented on an ICL 4/70 computer and moved to the CTL Modular One, using the CLSD language [6] to achieve translator portability. The first part of JOLT is independent of the target system and comprises a compiler, interpreter and a small set of primitive functions that provide the base for a self-extensible language. An early version of JOLT was reported to need 4000 16-bit words to map a somewhat limited GCL user image onto JCL for the IBM OS System [14]. To provide full GCL facilities in conjunction with normal satellite network software, JOLT will need to be segmented to remain within storage constraints and this may lead to performance problems.

The second problem relates to the most common criticism of a one to many translation process - that the high-level (GCL) end of the process must be constrained to a common subset of all target systems, i.e. the intersection of their facilities. The second part of Dakin's translator (JOLT) attempts to overcome this problem through provision of a number of sets of GCL statements that define a hierarchy of functions to perform the mapping from user level GCL statements on to the JCL of the target system, a different set being required for each target system. The functions at the top of the hierarchy (which provide simple OSCL facilities) are intended to be system independent by expressing actions in terms of user requirements. As one goes down through the hierarchy, the functions increasingly reflect the structure of the target system. In this way, users with more complex jobs may make use of the lower levels, at the price of machine dependence, all the way down to the ultimate of including target JCL within the GCL job. Although this philosophy of a hierarchic language overcomes the common intersection problem, it does present difficulties in arriving at a stable user image. Considerable work has been directed to this problem [11].

For a detailed description of the syntax and facilities of GCL, readers are referred to the previously mentioned reports. To summarise, GCL is based on the notation of mathematical functions where parameters are passed using the Algol call by value convention. Parameters are positionally defined and specify information which must be present. For information which may be defaulted (as often occurs in an

OSCL), an 'option' notation based on keywords is used. After working with Dr. Dakin on the project in the early months of 1974, I have reservations about the user acceptance of a function based syntax. At any level deeper than the basic (high-level) user image, the semantics of the language and effect of various statements are hard to follow. This is a pity, as the concept of a hierarchy of mapping functions whose dependence on the target system is restricted to a few accessible levels seems excellent. I understand that recent 'cosmetic' changes to the syntax are gaining wider acceptance, and that an increasing number of users are finding to their liking the generality of GCL, and the ability to create user functions for more complex tasks.

(ii) One of the better known machine independent command languages is UNIQUE [21-23], [25]. It was designed to provide a high-level job control interface to the University of Nottingham's ICL 1906A computer. Since its introduction, the UNIQUE language has achieved considerable popularity at the University - about 4000 jobs per week used the system in 1974 which is 80% of all jobs. Further implementations followed, producing target output for CDC7600s at Manchester and London Universities, an ICL 1906A at the Chilton Atlas Laboratory and the IBM 360/67 at Newcastle University. The major design aim was to produce a stable user interface which would offer simplicity for the novice plus extensibility for the sophisticated user, together with portability at the user level. Emphasis was placed on provision of a powerful default setting mechanism, use of keywords rather than positional parameters, use of task related high-level commands and, where possible, support for both batch and interactive use. The latter point is significant as most other 'translator' projects have concentrated on batch implementations. Like GCL, UNIQUE compiles into the target computers own OSCL and does not rely on special system-specific software modifications. Unlike GCL, emphasis was not initially put on the design of a portable UNIQUE compiler, but during 1975/76 a 'portable' compiler for a subset of UNIQUE was successfully run on ICL, CDC and IBM machines. Newman's work in this field has now extended to command language design for a network of processors [24].

(iii) It is worth mentioning two other high-level OSCLs that map onto a variety of target JCLs. The first is FOSSIL under development by the U.S. Navy [2]. The work grew out of the U.S. Navy's Cobol Compiler Validation System (CCVS) and initial implementations will interface with IBM OS, UNIVAC EXEC 8, and Honeywell GCOS. The philosophy is "to provide an interface between a knowledgeable user and more than one operating system". This project bears watching as the U.S. Navy has wielded considerable influence in the past in matters concerning language design (notably Cobol).

The other language ABLE (A Better Language Experiment), was developed at Bristol University in the U.K. [28]. It is an experiment in using a high-level programming language, Extended EULER, for job control. Where GCL and UNIQUE cater for about 80% of users, ABLE, is aimed at 100%, i.e. aimed to be the only JCL on a system. Compilers are written in Algol W (taking 42K bytes) and have been implemented on an ICL 4/75 with target languages of Multijob,

OS/360 JCL under HASP, SCOPE and GEORGE 3.

3.4 Programming Language Extensions

Previous sections of this paper have concentrated on efforts to design new, general (standard), portable OSCLs having features of higher-level programming languages. By contrast, we now look at work designed to eliminate the OSCL entirely or else to include OSCL facilities in existing high-level programming languages.

The integration of the command language into the programming language was one of the major items of discussion at the IFIP Working Conference. Little agreement was reached, probably because the question may be premature in our OSCL research experience. We still haven't reached agreement on the more fundamental issues of user requirements, operating system primitives and the ideal interface between the two. Three papers were presented at IFIP on this topic and a brief summary follows.

(i) Eiiti Wada from the University of Tokyo presented a paper entitled "On the Possibility of the Unification of Command and Programming Languages" [39] (p.109). He investigated features of languages that had been unified including this aspect of the Multics system at the Massachusetts Institute of Technology. Impediments to unification were discussed and a possible realisation of unification was sketched.

(ii) L. Trilling from the University of Rennes argued that the boundary between programming and command languages must be established in such a way that the command language is minimised as much as possible [39] (p.117). He described an ALGOL 68 based system designed according to these principles and known as SAR. The system was still at an experimental stage in 1974.

(iii) Loren Lauesen outlined extensions to the Algol language that render job control languages superfluous [39] (p.137). He further argued that such extensions could be applied to any high-level language. The advantages in this scheme are that an Algol user will not have to learn a new syntax, only extensions to Algol (some new semantics). It is claimed that this form of job control is more powerful than existing OSCLs and that the approach clarifies the basic concepts and mechanisms of job control. A new standard Algol procedure 'execute' is defined which can call a program file with a list of parameters and execute it. The system uses an Algol interpreter as its initial program (monitor) to process Algol command programs, e.g. a simple Fortran job.

```
job <user identification>

begin integer result;

execute ('ftncomp',in,'obj',out,0,result);

execute ('link','obj','prog',out,0,result);
```

```
execute ('prog,in,out,0);

end;

subroutine .. .

...

end

<data for Fortran program>
```

Obviously, one could define an external Algol procedure called 'fortran' to use the execute procedure and perform more complicated control. Initial programs (interpreters) could be written for other languages. Most importantly, the paper discusses the interface between the job process and the operating system when jobs like that shown above are executed.

Manufacturers are slowly designing OSCLs that attempt to give users an operating system interface that is at a higher level and is more natural to use. The ICL GEORGE 3 system is an early example. This was followed by ICL's new range system (2900 series).

Burroughs designed the Work Flow Language (WFL) for the B6700/B7700 computer systems [39] (p.153). WFL may be viewed either as a high-level OSCL or as an extension to an Algol-like programming language, but it is not designed to replace existing programming languages. The WFL Compiler takes over some of the traditional operating system functions (spooling & statement interpreting) and generates code just as any other compiler might. Its emitter part could be rewritten to generate code for another target system, e.g. JCL/360, so that despite its integration into the system, it is still relatively machine independent.

3.5 Network Command Languages

The rapid development of heterogeneous computer networks has opened up a whole new area for OSCL development. The work of ARPA has been mentioned (Section 2) but others are also concerned about the need to provide a general user interface to network resources.

The French have been active in this area as part of the development of the CYCLADES and SOC computer networks. Chupin [39] (p.311) proposed the concept of a Network Command Language (NCL) that may be used either directly by a user to reference a target OSCL and System or, indirectly, by using the NCL at a higher level as the interface between the two host command languages. The NCL could be extended to provide total network control, i.e. to be the only job control language available, in the event that we wish to view the network as a single computing facility. Much work remains to be done and the complications of network control probably only heighten the OSCL problem.

3.6 Formal Description

Enslow [39] (p.221) summed up the level of formal theory relating to OSCLs when he compared the present status of OSCLs with that of programming languages fifteen years ago. Then, languages like Fortran were developed without any idea of the theory of language structure or grammar and were really ad hoc solutions to immediate problems in specific environments. Languages such as Pascal and Algol 68 are a direct result of advances in the theory of language design. Enslow argued that without the same sort of theoretical advances in the understanding of user requirements and operating system interfaces, the development of generalised (standard) OSCLs is going to be severely hampered.

Some work is appearing in the literature. Neuhold has studied the formal semantics of operating systems [18,19]. Niggemann [39] (p.223) described a method for the semantic description of command languages based on the Vienna Definition Language (VDL). He gave an example of the method by applying it to a simple, non-existent, batch command language for a system based on sequential files. The developed model appears general enough to be applied to systems having more complex data structures although it has not been applied to existing OSCLs.

Weller [39] (p.237) is examining current descriptive methods to see if they may be applied to command languages. In particular, the aim of his work is:

- to examine which properties the semantic description of a command language has to fulfil in order to be understandable and useful for a normal user.

- to examine how far existing semantic description methods can fulfil this target; and

- to examine the extent to which new or altered methods are necessary.

He concluded that user-oriented descriptions require methods which differ from the existing methods, and that more attention should be paid to this area rather than to the more traditional motives of proving properties of command programs, investigating language mechanisms and gaining rigorous definitions for implementers.

4. THE FUTURE

After studying the many different research projects concerned with OSCLs, one wonders whether their varied approaches and conclusions will ever come together in the realisation of a JCL to end all JCLs or, more importantly, a clean general purpose operating system interface.

Although the IFIP Working Conference on Command Language didn't solve many OSCL problems, it did bring almost everyone in the field together so that each gained a much better understanding of the other's viewpoint. The apparent gulf between proponents of separate generalised command languages

and those advocating programming language extensions was narrowed down to a question of emphasis in the way one describes the problem solution. Surprising agreement was reached on a number of models that could be used to define problem areas. The proceedings of the conference [39] provide a valuable base upon which future work may be planned and from which past experience may be drawn.

An IFIP Working Group (WG2-7) was formed as a consequence of the conference and its members have a fairly wide cross section of views - D.W. Barron, P.H. Enslow, J.B. McKeehan, C. Unger, D. Beech, G. Gram, E.H. Sibley, I. Dahlstrand and Chairman, F. Hertweck. The only major groups not represented appear to be those working on formal descriptions and computer networks; unfortunately perhaps the two least advanced areas. With the appointment of Dr. P. Poole from Melbourne University as Australia's Representative on IFIP Technical Committee 2 (which oversees WG2-7), we should be in a much better position to monitor the progress of their work in the future and to contribute our own thoughts and ideas.

There can be no doubt that the field of OSCL research is both pressing and wide open. If we are to make best use of our computing resources and continually improving high-level programming languages, then a concerted effort must be made to improve the interface between an operating system and the very large body of non-sophisticated computer users.

REFERENCES

[1] K. Appel, Reduced Job Control Language for Non-Professional Users, Command Languages, Proceedings of the IFIP Working Conference, ed. C. Unger, p.71, 1975, North-Holland, Amsterdam.

[2] G.N. Baird, Fredette's Operating System Interface Language (FOSIL), Command Language, Proceedings of the IFIP Working Conference, ed. C. Unger, p.267, 1975, North-Holland, Amsterdam.

[3] D.W. Barron and I.R. Jackson, The Evolution of Job Control Languages, Software-Practice and Experience, Vol. 2, 1972, p.143.

[4] R.L. Basford, Command Languages - Alternatives to JCL, Proceedings of SHARE XXXIV, p.1940-1955, March 1970.

[5] G.D. Brown, System/360 Job Control Language, John Wiley & Sons, Inc., New York, 1970.

[6] M. Calderbank and V.J. Calderbank, A Portable Language for System Development, Software, Vol. 3, 1973, p.309-321.

[7] Clarke Computer Software, JOL, Concepts and Facilities Manual, CCS-J005-7508, 105 Collins St., Melbourne, 3000, Vic., Australia, 1975.

[8] Clarke Computer Software, JOL, Reference Manual, CCS-J006-7509, 1975.

[9] CODASYL-OSCLTG, The Operating Systems Command Language Task Group Technical Report, December 1973, 33 pp. Also presented by E.H. SIBLEY in Command Languages, Proceedings of the IFIP Working Conference, ed. C. Unger, p.353, 1975. North-Holland, Amsterdam.

[10] H.W. Codow, OS/360 Job Control Language. Prentice-Hall, Inc., Englewood Cliffs, N.J. 1970.

[11] R.J. Dakin, How Stable is the GCL User Image, Internal Culham Report, SEN 5/73, (Unpublished), 1973.

[12] R.J. Dakin, Preliminary GCL User Manual, CLM-PDN 1/74, UKAEA Culham Laboratory, Abingdon, Oxon, 1974.

[13] R.J. Dakin, A General Control Interface for Satellite Systems, Command Languages, Proceedings of the IFIP Working Conference, ed. C. Unger, p.281, 1975a, North-Holland, Amsterdam.

[14] R.J. Dakin, A General Control Language: Language Structure and Translation, The Computer Journal, Vol. 18, No. 4, 1975b, p.324-332, November, 1975.

[15] E.W. Dijkstra, The Structure of 'THE' - Multiprogramming System, CACM Vol. 11, No. 5, May 1968.

[16] P.H. Enslow, Jr., Draft Standard Definition of Digital Computer Operating System and Control Language Terminology, Document P486/Dl, Standards Sub-committee, IEEE Computer Society Technical Committee on Operating Systems, 15 September 1972, 216 pages.

[17] P.H. Enslow, Operating System Command Languages. A Brief History of Their Study, Command Languages, Proceedings of the IFIP Working Conference ed. C. Unger, p.5, 1975, North-Holland, Amsterdam.

[18] E.J. Neuhold, Towards the Formal Description of Operating Systems, SIGPLAN Notices, Vol. 8, No. 9, September, 1973, p.123-126.

[19] E.J. Neuhold, The Formal Semantics of Operating Systems, Proceedings of the International Computing Symposium 1973, Davos, p.127-133, North-Holland, Amsterdam 1974.

[20] A.J. Neumann, User Procedures Standardisation for Network Access, Institute for Computer Science and Technology, National Bureau of Standards, Washington, D.C. 20233, Report No. NBS-TN-799, October 1973.

[21] I.A. Newman, The UNIQUE Command Language - Portable Job Control, Proceedings of DATAFAIR, 1973, p.353-357.

[22] I.A. Newman, Job Control as a part of Standard Program Usage, Computing Europe, 6 June 1974, p.8-9.

[23] I.A. Newman, Machine Specific Facilities in a Machine Independent Command Language, Command Languages, Proceedings of the IFIP Working Conference, ed. C. Unger, p.91, 1975, North-Holland, Amsterdam.

[24] I.A. Newman and N.S. Fitzhugh, Command Language Design for Networks of Processors, Proceedings of EUROCOMP 75, 1975.

[25] I.A. Newman and M.A. McConachie, A Guide to the UNIQUE Command Language, University of Nottingham, 1974.

[26] OSCL 71, Report to SPARC From the Ad Hoc Committee on Operating System Control Language, July 6, Document Library, System Development Corporation, 2500 Colorado Ave., Santa Monica, California 90406. Also, the first nine pages of the report --- Introduction, History, Semantic, Conclusion, Recommended Scope and Program of Work of the Study Committee, Language/Function Survey, Types of Area of Command Language, Hardware Consideration, Recommendations --- were published in the ACM SIGPLAN Notice, November 1971, No. 10, p.41.

[27] Michael Padlipsky, A proposed protocol for connecting host computers to ARPA-like networks via front-end processors, Network Working Group, RFC No. 672, NIC No. 31117, TIP, October 1974.

[28] Parsons, A High Level Job Control Language, Software-Practice and Experience, Vol. 1, March 1975.

[29] Robert Rosenthal, Accessing on-line network resources with a network access machine, IEEE Intercon 1975, IEEE, New York.

[30] SHARE SSD226, IBM Response to Share Command Language Paper - May 25, 1972, published in SSD No. 226, September 1972.

[31] SHARE SSD270, Command Languages Project, an outline of the projects goals, SSD No. 270, December 1976, p.54.

[32] B.H. Shearing, Job Control Languages As They Are and As They Might Be, Proceedings of BCS Symposium on Job Control Languages --- Past, Present, and Future, National Computer Centre, Manchester, England. Also published as an article in Computing Europe, London, 2 May 1974, p.10.

[33] E.H. Sibley, Noted in Passing - short communication from Dr. E. Sibley, Operating Systems Review, Vol. 10, No. 3, p.8, 1976a, July 1976.

[34] E.H. Sibley, Economic Justification of an OSCL/OSRL, Operating Systems Review, Vol. 10, No. 4, p.7, 1976b, October 1976.

[35] D. Simpson, editor, Proceedings of BCS Symposium on Job Control Languages --- Past, Present, and Future. National Computer Centre, Manchester, England, 1974.

[36] SJCL, Standardised Job Control Language, Five reports numbered AD-742 540 to 544 obtainable from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, Virginia 22151 on either paper or microfiche, 1971.

[37] J.M. Taylor, Command Language Position Paper, SHARE Secretaries Distribution, SSD No. 221, p.69-87, April 1972.

[38] TELNET, TELNET Protocol Specification, NIC No. 186391, Network Information Centre, Stanford Research Inst., Menlo Park, California, August 1973.

[39] C. Unger, editor, Command Languages, Proceedings of the IFIP Working Conference on Command Languages. North-Holland, Amsterdam, American Elsevier, New York, 1975.

[40] H.J. Weegenaar and D.K. Wielenga, Towards a Generalized Command Language for Job Control, Command Languages, Proceedings of the IFIP Working Conference, ed. C. Unger, p.191, 1975, North-Holland, Amsterdam.

APPENDIX A

Possible Work Program for BCS JCL Group

1. Investigate the need for a common JCL.

    1.1 Conduct a survey to see who wants a common job control language.

    1.2 Assess the likely effect of developments such as the use of computer networks on the need for a common JCL.

2. Study existing JCLs, classify them, and analyse their strong and weak points.

3. Investigate the possibility of developing a common JCL for existing computer systems.

4. Investigate the possibility of developing a common JCL for future computer systems.

5. Establish the proper spheres of influence of JCL.

    5.1 What jobs can it control?
    e.g. offline or batch processing
    on-line text processing
    on-line interactive work
    system interrogation
    system control (operator commands)
    system generation and configuration.

    5.2 Should a common job control language be capable of specifying all possible tasks that can be undertaken on any computer system.

6. What should a JCL be like?
    e.g.
    - distinct from or integrated with programming languages?
    - based on an existing programming language?
    - high level or low level (or both)?
    - how should it identify resources such as programs, files, devices?
    - are common naming conventions possible?
    - should a common JCL be file-based?

10
What's So Special About LISP?
Peter W. Milne

ABSTRACT

LISP addresses a task basic in computing; the evaluation of symbolic expressions. Since its development, some 17 years ago, LISP has been the source of many ideas, both theoretical and practical, which have had an influence on programming language definition, design and implementation. This paper briefly reviews the history, applications and influence of LISP.

1. INTRODUCTION

LISP has been around now for about 17 years. Though implementations exist for most machines it is not supported by any manufacturer. There is no LISP standard so implementations are usually not particularly compatible but are often inovative. Though not widely used, LISP is almost universal in Artificial Intelligence and Natural Language research and has been used in several Algebraic Manipulation systems.

LISP is usually described as a list processing language. Though this is strictly true, leaving it at that does LISP a great injustice. In fact an argument can be made that LISP is not a particularly adequate language for general list processing. The only type of list node LISP can construct and manipulate directly is one with two fixed field pointers, thus many list structures will be expensive to represent and manipulate.

A more insightful view of LISP is that it represents an approach to the problem of mechanizing the evaluation of symbolic expressions. The representation chosen for symbolic expressions being lists.

A complaint one often hears voiced against LISP is that programs written in it execute too slowly. Part of the blame for this can be attributed to the fact that many implementations are interpreter based. However, when developing a program in an interactive environment, the slowness of an interpreter is more than offset by the way it facilitates debugging. The fact that the architecture of current machines is ill-suited to LISP implementation also contributes to slowness. Whilst we have strong intuitions about how long a reasonable FORTRAN program should take to find the roots and vectors of a matrix we don't have intuition to guide us in symbolic calculations. What is a reasonable time for a LISP program to take to differentiate a polynomial symbolically?

In spite of the fact that other languages are now available with better notation, greater run-time efficiency or larger varieties of data structures, LISP persists. LISP

with its unique blend of simplicity and power is increasingly being taught in Computer Science courses hopefully not so much for its features as a programming language but for what an understanding of the issues underlying LISP can reveal about programming languages in general. LISP facilitates such study by allowing one to avoid getting bogged down with details of syntax.

This revival of interest in LISP has led, in recent times, to the publishing of two books [4,14] aimed at the Computer Science market with a third [1], a major work, due late 77. Recently, several designs for LISP machines have been proposed [3,7]. A LISP machine [5,8] has actually been built at MIT and there is a good prospect that it will be taken up by a major computer manufacturer.

Right from the beginning LISP has inspired many ideas, both theoretical and practical, which have had an influence on programming language definition, design and implementation. This paper briefly reviews the history, applications and influence of LISP.

2. BRIEF HISTORY OF LISP AND ITS APPLICATIONS

LISP was developed by a group led by John McCarthy at MIT in the late 50's and early 60's. It was designed to facilitate experiments with an ambitious (and never fully realized) system called the Advice Taken [9]. A programming system was required which could manipulate symbolic expressions.

The system they developed was based on a particular implementation of the λ-calculus, a formal axiomatic system of logic developed by Alonzo Church in the 40's which can be interpreted in various ways as a model of computation.

The first implementation was on an IBM 704 from whose word layout the LISP primitives CAR and CDR received their names. When the 709 became available a new implementation was prepared which became known as LISP 1.5. Implementation followed for the 7090, 7094 and PDP-1 amongst others.

Up until now applications of LISP had suffered from the limited size of core memory available. With the advent of PROJECT MAC and the arrival of a PDP-6 at the AI lab. some relief from memory size problems was offered by virtual memory.

LISP, though developed as a batch system, turned out to be particularly suited to interactive use. Also on-line editors with a knowledge of LISP syntax greatly eased the classic LISP notational problem of unmatched parentheses. Debugging was assisted too by utilities which allowed selective tracing of function calls and results, facilities for setting breakpoints dynamically, prettyprint output formatting routines etc. With an editor, interpreter, compiler and debugging routines all available at the same level to a logged in user, LISP probably can lay claim to having introduced the now fashionable concept of a Programming Laboratory.

Much of the slowness of the early LISP interpreters was due to their use of a linear search to find a variable's current binding in an association list of variable/value pairs. For the PDP-6 implementation this a-list was replaced by a technique known as shallow-binding in which the current value of a variable is accessible without searching and a history of its previous bindings is stored on a stack so that they may be restored if necessary. Though offering a worthwhile improvement in speed, it turned out that it was difficult to implement a LISP system which was as general as one using an a-list [12]. This system was exported to Stanford University where it eventually became known as LISP 1.6.

On the applications front several large programs were constructed by AI workers which in some cases could rival humans at certain specific tasks. These programs attempted to solve problems by using domain specific heuristics to reduce to manageable proportions the amount of tree searching they had to do to find solutions. Such programs had to be developed somewhat empirically as the effect of adding or altering heuristics was usually unpredictable. Interactive LISP greatly aided their construction.

In his proposal to ARPA for AI funding in late 72 [10], Minsky notes that language development at MIT had been unusually stable for a decade with LISP used almost exclusively and that LISP development had been conservative. However, an AI oriented language called PLANNER, embedded in LISP, had been developed.

PLANNER was the first of a family of AI oriented languages which have been developed. LISP, though an excellent recursive evaluator, did not provide directly such features as back-tracking, coroutines, tree searching or pattern matching. If you needed these facilities you had to program them yourself. This led to much duplication of effort and the extra code, necessary to implement these features, added to program complexity. For example METEOR, CONVERT and SCHATCHEN (which is Yiddish for match-maker), were all pattern matching languages written in LISP by different people as part of larger programs. PLANNER had some of these features built in and also introduced the idea of pattern directed function invocation.

PLANNER and its descendants have, amongst other things, served to focus attention on the problem of writing clean but efficient interpreters in LISP for languages which support control structures other than recursion.

3. SOME INFLUENTIAL FEATURES OF LISP

LISP has always acted as a kind of counter example to the trend towards strongly typed, syntax heavy languages [11]. Programming languages usually consist of a mixture of applicative and imperative features. LISP is almost totally applicative. Recent work [15] has shown that many imperative constructs can be modelled in an applicative language.

Higher-order languages are now usually defined in terms

of a definitional interpreter. The use of this technique in LISP was novel and has led to the idea of abstract syntax and the Vienna Definition Language. Definitional interpreters given for LISP have usually been based on an abstract LISP machine which implicitly evaluates function arguments left to right, call-by-value and implicitly is capable of recursion. That is the interpreter doesn't define these aspects of LISP semantics though it is possible to write interpreters in which these things are explicit. LISP's use of this technique has served to focus attention on the advantages and disadvantages of language definition in this way and on the various ways to go about writing definitional interpreters [13].

In LISP, functions may return functions, which may contain free variables. Scoping in LISP is normally dynamic but lexical scoping is available for use in such cases. This implies the retention of environments, created by invocations of functions in which control is no longer nested. This, along with the desire to be able to support such control regimes as coroutines and backtracking in LISP, has led to the development of the spaghetti stack [2] which is an efficient technique for the retention of function activation blocks with the desirable property that, if no retention is required, it behaves as a conventional stack.

Because of the nested nature of lists, recursion was the natural control structure to choose for LISP. The predominant role given to recursion in LISP was perhaps a bit radical at the time but recursion is no longer novel, being available in most reasonable programming languages, though its full power and elegance are still being realized. Unfortunately recursive code can execute very slowly and is a factor in the slowness of LISP. This has inspired various techniques for unwinding recursions into iterative code for compilers and recently, a technique for interpreting tail-recursive LISP functions iteratively [6].

The representation of both data and programs as lists in LISP means that LISP is unusual amongst higher level languages in that programs can meaningfully manipulate other programs. This fact, plus a knowledge of the representation of expressions in LISP, makes it possible to write a LISP program which will evaluate any LISP expression. A programming language with this property is said to be UNIVERSAL. This property makes possible powerful LISP editing and debugging packages which can be used to modify or inspect code, call upon the LISP evaluator EVAL to evaluate it, and monitor its execution - all at source code level.

Garbage collection, which is not really a language feature of LISP, is necessary in practical implementations and was first introduced to computing by LISP.

4. CONCLUSION

This paper has presented a brief review of the history and applications of LISP. Over the years LISP has been both a test-bed and a source of many ideas. LISP still possesses features which are unusual or unique and will probably

continue for some time yet to be a source of influence out of all proportion to its practical usage. Mention has been made of some of these ideas and features.

REFERENCES

[1] J. Allen, Anatomy of LISP, to be published by McGraw-Hill, 1977.

[2] D. Bobrow and B. Wegbreit, A model and stack implementation of multiple environments. Comm. ACM 16,10, Oct. 1973, 591-603.

[3] L. Peter Deutsch, A LISP machine with very compact programs. Proc Third Internat. Joint Conf. on Artif. Intell. Calif., 1973, 697-703.

[4] D.P. Friedman, The Little LISPer. Science Research Associates, Inc., Palo Alto, Calif., 1974.

[5] R. Greenblatt, The LISP machine. Working paper No. 79, MIT, Nov. 1974.

[6] P. Greussay, Iterative interpretation of tail-recursive LISP procedures. TR-20-76, University of Vincennes, Paris, Sept. 1976.

[7] A. Guzman and R. Segovia, A parallel configurable LISP machine. Technical report 7, 133, Computer Science Dept., IIMAS, National University of Mexico, 1976.

[8] T. Knight, CONS. Working paper No. 80, MIT, Nov. 1974.

[9] J. McCarthy, Programs with common sense. Proc of the Teddington Conf. on the Mechanization of Thought Processes, H.M. Stationary Office, 1960.

[10] M. Minsky, Proposal to ARPA for continued research on A.I. MIT A.I. Memo No. 269, Oct. 1972.

[11] M. Minsky, Form and Content in Computer Science. J.ACM 17, 2, Apr. 1970, 197-215.

[12] J. Moses, The function of FUNCTION in LISP. SIGSAM BULL. Jul. 1970, 13-27.

[13] J.C. Reynolds, Definitional interpreters for higher-order languages. Proc 27th Nat. Conf. ACM 1972, 717-740.

[14] L. Siklossy, Let's talk LISP. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

[15] G. Steele and G. Sussman, LAMBDA : The ultimate imperative, MIT A.I. Memo No. 353, MIT, Mar. 1976.

11
Semantics — The Scott Era
Malcolm C. Newey

1. INTRODUCTION

A programming language without a formally specified semantics is like Frankenstein's monster. You cannot be sure of what it will do in all circumstances, and to put one's trust in such a creation is to live dangerously! On the other hand, the results of a program in a rigorously specified language may be predicted (computed) for particular input data. Of course, complete rigour in specification is only achieved using a precise formalism.

Predicting results in particular cases is not sufficient, however, to prevent the unexpected. Since it is possible to prove that a program meets its specifications for all inputs, we further require that a semantics formalism be suitable for use in *machine checkable* reasoning about programs in that language. In fact, the most important application of semantics is in the proving of program correctness.

Formalisms for specifying semantics are usually tagged with the adjectives operational, propositional and denotational. As names of flavours, they are (like caramel) not definable in absolute terms; expect only to get a feel for their use.

An operational semantics for a language is one in which language constructs are specified by giving their effect on the state of some machine. Garwick [1] may never live down having asserted that a language is defined by a compiler running on a particular digital computer. Others, notably Landin and the Vienna Definition Language group have employed abstract machines to define Algol [2-4] and PL/I [5].

In the propositional approach, as promoted by Floyd [6], and Hoare [7] and Dijkstra [8], a language is defined by a set of axioms and proof rules in such a way that properties of programs may be proved. A major achievement for this method was an axiomatic definition of Pascal [9].

We characterise denotational semantics as the association of programs in a language with the abstract functions they compute. These functions may yield a list of outputs (given a list of inputs) or may yield a new state (from an initial state). They are abstract in the sense that equivalent programs (ones that effect the same change of state) are associated with the same function. Mathematical semantics is used as a synonym since it is just what logicians understand by 'semantics'. The method was used by McCarthy for Pure LISP [10] and by Strachey for CPL [11] but awaited the inventions of Scott to achieve wide, effective and rigorous applicability. The main point of this paper is to introduce

this concept and explain some related jargon that is making more frequent appearance in computing literature. Other applications of Scott's work in semantics and verification will also be mentioned.

In summary, if we think of a program as computing a function then

(i) *operational* semantics specifies a method of computing the function;

(ii) *propositional* semantics specifies axioms and rules for establishing properties of the function;

(iii) *denotational* semantics gives an expression for the function in a mathematical logic.

Formal semantics can be specified in each of these styles in such a way that verification is possible.

2. DENOTATIONAL SEMANTICS

If a formal language has a mathematical semantics then the expressions in that language *denote* mathematical objects of some sort. It is clear, for example, that numerals are expressions in some language for representing numbers. Programs are also expressions in a formal language but what sort of mathematical objects could they denote?

I shall rely of your familiarity with numerals to illustrate some points:-

(i) One may have more than one language to talk about some collection of objects (e.g. Roman and binary numerals).

(ii) In a single language there may be different expressions denoting one object (2, 02, 002 etc).

(iii) Given a language, *and* a suitable meta language, one can specify a semantic function mapping expressions (in the object language) into objects denoted. B, below, maps binary numerals into numbers.

$B[\![0]\!]=\emptyset$ $\quad$ $B[\![\alpha 0]\!]=B[\![\alpha]\!]+B[\![\alpha]\!]$

$B[\![1]\!]=succ(\emptyset)$ $\quad$ $B[\![\alpha 1]\!]=succ(B[\![\alpha 0]\!])$

where α is a string of bits (0 or 1) and succ, $\emptyset$ and + belong to the metalanguage.

(iv) Equivalence of expressions from different languages may be indicated as equality between denoted objects.

e.g. $\quad B[\![11100]\!] \equiv D[\![28]\!]$

(v) The semantic function can be used to prove the correctness of syntactic procedures involving expressions of a language. For example, if bitwise addition of binary numerals is written $\oplus$, we can prove

$$B[\![x \oplus y]\!] \equiv B[\![x]\!] + B[\![y]\!]$$

It should be clear that the following expression from a well known programming language denotes a mathematical object, namely the function which is undefined on negative integers and which is 'factorial' on the non-negative integers.

```
integer procedure fac(n);

integer n;

fac := if n = 0 then 1 else n * fac(n-1);
```

What hindered Strachey, in his attempt to write down the semantic function for a similar language, was the lack of a suitable metalanguage in which to represent the functions denoted.

3. SCOTT'S LOGICS

There are a number of formal systems being used (in the specification of semantics) that arise from Scott's proposals. They have several traits in common: *domains, types, approximation* and a characterisation of *computable functions.* The notion of approximation (as an ordering of domains) is basic and is the reason the enterprise has been labelled the lattice-theoretic approach to theory of computation.

One should be more than a little curious about the necessity of mathematical breakthroughs for success in denotational semantics. We can start with the shortcomings of classical predicate calculus.

The largest single difficulty with predicate calculus is its orientation towards *total* functions and predicates (ones that are defined for all arguments). In discussion of real programs one often has to state that the result of some procedure is undefined for certain inputs. If a logic has to be augmented by English for such statements to be made, then it is simply inadequate.

There is a problematic emphasis in predicate calculus on introducing functions by means of axioms that are supposed to capture all relevant properties. It is often much more convenient and reliable to write down the function explicitly; hence the deficiency is the lack of terms comparable to those of the λ-calculus.

Finally, there are empirical problems with mechanizing predicate calculus. A lot of work has been done on theorem-proving in first-order predicate calculus without great success and almost no substantial attack has been made on higher order logics (because of the greater difficulty). If we were looking to conventional logics as a basis for mathematical semantics we would be faced with a grim choice; on one hand the higher-order ones have poor prospects for mechanization whereas first order predicate calculus is too restrictive as regards types.

The Scott logics are a neat solution to the problems of partial functions, λ-calculus terms and typing. It is too early to claim anything about mechanization (almost no work has been done) but there is an informed optimism. Moreover, Scott was able to found a very general induction rule on a characterization of computable functions.

4. LATTICES

Before starting on a description of an interesting Scott logic, I want to cut through some mystique that surrounds lattice theory. Although this is an uncommon area the basic notions are simple, and rather important now in computer science.

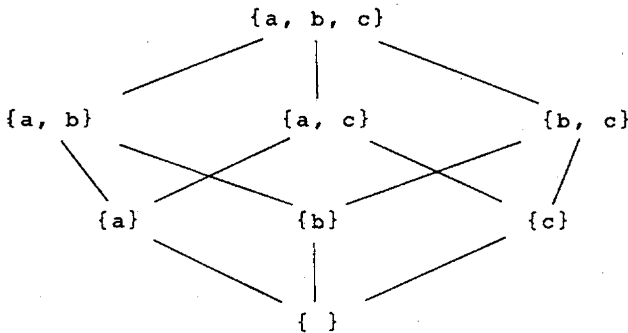
I remind you that a partial order (on a set) is a relation that is transitive, reflexive and antisymmetric:

$$x \leq y \wedge y \leq z \supset x \leq z$$

$$x \leq x$$

$$x \leq y \wedge y \leq x \supset x \equiv y$$

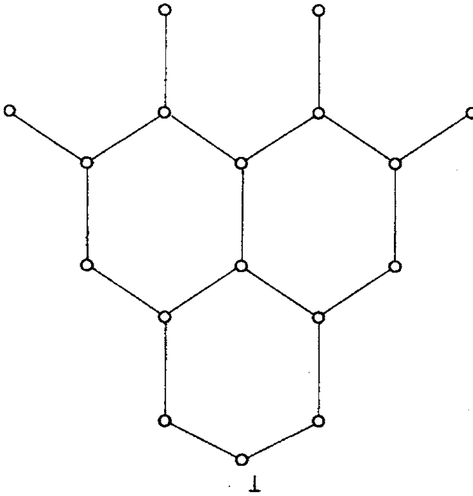
A *complete lattice* D, is a partially ordered set in which each subset has a *least upper bound* (in D) and a *greatest lower bound* (in D). An example of a complete lattice is the power set of a set partially ordered by inclusion ($\subseteq$):



Complete lattices are used extensively in Scott logics but the important properties of them are shared by *complete lower-semi-lattices (complete partial orders)*.

A *complete partial order* $(D,\prec)$ is a partially ordered set which has a minimum element $\perp_D$ and in which every chain has at least upper bound in D. (A chain is a set $\{x_i | i \geq 0\}$ such that $x_0 \prec x_1 \prec x_2 \prec \ldots\ldots$.)

Finite trees, with the root at the bottom, are examples of complete partial orders, as is the following:



5. LOGIC FOR COMPUTABLE FUNCTIONS (LCF)

LCF grew out of unpublished lecture notes by Scott in 1969. This particular Scott logic is notable because it was used as a basis for a computer program which both checked and assisted in the generation of proofs. This program [12] implemented the logic as presented rigorously in [13] and was the vehicle for several projects in the mathematical theory of computation.

That version of LCF has been superseded [14] and an improved implementation done [15]. This revised form of the logic is the one to be described below. We will see that it is a *calculus* of *equations* between *typed lambda-terms*.

(a) Domains and Types

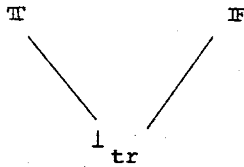
The objects in LCF's universe of discourse are grouped into *domains* according to their type. There will be domains of data objects, domains of structured data, domains of predicates, domains of functions and domains of functionals. Each of these domains has a label which is the *type* of the objects in the domain.

For example, there is a domain of integers, a domain of pairs of integers, and a domain of binary functions on the integers. The types of these domains are int, int × int and (int × int) → int, respectively.

Each LCF domain is a complete partial order with ordering relation $\sqsubseteq$, which is called 'approximation'. s $\sqsubseteq$ t (read s approximates t) means that t is at least as well defined as s but consistent with s (examples follow). The minimum element of a domain D_x, is written $\perp_x$ and is the totally undefined element of that type. $\perp_x$ is written $\perp$ when no ambiguity is possible and read as 'bottom'.

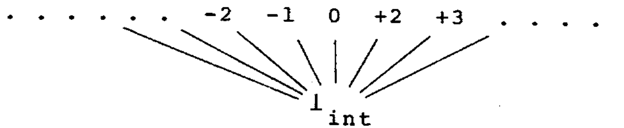
An important domain in LCF is that of truth values - D_{tr}. Its type is tr, its elements are $\mathbb{T}$, $\mathbb{F}$ and $\perp_{tr}$ (true, false

and undefined) and the partial ordering is illustrated by the diagram:-



Of course, we write $\bot_{tr} \sqsubseteq \mathbb{F}$, $\mathbb{F} \sqsubseteq \mathbb{F}$ etc.

Other 'base' domains may be axiomatised to characterise various data types. Of particular importance is the domain of integers (D_{int}):



For each pair of domains D_x and D_y, there is another domain $D_{x\to y}$ of continuous functions (see below) from D_x to D_y. The objects (functions) in the domain have type $(x\to y)$. Starting with several base types, we can generate a countable set of types (which correspond to domains). To illustrate that familiar functions are to be found in these domains, we note that the oddness predicate on integers is in $D_{int\to tr}$ and (arithmetic) negation is a function of type $(int\to int)$.

A *continuous* function (f) from one complete partial order (D) to another (E) is one which preserves least upper bounds of chains. That is for all chains, χ, in D

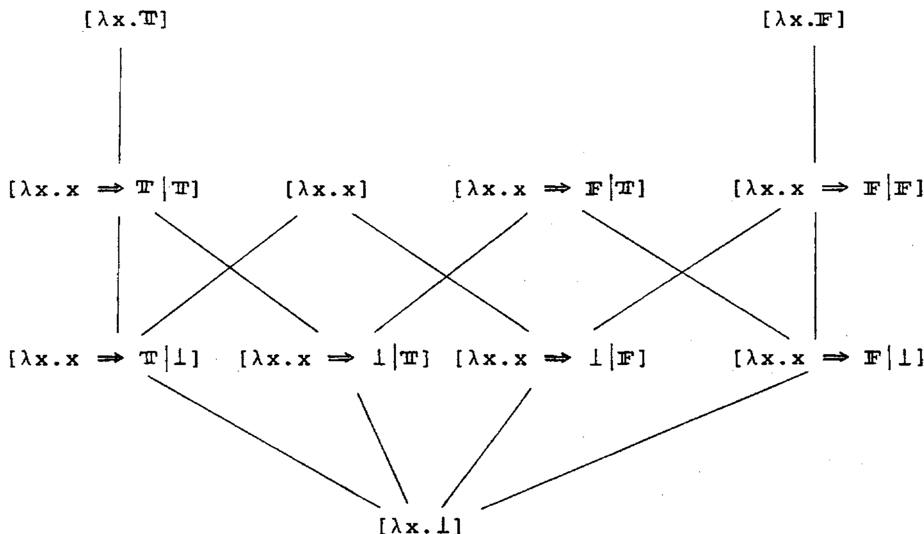
$$lub(\{f(x) \mid x \in \chi\}) \equiv f(lub(\chi))$$

It is Scott's thesis that the computable functions are the continuous ones.

The most important property of continuous functions is their monotonicity. A function f is *monotonic* if, for all x and y satisfying $x \sqsubseteq y$, $f(x) \sqsubseteq f(y)$. This is readily proved by considering the chain $\{x,y\}$.

The ordering in a functional domain, $D_{a\to b}$, reflects the orderings in domains D_a and D_b. We say $f \sqsubseteq g$ in $D_{a\to b}$ iff for all x in D_a, $f(x) \sqsubseteq g(x)$ (in D_b, of course). It should be clear that the least element, $\bot_{a\to b}$, in $D_{a\to b}$ is $(\lambda x.\bot_b)$.

The domain $D_{tr\to tr}$ illustrates these points:-



The $p \Rightarrow x|y$ construct is interpreted as "if p then x else y". Note that each of the functions in this diagram is monotonic. You may also check that none of the 16 other functions from the set D_{tr} to itself is monotomic. Finally, you can check that some function F is shown to be 'less than' some other function g in the diagram exactly when $f \sqsubseteq g$ follows from the definition.

Two other type (domain) building operations are available in the logic - cartesian product (x) and disjoint sum or union ($\oplus$). $D_{a\times b}$ is the set of pairs of objects (x,y) where x is from D_a and y is from D_b. Of course, $(x_1,y_1) \sqsubseteq (x_2,y_2)$ iff $x_1 \sqsubseteq x_2$ and $y_1 \sqsubseteq y_2$. The bottom element of $D_{a\times b}$ is $(\bot_z,\bot_b)$. $\times$ may be used in conjunction with $\rightarrow$ for functions of several variables. $\oplus$ is used to give domains which are unions of other domains. For example, it can be useful to have a domain whose elements are integers <u>or</u> identifiers.

(b) <u>Terms</u>

Terms are built from variables and constants by application, lambda-abstraction, pairing, by the conditional expression and the least-fixed-point constructions and a sprinkling of syntactic sugar. Variables are simply identifiers while constants may be either identifiers or certain special symbols such as 0, +, = etc.

Infix comma is used for pairing, t_1,t_2 has type $\alpha\times\beta$ iff t_1 has type α and t_2 has type β.

An application (of a function to an argument) is built

from two terms, s and t, say, by writing s(t). Sometimes we omit the parentheses, as in ~p, or write a binary function in infix notation, as in x+1, instead of using +(x,1). In an applicative term, if the function has type $(\alpha \to \beta)$ then the argument must have type α and the type of the whole term is β.

In LCF, a lambda term is written $(\lambda v.t)$, where v is a variable (the formal parameter) and t is a term (the body). If the types of v and t are α and β, then the type of $(\lambda v.t)$ is $(\alpha \to \beta)$. Syntactic sugar for functionals lets us write $(\lambda x\ y.t)$ where we mean $(\lambda x.(\lambda y.t))$.

A conditional expression, $(s \Rightarrow t_1 | t_2)$, is constructed from three terms: the condition (s), of type tr, and two arms (t_1 and t_2), which agree in type with the conditional as a whole. $s \Rightarrow t_1 | t_2$ denotes t_1 in the case that s is true ($\mathbb{T}$). t_2 is case s is false ($\mathbb{F}$) and otherwise (where s is $\perp$) is undefined. It is, of course, read as "*if* s *then* t_1 *else* t_2".

The least fixed point construction is the main means of building terms which denote recursive functions. The term $(\mu f.\ t)$ denotes the function forced by the recursion in the definition $G \Leftarrow t[G/f]$.

An an example, $(\mu f.(\lambda x.(x=0) \Rightarrow 1|(x*f(x-1))))$ is that function specified by

$$FAC \Leftarrow (\lambda x.(x=0) \Rightarrow 1|(x*FAC(x-1))),$$

namely the factoral function (from D_{int} to D_{int}) which is undefined on the negative integers.

Let's explain the term 'least fixed point'. We say that a function F has a fixed point X if $F(x) \equiv X$. Functions may have multiple fixed points but it is provable in Scott logics that there is always a minimum one. That is, for any function, F, there is an X_0 such that $F(X_0) \equiv X_0$ and if $F(X) \equiv X$ then $X_0 \sqsubseteq X$. Finally, $(\mu f.t)$ is the least fixed point of the function $(\lambda f.t)$, by definition. Now, if G is used as a name for $(\lambda f.t)$ we get $G \equiv (\lambda f.t)(G)$ or $G \equiv t[G/f]$.

As an illustration of multiple fixed points we note that the function $(\lambda x.\ x \geq 0 \Rightarrow FAC(x)|0)$ is also a fixed point of $(\lambda f.(\lambda x.(x=0) \Rightarrow 1|(x*f(x-1))))$ but that FAC is 'less than' it.

(c) <u>Formulae, Sentences and Proofs</u>

The *atomic formulae* of LCF are *equations* of the form $t_1 \sqsubseteq t_2$ or the form $t_1 \equiv t_2$ (where the terms t_1 and t_2 must be of the same type). *Formulae* will be atomic or built from atomic ones by means of conjunction, implication and universal quantification. For example, $\forall x\ y.\ (x+y \equiv y+x \wedge (x>0 \sqsubseteq y<(y+x)\ ))$ is a formula (a true one, as it happens).

Sentences, proofs and theorems are defined in the usual

(first order predicate calculus) way.

(d) An Example

We give below a set of four axioms (formulae) which introduce 3 constants (an individual, 0, and two functions *succ* and *pred*) and completely characterise a domain of natural numbers D_{nat}.

$$\partial(0{:}nat) \equiv \mathbb{T}$$

$$pred(0) \equiv \bot$$

$$pred(succ(x)) \equiv x$$

$$[\mu G.[\lambda x.\ x{=}0 \Rightarrow 0 \mid succ(G(pred(x)))]] \equiv [\lambda x.x]$$

The functions ∂ and $=$ (not to be confused with $\equiv$) are previously axiomatised. ∂ is a definedness predicate and true ($\mathbb{T}$) for elements which not strictly dominated by other elements. $=$ gives $\mathbb{T}$ or $\mathbb{F}$ (as appropriate) when comparing defined elements and $\bot$ otherwise. (Clearly, $x{=}y \equiv \mathbb{T}$ entails $x \equiv y$.)

6. IN CONCLUSION

Denotational semantics is now thought of in terms of Scott logics and the Scott/Strachey paper [16] is essential reading for the serious student. The area is presented in more detail in [17] and amplified enormously by Milne in two volumes [18]. Scottery has been the semantic vehicle for formalisation of LISP, ALGOL 60 and Pascal. References to these experiments appear in [19].

Scott logics have also become an important tool in other approaches to semantics. Milner in [19] illustrates the use of LCF by giving, as well as a denotational semantics for a simple language. LCF has also been used recently [20] to give an operational semantics of a typical computer. It is also easy to see that LCF could be used in place of first order logic in a Hoare-style propositional-semantics. Although some advantages are visible no investigations have been reported.

REFERENCES

[1] J.V. Garwick, "The Definition of Programming Languages by their Compilers", Formal Language Description Languages for Computer Programming, T.B. Steel (Ed), North Holland, 1966.

[2] P.J. Landin, "The Mechanical Evaluation of Expressions", Computer Journal, 6, 1964, pp.308-320.

[3] P.J. Landin, "A Formal Description of Algol 60", Formal Language Description Languages for Computer Programming, T.B. Steel (Ed), North Holland, 1966.

[4] P.E. Lauer, "Formal Definition of ALGOL 60", IBM Vienna Labs., Report TR 25.088, 1968.

[5] P. Lucas and K. Walk, "On the Formal Description of PL/I", Annual Review in Automatic Programming, Vol. 6, Part 3, 1969.

[6] R.W. Floyd, "Assigning Meanings to Programs", Proc. Amer. Math. Soc., Symposia in Applied Math. 19, 1967.

[7] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", Comm. A.C.M. 12, (1969), No. 10.

[8] E.W. Dijkstra, "A Discipline of Programming", Prentice-Hall, 1976.

[9] C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal", International Symposium on Theoretical Programming, Lecture Notes in Computer Science 5, Springer-Verlag 1974.

[10] J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine - Part 1", C.A.C.M. 3 (1960), No. 4.

[11] C. Strachey, "Towards a Formal Semantics", Formal Language Description Languages for Computer Programming", T.B. Steel (Ed), North Holland, 1966.

[12] R. Milner, "Logic for Computable Functions: Description of a Machine Implementation", A.I. Memo 169, Computer Science Dept., Stanford University, 1972.

[13] R. Milner, "Models of L.C.F., AI Memo 186, Computer Science Dept., Stanford University, 1973.

[14] R. Milner, F.L. Morris and M. Newey, "A Logic for Computable Functions with Reflexive and Polymorphic Types", Proc. Conf. on Proving and Improving Programs, Arc-et-Senans, 1975.

[15] M. Gordon, R. Milner and C. Wadsworth, "Edinburgh L.C.F.", Technical Report, Dept. of Computer Science, Edinburgh University, 1977.

[16] D. Scott and C. Strachey, "Towards a Mathematical Semantics for Computer Languages", Proc. Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, 1971. Also Technical Monograph PRG-6, Programming Research Group, Oxford University.

[17] R. Tennent, "The Denotational Semantics of Programming Languages", Comm. A.C.M. Vol. 19, No. 8, 1976.

[18] R. Milne and C. Strachey, "A Theory of Programming Language Semantics", Chapman and Hall, 1976.

[19] R. Milner, "Program Semantics and Mechanised Proof", Foundations of Computer Science 2,

[20] M. Newey, "Proving Properties of Assembly Language Programs", Information Processing 77, (to appear).

12

On the Semantic Characterisation of Some Advanced Control Structures

Paul A. Pritchard

1. INTRODUCTION

The problem addressed in this paper is that of providing Hoare-style input/output semantic characterisations for some control structures, variants of which have appeared in relatively recent programming languages, viz. Algol 68 and Simula 67. Such characterisations provide techniques for the formal definition of programming language semantics and for proving program correctness, as well as insights into good programming language design and usage. The presentation is necessarily concise; the reader is referred elsewhere [12, 13] for details.

2. HOARE-STYLE SEMANTICS

Hoare [8] constructed an axiomatic program logic which enables the proving of correctness assertions of the form

$$P\{A\}Q\ .$$

This has the interpretation "if P is true immediately before execution of the program segment A, then Q holds if and when A terminates". Since this method doesn't deal with proofs of program termination, it is called a partial correctness method. The documentation assertions P and Q come from some subsidiary theory giving the properties of the basic expression-forming components of the programming language. This is generally taken to be an applied, first-order predicate calculus, but stronger systems are needed for "complete" semantic characterisations [15].

The deductive system consists of general logical rules of inference used to link and modify the program assertions, axioms for the basic statement types, and proof rules for the structured statement forms. These proof rules are written with the premises above and conclusion below a dividing line. The proof rules for the classic "Structured Programming" [6] control structures are given below.

Concatenation:
$$\frac{P\{A_1\}Q,\ Q\{A_2\}R}{P\{A_1;\ A_2\}R}$$

Selection:
$$\frac{P \wedge b\{A_1\}Q,\quad P \wedge \sim b\{A_2\}Q}{P\{\underline{\text{if}}\ b\ \underline{\text{then}}\ A_1\ \underline{\text{else}}\ A_2\}Q}$$

Iteration:

$$\frac{P \wedge b\{A\}P}{P\{\underline{\text{while}}\ b\ \underline{\text{do}}\ A\}P \wedge \sim b}$$

These simple proof rules enable quite manageable correctness proofs to be given for programs using these control structures. Furthermore, since the property of correctness of a program is second in importance only to the program's existence (pace philosophical diputes about whether existence is a property), it is felt that the simplicity of a programming language feature's proof rule correlates with the "intellectual manageability" of that feature. However this claim needs to be modified in view of the simply stated rules for the goto statement given in [10]. The simplifity of these rules is illusory because the syntax-directed nature of correctness proofs is violated. The moral here is that care must be taken in using the simplicity of proof rules as a guide to good programming language design - the final arbiter must be the proofs themselves.

3. EXPRESSION LANGUAGE CONTROL STRUCTURES

A statement-oriented programming language can be generalised by associating (where convenient) a value with each imperative statement form, and allowing the resulting construct to be used as an expression of the appropriate type. A language based on such generalised expressions is called an expression language - a notable example is Algol 68 [18].

It is desirable to give proof rules for expression language constructs for the reasons advanced in §1. In order to do so an extended notation is required for the documentation assertions. These assertions will be denoted P*,Q*,... and can take two forms:

(i) P : an assertion in the underlying logical theory.

(ii) $\bigvee_j P_j \wedge \underline{st}(p_j)$: an assertion in an extended theory, stating that for some j, P_j is true of the state of computation and the value p_j has been "stacked", i.e. is available for use as an operand (and possibly as an operator, but such refinements will not be discussed here). The disjunction over j is needed for expressions which can conditionally stack different values.

It is convenient to introduce a combinatory device to manipulate assertions of the second form. Let pred be a one-place predicate, and $\mathcal{P}$ a finite set of ordered pairs (P_j, p_j) of assertions and values. Then

$$\mathcal{P}\ \Diamond\ \underline{\text{pred}}$$

stands for

$$\bigvee_j P_j \wedge \underline{pred}(p_j).$$

Thus, for instance, the second form of assertion is written

$$\mathcal{P} \lozenge \underline{st}\ .$$

We now give proof rules for simple expression language generalizations of the control structures in §2. Capitals E,B,... denote generalized expressions stacking values of the appropriate type; lower case letters are used for values, i.e. expressions in the underlying applied logical theory.

Concatenation:

$$\frac{P\{E_1\}Q,\ Q\{E_2\}R^*}{P\{E_1;E_2\}R^*}$$

The value, if any, of the combined expression is that returned by the second component. The use of Q rather than Q* ensures that the value of E_1 is ignored.

Selection:

$$\frac{P\{B\}Q\lozenge\underline{st},\quad Q\lozenge\underline{id}\{E_1\}R^*,\quad Q\lozenge\underline{not}\{E_2\}R^*}{P\{\underline{if}\ \ B\ \ \underline{then}\ \ E_1\ \ \underline{else}\ \ E_2\ \ \underline{fi}\}R^*}$$

The two predicates introduced are

$$\underline{id} : \lambda b.b$$

$$\underline{not} : \lambda b.\neg b$$

The value of the generalized conditional is that of the chosen component.

Iteration:

$$\frac{P\{B\}Q\lozenge\underline{st},\quad Q\lozenge\underline{id}\{E\}P}{P\{\underline{while}\ \ B\ \ \underline{do}\ \ E\ \ \underline{od}\}Q\lozenge\underline{not}}$$

P is the invariant assertion; no value is returned.

These rules are simple in both their statement and usage, and yet are quite powerful. The selection rule covers both conditional expressions and statements (and anything in between) and the iteration rule encompasses rules for the "while", "repeat" and "n + ½" loops [10] as well as more complex forms [13]. It seems that these expression language features have strong claims to "intellectual manageability".

4. BINARY OPERATORS IN AN EXPRESSION LANGUAGE

The complexity of proof rules for expression language constructs seems to increase significantly for those constructs which require the evaluation of several component expressions before using their value in unison. Situations where this phenomenon occurs are the evaluation of the operands of a binary operator and the evaluation of an actual parameter of a procedure call.

Consider a binary operator term

$$E_1 \cdot E_2$$

Now, if the ALGOL 68 policy of collateral elaboration of E_1 and E_2 is adopted, the following rule applies:

$$\frac{P\{E_1\}\mathcal{Q}\Diamond\underline{st},\ P\{E_2\}\mathcal{R}\Diamond\underline{st},\ Q_j \wedge R_k\{q_j \cdot r_k\}S^*\ \forall j,k}{P\{E_1 \cdot E_2\}S^*}$$

provided that neither E_1 nor E_2 side effects the other. However, the proviso on this rule is virtually impossible to formalise in a machine-independent fashion, and thus the complete scope of applicability of the rule is unclear. By making assumptions about "reasonable implementations", it may be possible to dispense with provisos and give a semantics in which all possible consequences of a collateral elaboration can be deduced [14]. Such a semantics would require a relational calculus and an enormous increase in complexity. The programmer should instead adopt an incomplete but simple and safe rule by observing this stronger proviso:

> no variable appearing in both E_1 and E_2 should be subject to change (by assignment or procedure call) in E_1 or E_2.

This proviso admits of a simple syntactic check.

Suppose an alternative policy of left to right evaluation is chosen. Then the proof rule becomes

$$\frac{P\{E_1\}Q\Diamond\underline{st}, \quad Q_j\{E_2\}R_j\Diamond\underline{st}\ \forall_j, \quad R_{jk}\{q_j\cdot r_{jk}\}S^* \quad \forall_j,k}{P\{E_1\cdot E_2\}S^*}$$

provided no q_j contains variables subject to change by E_2. This rule is more robust because the proviso can always be satisfied by the logical device of introducing Skolem constants. That is, the result of evaluating the expression is always defined. Furthermore, if the axiomatic semantic specification given by the rule is used as the absolute language definition, then the advantages of optimization (typically of arithmetic expressions) and real parallel evaluation touted for the collateral approach are also applicable for the left-right strategy - the compiler has only to check the syntactic proviso given above. This illustrates an advantage of non-operational semantic definition methods.

Finally, a comment is offered on the complexity of these operator rules. Situations which generate large numbers of premises are those where selection expressions are deeply nested to give very compact code. There is independent evidence [17] to suggest that such situations are difficult to understand and hence should be avoided. The operator rules do not become unwieldy if a sensible programming style is employed.

5. COROUTINES

As a final application of Hoare's semantic methods, we investigate coroutines. Those readers who are not familiar with this control structure, and/or unconvinced of its utility, are referred to the papers and unsolicited testimonials [4,7,9,11].

Simula 67 [5] is perhaps the only well-known language which supports a form of coroutines. Its coroutines are dynamically created <u>class</u> objects and communicate via three sequencing operations: <u>call</u>, <u>detach</u> and <u>resume</u>. The first two are used in conjunction to give semi-coroutine linkage, which essentially provides coroutines as generators. The <u>resume</u> operation gives the more familiar symmetric coroutine linkage.

There have been two attempts in the literature [2,3] to give Hoare-style proof rules for coroutine systems based on that of Simula 67. Both limit sequencing to the <u>call</u>/<u>detach</u> mechanisms, and give formal rules only for the case of a single semi-coroutine interacting with a "master" routine. There seems little hope of extending this work to give a comprehensive Hoare-style semantic definition of Simula 67's coroutine facility - the dynamic creation of coroutines and the run-time restrictions of the interaction of the three sequencing operations are very resistant to a non-operational proof-theoretic approach. The difficulty here is further evidenced by the three mutually inconsistent descriptions [4,5,16]. We note that formal semantics can be given in

LCF[1], but at present such definitions are far removed from the intuitive and verification-oriented nature of Hoare's method.

An alternative coroutine system is proposed in [11], mainly to achieve sensible implementation compromises. The proposal is for recursive co-operations of a fixed number of coroutines communicating via resume/resume sequencing. It is possible to give a simple Hoare-style semantic characterization of this feature by extending the rule given in [12] to accommodate recursion. The co-operation is named c, has communication variables $\tilde{x}$ initialized by the statement S, and coroutines c_i with bodies A_i. It is called via an activate statement which is also available in the coroutine bodies A_i. The rule becomes

$$c \ \underline{\text{pool}} \ \tilde{x};\ S \ \underline{\text{coroutines}} \ c_1{:}A_1;c_2{:}A_2;\ldots;c_n{:}A_n \ \underline{\text{end}}$$

$$\frac{P_j\{\underline{\text{resume}}\ c_j\}P_i \text{ for } 1\leq j\leq n, j\neq i,\ P\{\underline{\text{activate}}\ c\}R \vdash P_i\{A_i\}R \text{ for } 1\leq i\leq n}{P\{\underline{\text{activate}}\ c\}R}$$

where (i) no P_i may contain free occurrences of a local variable of a coroutine;

(ii) neither P nor R may contain free occurrences of a variable in $\tilde{x}$ or a local variable of a coroutine.

Further rules are needed to pass local and communication information across resume and activate statements, and to add a parameters facility. But these present no difficulties. The rule as it stands is certainly incomplete, but it does suggest that the proposed system is well designed and worthy of serious consideration.

6. CONCLUSIONS

Some advanced control structures have been examined with a view to providing Hoare-style semantic characterisations for them. It is hoped that this has given the reader some appreciation of the insights into the design and use of programming languages that this particular area of formal investigation can provide.

7. REFERENCES

[1] L. Aiello, M. Aiello, G. Attardi, P. Cavallari, and G. Prini, Formal definition of semantics of generalized control regimes, Lecture Notes in Computer Science 45, 1976, 173-179.

[2] M. Clint, Program proving : coroutines, Acta Informatica 2, 1973, 50-63.

[3] O.-J. Dahl, An approach to correctness proofs of semicoroutines, Lecture Notes in Computer Science 28, 1975, 157-174.

[4] O.-J. Dahl, and C.A.R. Hoare, Hierarchical program structures, Structured Programming (Academic Press), 1972, 175-220.

[5] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, Simula 67 Common Base Language, Pub. No. S-22, Norwegian Computing Center, 1970.

[6] E.W. Dijkstra, Notes on structured programming, Structured Programming (Academic Press), 1972, 1-82.

[7] R.E. Griswold, Extensible pattern matching in Snobol 4, Procs. of ACM Nat. Conf., 1975, 248-252.

[8] C.A.R. Hoare, An axiomatic basis for computer programming, C.A.C.M. 9, 1969, 576-580, 583.

[9] D.E. Knuth, A review of "Structured Programming", Report STAN-CS-73-371, Comp. Sci. Dept., Stanford University, 1973.

[10] D.E. Knuth, Structured programming with go to statements, Computing Surveys 6, 1974, 261-301.

[11] B. Krieg, A class of recursive coroutines, IFIP 74 Proceedings (North-Holland), 1974, 408-412.

[12] P.A. Pritchard, A proof rule for multiple coroutine systems, Information Processing Letters 4, 1976, 141-143.

[13] P.A. Pritchard, Program proving - expression languages, IFIP 77 Proceedings, to appear.

[14] R. van Vliet, A problem in collateral elaboration, Report IW 44, Mathematisch Centrum, Amsterdam, 1976.

[15] M. Wand, A new incompleteness result for Hoare's system, Tech. Report No. 35, Computer Science Dept., Indiana University, 1976.

[16] A. Wang, and O.-J. Dahl, Coroutine sequencing in a block-structured environment, B.I.T. 11, 1971, 425-449.

[17] G.M. Weinberg, D.P. Geller, and T. W-S Plum, IF-THEN-ELSE considered harmful, Sigplan Notices 10, No.8, 1975, 34-44.

[18] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker (eds.), Revised report on the algorithmic language ALGOL 68, Acta Informatica 5, 1975, 1-236.

13
Towards a Programming System

Ken A. Robinson

ABSTRACT:

Programmers today, using standard software products, modify their programs using what could be described as first generation techniques.

This situation is due to the fact that programming language designers and their associated compiler implementors, in general, ignore the fact that a computer program is an object which will undergo, sometimes extensive, modification.

This paper discusses some of the facilities that are required for program modification and develops a proposal for a programming system for the programming language Pascal.

1. INTRODUCTION

Programming as it is practised today is carried out with a miscellany of software tools, many of them forced into service for purposes for which they were not designed, or in which they are not particularly effective.

Integrated programming systems consisting of tools designed for program development, program verification, program testing and program performance measurement are required. The area addressed here is that of program development.

Programmers developing computer programs, using what might be termed "standard" software, use a form of organization which has not changed essentially over the last 20 years.

The typical programmer will at some stage use a compiler to translate from a source program (a test string in some programming language) to some other form (a string of "machine" instructions) which will be interpreted by either hardware or software. The program presented to the compiler is the object of particular interest in the subsequent discussion. This program may be a new program or more usually a modification or extension of a program created earlier. How that earlier program was modified to its current form is usually left to the devices of the individual programmer.

It can be safely asserted that compilers spend most of their time compiling modified programs and yet this process of program modification is ignored by most compiler implementors and certainly by most language designers. It will be argued here that program modification is a process that should not be ignored by compiler implementors and possibly should also be the concern of language designers.

The language Pascal [1] and the Pascal compiler implemented for CDC 6000 series and Cyber machines (described within [1]), which will be referred to as the Pascal 6000 compiler, have been chosen for particular case studies.

It has to be emphasized that Pascal has been chosen as a representative of the best current general-purpose programming languages in fairly widespread use and Pascal 6000 as a representative of recent, well engineered compiler implementations. Any critical comment which follows is directed as both Pascal and Pascal 6000 as representatives; that is, the criticism is of current accepted practice.

2. PRIMARY TOOLS AVAILABLE TO A PROGRAMMER

1 A program updating utility.
This program is typically a line editor of some sort, allowing the insertion, replacement or deletion of lines in a text file. For programming languages such as Pascal the concept of "line" is not very meaningful.

2 A text editor.
This program typically allows the insertion, replacement or deletion (plus other more specialized operations) of sub-strings, as well as lines. The editor, being general purpose, is usually ignorant of any structure in the strings being manipulated.

3 A compiler.

3. THE FUNCTIONS OF A COMPILER

The following functions of a compiler are prominent:

1 To translate from a text program to efficient machine code. This task was dominant in the design of the first FORTRAN compiler (it also influenced the design of the language) [2] and has been important to some degree ever since. It is an important consideration in the design of the Pascal compiler and even the design of the Pascal language [4].

2 To diagnose syntax errors.
Detection of syntax errors is essential to prevent a compiler from aborting and for some compilers this task is undertaken for that reason alone; others also regard it as part of the task of checking a program for correctness.

3 To diagnose semantic errors.
This task, which is very restricted for many compilers due to the design of the associated language, is very important for any Pascal compiler. The Pascal language has been designed with very tight binding of data type attributes, allowing many statements to be checked for correctness (or rather incorrectness) during compilation. The programmer is

thereby alerted before a program is executed and additionally the compiler may dispense with some run-time checks.

4. INCREMENTAL PROGRAM MODIFICATION AND INCREMENTAL COMPILATION

Program modification requires the insertion, replacement or deletion of "program elements" such as procedures, declarations, statements, expressions, variables or constants and thus incremental modification requires increments ranging from procedures down to variables and constants.

If we are interested in compilation of a program into "high-quality" machine code then the smallest independent code segment which can be easily modified is usually the procedure. Thus it is suggested the "incremental compilation" be limited to increments no smaller than an individual procedure.

5. A PROGRAM EDITOR

As has been observed earlier the conventional line editor or text editor does not provide direct access to the "program elements".

What is proposed is a syntax driven editor which edits a tree representation of an abstract program (similar to that used by the Vienna Definition Language [3]). The input to the editor from the programmer would consist of a text description of modifications to be applied to the abstract program. Notice that in one extreme case the abstract program is initially empty and the modification described by the text is a complete program.

6. A PASCAL PROGRAM EDITOR

A Pascal program can be represented by the tree shown in fig. 1.

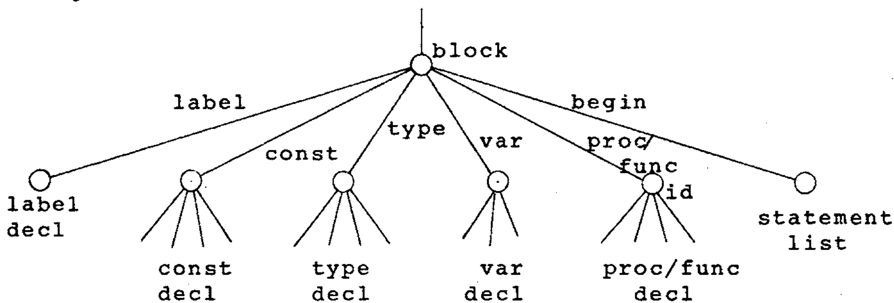


fig. 1

The keywords label, const, type, var and begin are used as selectors, i.e. each keyword may appear as a label on a branch in the tree and the label is used to select a particular sub-tree. Similarly the names of procedures/functions are used as selectors to select procedure sub-trees and the names of

variables are used to select type sub-trees.

Statements within a statement sequence may be selected by the ordinal numbers of their position within the sequence or they may be selected by a structured pattern similar to the patterns used by text editors to select sub-strings; the difference is that a structured pattern selects sub-trees.

Sample modification text (input to the editor by the programmer):-

```
.PROG.PROC1.PROC2
var
  x: real;
begin
´nn := n´ s ´n´n+1´
´z := f(y)/g(x)´ s ´<s>´if x<> 0 then <s> else z := 1´
```

The above sample modification segment shows first the selection of the path via program PROG, procedure PROC1 and procedure PROC2. That is we have selected for editing procedure PROC2 which was declared within PROC1 which in turn was declared within PROG.

Next <u>var</u> selects the variable declaration sub-tree and ´x: real;´ either replaces a previous declaration or inserts a new declaration for x.

Next <u>begin</u> selects the block body of procedure PROC2.

Within the block body we see:

- a statement ´nn := n´ being changed to the statement ´nn := n+1´;

- an assignment statement ´z := f(y)/g(x)´ is imbedded is a new alternative statement - the pattern <s> matches a complete statement.

<u>Note</u>: The form of the substitution command used above is:

´ s1 ´ s ´ s2 ´ s3 ´

where:

s1 is a statement selector;
s2 is a subtree of statement selected;
s3 is a replacement for s2.

The above example is intended to suggest, tentatively, the form and facilities to be offered by the editor.

7. COMBINING THE EDITOR WITH THE COMPILER

The process of building and modifying the tree representation of the abstract program obviously requires much of the syntactical analysis performed by a compiler. It would therefore seem profitable to either combine the process of program modification with the process of compiling, or to

partition the process of compiling into two serial processes as shown in fig. 2.

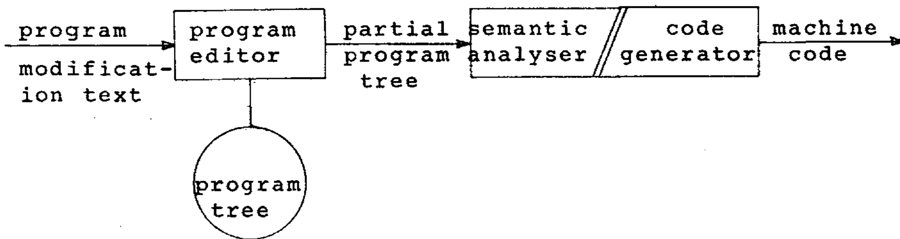


fig. 2

The program editor edits the complete program tree and passes on to the semantic analyser a partial program tree representing only those procedures which have been modified (or those whose environment has been modified). The semantic analyser and the code generator carry through with the remainder of the compilation process.

The system should also be capable of producing an appropriately formatted listing of the program. This feature would overcome another problem which occurs when a text program has been modified, namely the lexical structure no longer reflects the current program structure.

Notice that the above function implies that the abstract program must have provision for retaining program documentation. It is proposed that each node in the program tree should have provision for documentation.

8. INDEPENDENT COMPILATION OF PROCEDURES

It has been suggested above that the compiler should allow independent compilation of procedures. This suggestion is made both for reasons of convenience and efficiency. During the development of a large program only a small number of procedures are at any time undergoing change and it is very expensive to continuously compile the complete program.

9. COMPILING PROCEDURES SEPARATELY WITH THE PASCAL 6000 COMPILER

The Pascal 6000 compiler has a facility for declaring procedures to be external, thus implying the possible existence of independently compiled procedures. The facility appears to have been added on as an afterthought; a concession to allow the occasional independent compilation of a procedure. The first Pascal language report [4] would seem to expressly forbid the concept of independent compilation.

Consider some of the problems encountered when compiling procedures independently with the Pascal 6000 compiler:

1 Type and mode of actual procedure parameters cannot be checked reliably against the type and mode of the formal parameters.

Procedures which are compiled independently are required to be declared within any program in which they are to be called, by an external declaration which consists of a replication of the procedure heading, showing the procedure name and parameter declarations, followed by the keyword extern in place of <block>. Thus there will exist at least two independent declarations of the formal parameters of a separately compiled procedure: one in the actual procedure declaration and at least one in an external procedure declaration. These two declarations cannot be checked by the compiler for consistency and there are no run-time checks within the called procedure, since they are unnecessary when the actual arguments have been checked against the formal parameter declarations. It is very easy for errors to occur: consider the fragments shown in fig. 3 which may lead to an error at run-time which is very difficult to diagnose.

```
procedure inc(var n: integer):
begin
  n := n+1
end {inc};
```

fig. 3a Actual procedure declaration.

```
procedure inc(n: integer):  extern;

 ...

  inc(i);

 ...
```

fig. 3b External declaration and call.

Challenge: The reader, if familiar with the Cyber architecture, might like to contemplate the consequence of the call ´inc(i)´ shown above for the case when i=1 !

Solution:
The problem would cease to exist under the proposed program modification system since the programmer does not have to replicate any declarations.

2 A procedure may run in the wrong environment.
The actual procedure declaration may be placed at a different nesting level to any of the other external procedure declarations: consider the example shown in fig. 4.

```
procedure A;
  procedure B;

  ...

  end {B};
end {A};
```

fig. 4a Actual compilation of procedure B.

```
procedure B; extern;
procedure A;
  ...
  B;
  ...
end {B};
```

fig. 4b External declaration and call of procedure B.

Solution:
The problem would cease to exist under the proposed program modification system since the environment of any procedure would be determined from the program tree.

3 Procedure name conflicts.
When compiling independently the first 7 characters of actual procedure names are used for internal identification of procedures and thus the first 7 characters of all procedure names must be distinct. Normally, when compiling complete programs, the compiler generates a sequence of distinct internal procedure names. When procedures are compiled independently it is obviously not possible, in most cases, for the compiler to detect internal name conflicts, and such conflicts may lead to very obscure behaviour when a program is executing!

Solution:
The compiler must generate "unique" internal names for procedures.

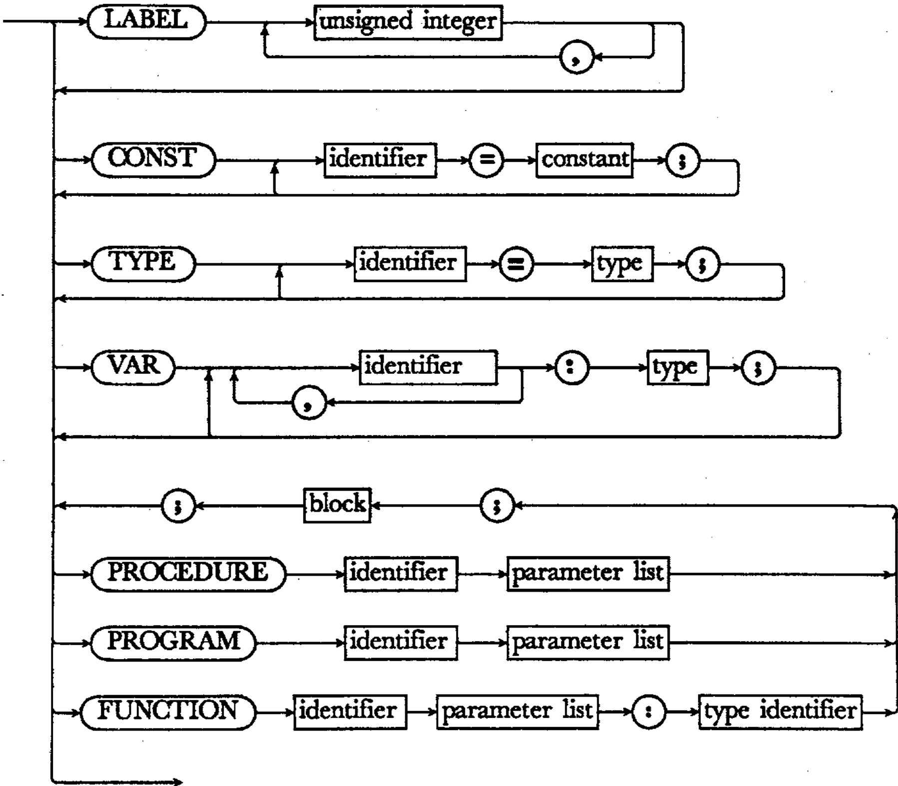
4 The compiler compiles programs not procedures.
To compile a procedure independently the procedure must be dressed up to look like a program by imbedding the procedure within a dummy program, even when the procedure does not require access to global declarations or variables. What is in evidence here is not only the ad hoc nature of the facility provided for independent compilation but also and inadequacy in the language design: the language does not cope with the concept of an environment external to the program. As a side effect notice that an external library procedure is executed in technically the wrong environment. Such a procedure has access to the program variables even though it does not (should not) require such access.

Solution:
The following suggested syntax change to Pascal should be adopted.

10. A SUGGESTED SYNTACTIC CHANGE TO PASCAL

A change to the syntax is suggested to allow declarations to occur at level 0 and to change the program header so that it is no different to the procedure header. A program obviously resembles a procedure, the main difference is that it is the entry point for execution. The ad hoc and inconsistent form of the program header has been frequently been commented upon. The syntax revision is shown in fig. 5. Notice that a new non-terminal, <module>, has been introduced. A module defines an object which can be compiled separately.

blockhead



block

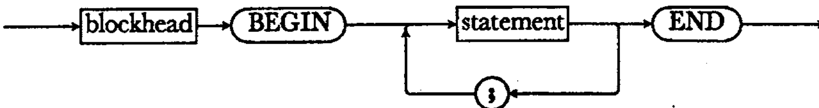


fig. 5 Revised syntax of Pascal <block>.

A semantic rule would restrict program declarations to level 0 and the number of such declarations to one only.

11. EFFECT OF THE SYNTAX CHANGE

Procedures and/or functions could be declared and compiled as true external procedures/functions.

File parameters of a program would have to be declared as var parameters, as for a procedure, and the type of any file other than text would have to be declared as level 0 as shown in fig. 6.

```
program inventory(input,output,catalog);
type
  book = record
           author:
           title:
          ...
         end;
var
  catalog: file of book;
```

fig. 6a Declaration under current syntax.

```
type
  book = record
           author:
           title:
           ...
         end;
  bookfile = file of book;
program inventory(var input,output: text;
                  var catalog: bookfile);
```

fig. 6b Declaration under revised syntax.

Given an environment at level 0 the files input and output could be pre-declared as variables at that level. This would allow input and output to be omitted from the program header declaration with the possible implication that such a program cannot be called with file arguments.

A program could be passed arguments other than file arguments.

Note: Under the new syntax the earlier suggestion, that the compiler should generate "unique" internal procedure names, would apply only to procedures declared at level 1 or higher. All procedures, and the main program, which are declared at level 0 should retain their own name for the internal name.

12. CONCLUSION

Programs written in a modern programming language, such as Pascal, are structured objects and program modification should be viewed as structural modification.

In order to modify such a structured object we have argued that a more advanced tool than a general purpose line or text editor is required.

A special program editor is required, but rather than create another utility program it has been suggested that the function of the compiler should be expanded to incorporate program editing, producing a system which begins to approximate a programming system.

ACKNOWLEDGEMENTS

I would like to thank David Carrington and Ian Hayes for their assistance in proof-reading this paper and for the very valuable discussions I have had with them on many of the matters discussed in the paper.

REFERENCES

[1] Jensen and Wirth: Pascal user manual and report. Springer-Verlag 1975.

[2] Backus and Heising: FORTRAN. IEEE Trans on Elect Comp, EC-13, 4 1964, 382.

[3] Wegner: The Vienna definition language. ACM Computing Surveys, 4,1, 1972, 5-64.

[4] Wirth: The programming language PASCAL. Acta Informatica 1, 1, 1971, 35-63.

14

Recursion as an Alternative to Back-tracking and Nondeterministic Programming

Jeff S. Rohl

Programming problems have always been classified, and one of the earliest classifications was that of combinatorial problems. These problems have always appealed to teachers and I well remember setting one in the first year of the first Computer Science degree course in England in 1965 [1]. More recently Wirth [7] has used such a problem in his introductory text, a problem we will use later in this paper.

The time-honoured problem in this area is the n-queens problem: that of determining the ways in which n queens may be placed on an n by n chessboard so that no queen is under attack from any other; and the time-honoured method of solution is backtracking. I remember, too, how those first students struggled to master the technique. As those who've written back-tracking problems will know, it's a mentally demanding exercise (especially if we wish to constrain ourselves to the basic constructs of concatenation, alternation and interation).

We present here a different solution and accordingly we will use different examples as well.

1. GENERATING COMBINATIONS

We start by considering what is probably the simplest combinatorial procedure: one which produces the combinations of the first n natural numbers taken r at a time. The order of generating combinations is irrelevant, but since they must be produced in some order we choose to produce them in "ascending order" as illustrated in Figure 1.

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Fig. 1: The order of producing the combinations of the integers 1 to 5 taken 3 at a time.

We observe that the first element of the combination ranges from 1 to that number (here 3) which allows one choice for each of the subsequent r - 1 elements (i.e. n-r+1); that the second element ranges from one more than the (current) value of the first element to n-r+2; ... ; that the kth element ranges from one more than the $(k - 1)^{th}$ to n-r+k; ... .

Let us suppose that the elements are held in the integer array c[1:r]. The basic structure is certainly that of a nest of loops and we can conceive of a solution like that of Fig. 2, where to help the exposition an extra element c[0] has been introduced.

```
c[0] := 0;
for c[1] := c[0]+1 step 1 until n-r+1 do
     for c[2] := c[1]+1 step 1 until n-r+2 do
                  :
                  :
         for c[r] := c[r-1]+1 step 1 until n do
              comment combination available
```

Fig. 2: The loop structure of the combination problem

The problem of course is represented by the three dots: how do we write a nest of loops of variable (and even varying) depth? This problem presented itself originally in more complicated situations where an optimum was sought, and where solutions had to satisfy some (often demanding) criteria. We consider 2 such problems later.

The original solution attributed by Golomb and Baumert [5] to Lehmer was the back-tracking referred to earlier. This was effectively the simulation of a nest of loops by two loops (sometimes degenerate, sometimes merged into one) embedded within an outer loop. One inner loop simulated the initialisation phase of the simulated loops; the other simulated the incrementing and testing phase of the simulated loops. We give an example in the appendix, but it is sufficient to say here that the difficulty of writing such programs lies in the details of the initialisation and termination of the two inner loops.

One attempt to overcome this problem was the development of non-deterministic programming by Floyd [4]. An example of this is given in the appendix too. A non-deterministic program cannot be obeyed directly: it must first be pre-processed to produce a normal (deterministic) program. This is because in a non-deterministic program only the values over which a variable (such as an element of a combination) may range are specified, while the mechanism controlling the assignment of these values is implicit.

The technique we are going to consider achieves the same result directly by using recursion, one recursive activation of a procedure corresponding to each nesting of the loop. We give all future examples as Algol procedures indicating by a comment where the results are available even if it is syntactically invalid. Fig. 3 gives a procedure for generating the combinations.

```
procedure generate combinations (n,r);
value n, r; integer n, r;
     begin
     integer array c[0:r]

     procedure choose (k);
     value k; integer k;
          for c[k] := c[k-1]+1 step 1 until n-r+k do
          if k ≠ r then choose (k+1)
          else comment combination available;

     c[0] := 0;
     choose (1)
     end
```

Fig. 3: A recursive procedure for generating combinations

When interpreting this (and subsequent procedures) it is fruitful to think of each activation of choose as associated with a given element. Thus the first activation ensures that c[1] takes on values from 1 to n-r+1; and as it assigns each value in turn it calls on a second activation to ensure that c[2] takes on values from c[1]+1 to n-r+2; ... ; and as it assigns each value in turn, it calls on a kth activation to ensure that c[k] takes on values from c[k-1]+1 to n-r+k; ... ; and as it assigns each value in turn, a combination is generated. [A picturesque view is to regard the activations of the recursive procedure as a hierarchical structure of automatons each of whom obeys his immediate superior and commands his immediate inferior. The automaton at each end of the hierarchy is special, the general has to be externally commanded, and the private has all the dirty work to do.]

The key to the success of this technique lies in the fact that the recursive procedure is written by concentrating on the operations involved at the kth level. Although the components of the kth activation are obeyed sequentially they are interspersed with the components of lower activations. (A self-co-operating sequential procedure?) The difficulty with writing back-tracking procedures is that the programmer's concentration is on the sequential actions of the total program. The comparison between back-tracking and recursion in combinatorics is precisely that between using queuing procedures embedded in Fortran, say, and using Simula or Sol in simulation.

2. GENERATING PARTITIONS

The procedure for generating combinatorics is probably the simplest possible combinatorial one. Although from call to call the depth of recursion (r) varies, nevertheless it is constant during a call, and it is easy to conceive of r nested loops. We consider now a problem in which the recursive depth varies during the call so that it's more difficult to conceive of the appropriate nest of loops. However, if we ignore the nest of loops, regard recursion as the fundamental rather than the derived technique, and follow the technique described above, then that difficulty disappears.

The partitions of an integer n are the sets of integers whose sum is n. Thus Fig. 4 gives the partitions of 5.

```
5
4 1
3 2
3 1 1
2 2 1
2 1 1 1
1 1 1 1 1
```

Fig. 4: The 7 partitions of 5 in the order produced by the program of Fig. 5.

Suppose we produce them in the order of Fig. 4. Then a simple analysis shows that the first element of the partition ranges from n down to 1; that for a given first element the second element ranges down to 1 from the smaller of the first element and the residue of n (i.e. what is left of n after the choice of the first element) ... , and in general that for a given choice of the first, second, ... k^{th} elements, the value of the $k + 1^{th}$ ranges from the smaller of the kth element and the residue, down to 1. The recursion stops when the residue is 0.

If the elements are referred to as p_1, p_2, ... we can express this upper limit more formally as:

$$\min \left\{ p_{k-1},\ n - \sum_{i=1}^{k-1} p_i \right\}$$

Suppose we generate the elements in an integer array p and augment it by p[0] which is identically equal to n the formula above holds for p[1] as well.

```
procedure generate partitions (n);
value n; integer n;
     begin
     integer array p[0:n];
     procedure choose (k);
     value k; integer k;
          begin
          integer residue;
                              k-1
          residue := n -       Σ   p[i]
                              i=1
          for p[k] := min (p[k-1], residue) step -1
          until 1 do
               if residue ≠ p[k] then choose (k+1)
               else comment partition available
          end of procedure "choose";
    p[0] := n;
    choose(1)
    end of procedure "generate partitions"
```

Fig. 5: A naive version in crypto-Algol of a procedure to generate partitions

I have described the procedure in Fig. 5 as naive since the residue is calculated at each recursive level. [This was done deliberately and discussion above was phrased in such a way that the procedure produced seemed natural.] As it stands each activation of choose has its own local variable, residue, which is calculated in a loop (corresponding to the Σ of Fig. 5). A more subtle approach then would be to recognise that a given choice of an element reduces the residue by that amount, and so make residue a parameter, each activation passing on to the next the updated value of the residue. Fig. 6 illustrates this.

```
procedure generate partitions (n);
value n; integer n;
     begin
     integer array p[0:n];

     procedure choose (k, residue);
     value k, residue; integer k, residue;
          for p[k] := min (p[k-1], residue) step -1
          until 1 do
               begin
               if residue ≠ p[k] then choose (k+1,
                                          residue - p[k])
               else comment partition available
               end of procedure "choose";

     p[0] := n;
     choose (1, n)
     end of procedure "generate partitions"
```

Fig. 6: A refined version of a procedure to generate partitions

In more complicated situations, (such as the n queens problem), the information to be transferred between activations (the state of the chessboard in the queens problem) is too extensive for it to be done efficiently. In this situation we use global variables, the effect of handling on an updated version being obtained by modifying the global variables before calling and restoring them after returning.

It is important to note that both versions work, and that the second is a refinement of the first. This will be our design technique. We will produce a naive version and then progressively refine it. It would have been nice to be able to call this process "step-wise refinement" but that phrase is already in use for what seems to me to be the step-wise development of programs.

One fascinating aspect of these procedures is the ease with which they can be modified to solve related problems. For example, suppose we want the partitions of n restricted so that no element of the partition is less than some limit ℓ. Fig. 7 gives an example.

```
8
6 2
5 3
4 4
4 2 2
3 3 2
2 2 2 2
```

Fig. 7: The partitions of 8 whose elements are $\geqslant 2$

Such a program can be produced merely by changing the lower limit of the loop from 1 to ℓ. Consider what happens if we activate level k with a residue of 1. (In the example of Fig. 7 this would have happened when p[1] = 7.) The body of the loop is skipped and so no partition stems from that path.

The two examples we've considered so far are relatively simple in that the criteria that solutions have to conform to can be expressed within the control structure. In general, the criteria are more extensive. Further more, there are in general three classes of problem: finding all solutions which satisfy a criterion, finding one solution which satisfies the criterion, and finding an optimal solution. We consider the 3 cases in turn.

3. GENERATING WIRTH SEQUENCES

Consider Wirth's problem [7]: Generate sequences of n characters, chosen from an alphabet of three elements (1, 2, 3) so that no two immediately adjacent substrings are equal. Fig. 8 gives some examples and counter-examples.

Acceptable	Unacceptable
1	
1 2	1 1
1 3	
1 2 1	1 2 2
1 2 3	
1 2 1 3	1 2 1 1
	1 2 1 2

Fig. 8: Some Wirth sequences and some unacceptable sequences

The basic strategy is trivial. Suppose we call the elements s_1, s_2, ... s_n. Then for any given choice of s_1, s_2, ..., s_k, the possible values of s_{k+1} are simply 1, 2 and 3. After s_n is chosen, the values s_1, ... s_n are checked for validity. (This is a naive solution!) Thus the activation at level k, has simply to ensure that s_k takes on values 1, 2 and 3, and as it assigns each value to call on the $(k+1)^{th}$ activation.

```
procedure generate Wirth sequences (n);
value n; integer n;
     begin
     integer array s[1:n];

     procedure choose (k);
     value k; integer k;
          for s[k] := 1 step 1 until 3 do
               if k ≠ n then choose (k+1)
               else if valid then comment sequence available;

     Boolean procedure valid;
          comment This procedure checks the validity of the
                  sequence by comparing all adjacent sub-
                  strings of all lengths up to n÷2;
     choose (1)
     end of procedure "generate Wirth sequences"
```

Fig. 9: A naive procedure for generating Wirth sequences

Clearly this procedure is woefully inefficient because it generates all potential sequences (here the elements of $\{1, 2, 3\}^n$) and tests each in turn (independently to the others). Thus it will generate for n = 5 the strings shown in Fig. 10.

```
1 1 1 1 1
1 1 1 1 2
1 1 1 1 3
1 1 1 2 1
    .
    .
    .
```

Fig. 10: The sequences generated by the naive program of Fig. 9 (n=5)

A better solution would be to refine the procedure as follows. The activation at level k still ensures that s_k takes on values 1, 2 and 3; but as it assigns each value, it checks that the assignment leaves the partial string $s_1 \ldots s_k$ valid, and only if it does is a call made on the $(k+1)^{th}$ activation. Since the empty string is valid we have no initialisation problems. Thus the partial strings tried are as shown in Fig. 11.

```
1                       Valid
1 1                     Not valid
1 2                     Valid
1 2 1                   Valid
1 2 1 1                 Not valid
1 2 1 2                 Not valid
1 2 1 3                 Valid
1 2 1 3 1               Solution
1 2 1 3 2               Solution
1 2 1 3 3               Not valid
1 2 2                   Not valid
1 2 3                   Valid
          .
          .
          .
```

Fig. 11: The partial Wirth strings considered by the program on Fig. 12 (n=5)

The refinement is quite simple as Fig. 12 shows. [To produce the procedure directly rather than going through the process of refinement is straight-forward too.]

```
procedure generate Wirth sequences (n);
value n; integer n;
     begin
     integer array s[1:n];

     procedure choose (k);
     value k; integer k;
          for s[k] := 1 step 1 until 3 do
          if still valid (k) then
               begin
               if k ≠ n then choose (k+1)
               else comment sequence available
               end of procedure "choose";

     Boolean procedure still valid (k);
          value k; integer k;
          comment This procedure checks that string in s[1] ...
                  s[k] is still valid, knowing that the string
                  is s[1] ... s[k-1] is valid;

     choose (1)
     end of procedure "generate Wirth sequences"
```

Fig. 12: A refined procedure for generating Wirth sequences

4. GENERATING ONE WIRTH SEQUENCE

Suppose now we want only one Wirth sequence of length n. (This is the form originally proposed by Wirth). We provide the procedure with a flag which is initially lowered, but

which is raised by the nth activation when it finds the first valid sequence. All other activations of the procedure terminate immediately if the flag is set.

```
procedure generate one Wirth sequence (n);
value n; integer n;
     begin
     integer array s[1:n];
     Boolean flag;

     procedure choose (k);
     value k; integer k;
          for s[k] := 1 step 1 until 3 while ¬flag do
               if still valid (k) then
                    begin
                    if k ≠ n then choose (k+1)
                    else
                        begin
                        flag := true;
                        comment sequence available
                        end
                    end of procedure "choose";

     Boolean procedure still valid (k);
     value k; integer k;
          comment as in Fig. 12;

     flag := false;
     choose (1)
     end of procedure "generate one Wirth sequence"
```

Fig. 13: A procedure for generating a single Wirth sequence

5. SOLVING THE WATERS-EDWARDS PROBLEM

We conclude with a "realistic example". Consider a program running in a computer with a single channel to which the n tape transports required by the program are attached. If we make the usual DP assumption that the processing time is negligible, then the time taken is that of tape reading and writing. Further, since the amount of information is fixed, the only variable is the stop-start time, which we therefore seek to minimise. This in turn means that we seek to minimise the number of blocks.

More formally given that:

R_i = no. of records in file i

C_i = no. of characters/record in file i

we wish to choose:

B_i = block size (i.e. records/block) for file i

to minimise:

$$\text{number of blocks} \quad \sum_{i=1}^{n} \frac{R_i}{B_i}$$

subject to:

$$\text{buffer space required} = \sum_{i=1}^{n} B_i C_i \leq s$$

There exists an analytical solution due to Waters [6] which unfortunately produces real values for B_k. Further the optimal integer values are in general neither the truncated or the rounded values of the real results and so the solution is found by a search of integer values around the analytic solution [3].

Suppose the range of search for B_k is between L_k and U_k. Then the naive solution of Fig. 14 is readily established.

```
procedure solve the Waters Edwards problem (n,R,C,s,U,L);
value n, s; integer n, s; integer array R,C,U,L;
    begin
    integer array B, opt B[1:n];
    integer min;

    procedure choose (k);
    value k; integer k;
        for B[k] := L[k] step 1 until U[k] do
            if k ≠ n then choose (k+1)
            else begin
                      n   R
                 if   Σ   --i < min then
                     i=1  B
                           i
                   begin
                        n
                   if   Σ   B *C  < s then update min
                       i=1   i  i
                   and opt B
                   end
                 end;
    initialise min and opt B;
    choose (1);
    comment optimum solution available
    end
```

Fig. 14: A naive solution (in crypto-Algol with some deliberate distortion) for solving the Waters-Edwards problem

A refined solution would be one in which the calculation of the number of blocks and the size of buffer required was not left until B_n was chosen, but was done on the run, each choice of B_k adding to the running totals. Furthermore at any level k:

(i) If the number of blocks exceeds the minimum so far found, the all selections stemming from the partial selection so far made can be ignored, since they can't possibly be optimal.

(ii) If the size of the buffer required exceeds the buffer space, then all selections stemming from the partial selection so far made can be ignored; as can all those stemming from larger values of B_k

since the buffer space increases with increasing B_k. Fig. 15 implements these refinements.

```
procedure solve the Waters Edwards problem (n,R,C,s,U,L);
value n, s; integer n, s; integer array R,C,U,L;
     begin
     integer array B, opt B [1:n];
     integer min;

     procedure choose (k, blocks, buffer);
     value k, blocks, buffer; integer k, blocks, buffer;
          begin
          Boolean over;
          over := false;
          for B[k] := L[k] step 1 until U[k] while ¬over do
               if blocks + R[k]÷B[k] < min then
                    begin
                    if buffer + B[k]*C[k] ≤ s then
                        begin
                        if k ≠ n then
                        choose (k+1, blocks + R[k]÷B[k],
                                     buffer + B[k]*C[k])
                        else update min and opt B
                        end
                    else over := true
                    end
          end;

     initialise min and opt B;
     choose (1, 0, 0);
     comment optimal solution available
     end
```

Fig. 15: A refined procedure for solving the Waters-Edwards problem

6. CONCLUSIONS

It is quite unrealistic to claim that this method is obviously superior to the techniques of back-tracking and non-deterministic programming. The most that can be said is that for some people this technique matches rather well with the way they conceive of a problem and its solution. [The author, for example, can usually get the procedures correct the first time, but has untold trouble with the back-tracking equivalents.] Perhaps the reader would care to try to solve a problem (such as the n-queens problem, finding a path through a maze, generating polyominoes and so on) and judge for himself. Alternatively he might like to write a back-tracking solution to the Waters-Edwards problem.

7. APPENDIX

Fig. 16 gives a back-tracking procedure for generating Wirth sequences. It has been produced in this form to illustrate the simulation of a nest of loops by two loops embedded in a third.

```
procedure generate Wirth sequences (n);
value n; integer n;
     begin
     integer array s[1:n];

     Boolean procedure still valid (k);
     value k; integer k;
          comment as in Fig. 12;

     Boolean more;
     k := 0;
     repeat
          while k ≠ n ∧ still valid (k) do
               begin
               k := k+1; s[k] := 1
               end;
          if still valid (k) then comment sequence available;
          while s[k] = 3 ∧ k ≠ 1 do k := k-1;
          more := s[k] ≠ 3
          if more then s[k] := s[k]+1
     until ¬ more
     end
```

Fig. 16: A back-tracking procedure for generating Wirth sequences

The procedure has been drafted as "simply" as possible (as seen by the author) to concentrate on the essential structure. A simple analysis of the problem shows that both inner loops can be obeyed at most twice, and this fact might be capitalised on. Further, the usual programming devices could simplify the loop termination and the procedure, still valid, is invoked twice when once would be sufficient. [For an alternative development taking account of this see Wirth's book.]

Fig. 17 gives a non-deterministic procedure. Since non-deterministic programming was developed in a statement-by-statement language and the only preprocessor known to the author [2] was based on Fortran, the solution here is meant to give the flavour only.

```
procedure generate Wirth sequences (n);
value n; integer n;
     begin
     integer array s[1:n];

     Boolean procedure still valid (n);
     value n; integer n;
          comment As in Fig. 12;

     integer k;
     for k := 1 step 1 until n do
          begin
          s[k] := choice (1, 2, 3);
          if ¬still valid then fail
          end;
     success
     end
```

Fig. 17: A non-deterministic procedure for generating Wirth sequences

8. REFERENCES

[1] R.A. Brooker and J.S. Rohl, Experience with a First-year Computer Science Course, Computer Bulletin, Vol. 10, No. 3, 1966.

[2] J. Cohen and E. Carton, Non-deterministic Fortran, Computer Journal, Vol. 17, No. 1, 1974.

[3] B.J. Edwards, A paper to be published in the Computer Journal.

[4] R.W. Floyd, Nondeterministic Algorithms, Journal of A.C.M., Vol. 14, No. 4, 1967.

[5] S.W. Golomb and L.D. Baumert, Backtrack Programming, Journal of A.C.M., Vol. 12, No. 4, 1965.

[6] S.J. Waters, Blocking Sequentially Processed Magnetic Files, Computer Journal, Vol. 14, No. 2, 1972.

[7] N. Wirth, Systematic Programming: An Introduction, Prentice-Hall, 1973.

15
Some New Techniques for Verifying Programs
Rodney W. Topor

1. INTRODUCTION

The most commonly used method for verifying imperative programs is that of inductive assertions as described by Floyd [5] and Hoare [6]. Other methods used for recursive programs include computation induction [7] and structural induction [1,7]. In recent years, however, several new approaches to program verification have been developed, and it is the aim of this paper to give brief descriptions of these new methods, indicating the relationships between them, and the circumstances under which each method is most suitable. The presentation will be informal and will rely on examples, so non-specialists in the field need not be afraid to read on. The methods which we shall consider are subgoal induction [9], continuation induction [11,12] predicate transformers [4], and intermittent assertions [2,8].

2. SUBGOAL INDUCTION

The essence of this approach can be seen by considering the flowchart for a simple iterative program, shown in Fig. 1. We wish to show that a relation $R(x_o\ ;\ x_f)$ holds for any path from A to C (x_o, x_f refer to the initial and final values of x respectively).

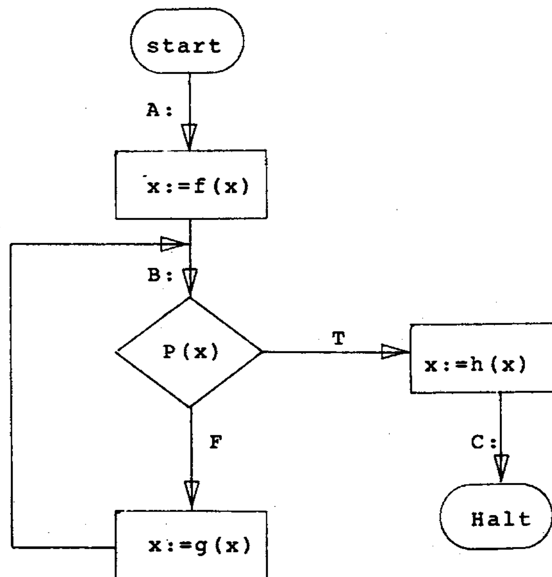


Fig. 1

In a proof by inductive assertions one invents the subproblem of showing that a relation $I(x_o; x)$, called a loop invariant, holds for all paths from A to B. One then uses I to prove that R holds between A and C. Thus one must prove:

(F1) $I(x_o; f(x_o))$

(the invariant holds initially)

(F2) $I(x_o; x) \wedge \sim P(x) \Rightarrow I(x_o; g(x))$

(the invariant still holds after passing once around the loop).

(F3) $I(x_o; x) \wedge P(x) \Rightarrow R(x_o; h(x))$

(the invariant implies the output specifications after exit from the loop)

The subgoal induction method decomposes the problem in the reverse way. It required one to invent a relation $Q(x; x_f)$, called a subgoal, which describes the effect of any computation from B to C, and then uses Q to prove that R holds between A and C. This results in the following verification conditions to be proved:

(M1) $P(x) \Rightarrow Q(x; h(x))$

(The subgoal holds when we are about to exit from the loop).

(M2) $Q(g(x); x_f) \wedge \sim P(x) \Rightarrow Q(x; x_f)$

(if the subgoal holds after passing once more around the loop, then it holds for the current state as well).

(M3) $Q(f(x_o); x_f) \Rightarrow R(x_o; x_f)$

(if the subgoal holds the first time we reach the loop, then the output specification is satisfied).

As an example, consider the simple program shown in Fig. 2 to compute the greatest common divisor of two integers (expressed using Dijkstra's non-deterministic guarded command constructs [4]). The output assertion is

$$R(x_o, y_o; x_f) \equiv x_f = \gcd(x_o, y_o)$$

where gcd(x,y) represents the greatest common divisor of x and y.

```
do  x > y → x := x - y
 ▯  x < y → y := y - x
od
```

Fig. 2

In this case we can simply choose $Q \equiv R$ as the subgoal. Two verification conditions then need to be established to prove Q by subgoal induction. The first, from M1, is

$$x = y \Rightarrow x = gcd(x, y) .$$

The second, from M2, splits into two cases which are

$$x > y \wedge x_f = gcd(x-y,y) \Rightarrow x_f = gcd(x,y), \text{ and}$$

$$x < y \wedge x_f = gcd(x,y-x) \Rightarrow x_f = gcd(x,y) .$$

Each of these formulae is easily proved using properties of the gcd function. (The verification condition arising from M3 follows trivially).

Notice that it was not necessary to invent an invariant assertion to prove R for this program; R itself was a sufficient inductive hypothesis. To verify the program by inductive assertions, however, it is necessary to think of the invariant $gcd(x_o,y_o) = gcd(x,y)$.

```
j  :=  1;
do A[j] ≠ 0 → j := j+1 od;
A[j]  := 2
```

Fig. 3

Another example where subgoal induction is more convenient than inductive assertions is shown in Fig. 3. Here the relation to be proved is

$$R \equiv (\forall i)\ [A_f[i] = \underline{\text{if}}\ i{=}m\ \underline{\text{then}}\ 2\ \underline{\text{else}}\ A_o[i]\ \underline{\text{fi}}]\ ,$$

where $m = \min\ \{i | i \geq 1 \wedge A_o[i] = 0\}$. To prove this by inductive assertions one would use the invariant

$$I(A_o;A,\ j) \equiv A_o = A \wedge (\forall i | 1 \leq i < j)\ A[i] \neq 0.$$

Having established it, one must then show that the final step achieves R. The induction hypothesis needed for a proof by subgoal induction is the minor generalization one gets by replacing 1 in R by j, i.e.

$$Q(A,\ j;\ A_f) \equiv (\forall i)[A_f[i] = \underline{\text{if}}\ i{=}m\ \underline{\text{then}}\ 2\ \underline{\text{else}}\ A[i]\ \underline{\text{fi}}]\ .$$

where $m = \min\{i | i \geq j \wedge A[i] = 0\}$. Once this is proved, it is trivial to show that R itself holds. In this case the subgoal induction approach has the two advantages of reducing the effort in devising an induction hypothesis and

simplifying the final step of the proof.

It is easy to show that proofs by either one of these two methods may be used to produce a proof by the other. Suppose one has a proof by inductive assertions, i.e. proofs of F1, F2, and F3. To prove R by subgoal induction define

$$Q(x\ ;\ x_f) \equiv (\forall x_o)\ [I(x_o\ ;\ x) \Rightarrow R(x_o\ ;\ x_f)]\ .$$

Then M1 is equivalent to

$$P(x) \wedge I(x_o\ ;\ x) \Rightarrow R(x_o\ ;\ h(x))$$

which is just F3. M2 becomes

$$\sim P(x) \wedge (\forall x_o)[I(x_o\ ;\ g(x)) \Rightarrow R(x_o\ ;\ x_f)]$$

$$\Rightarrow (\forall x_o)[I(x_o\ ;\ x) \Rightarrow R(x_o\ ;\ x_f)]\ .$$

To prove this, it is sufficient to show

$$\sim P(x) \wedge I(x_o\ ;\ x) \Rightarrow I(x_o\ ;\ g(x))$$

which is just F2. M3 becomes

$$(\forall x_o)[I(x_o\ ;\ f(x_o)) \Rightarrow R(x_o\ ;\ x_f)] \Rightarrow R(x_o\ ;\ x_f)$$

which follows from F1, completing the argument. To rephrase a proof by subgoal induction as a proof by inductive assertions, define

$$I(x_o\ ;\ x) \equiv (\forall x_f)[Q(x\ ;\ x_f) \Rightarrow R(x_o\ ;\ x_f)]\ ,$$

and the details are similar to the above.

Despite the formal equivalence between the methods, we have seen cases where subgoal induction makes both the discovery of the induction hypothesis and the subsequent proof simpler. In other cases, however, proofs using inductive assertions are both simpler and more natural.

In addition to the simple programs we have considered, subgoal induction can also be applied to more general program structures, including multiple-exit loops, mutually recursive functions and nested recursive calls. In such cases it has further advantages which we do not consider here. A complete description of the method is given in [9].

3. CONTINUATION INDUCTION

This method, developed by R.M. Burstall and the author, treats program verification from an unusual standpoint. It arose from a desire to use symbolic execution effectively

and to construct proofs which would appeal to "programmers" rather than "mathematicians". Although developed quite independently of subgoal induction, the two methods turn out to be very similar.

In continuation induction, the specification R is replaced by an alternative (or virtual) program which is then shown to be equivalent to the original (or actual) program. The virtual program may use arbitrary functions or predicates defined by axioms in addition to the standard primitives of the programming language, and hence may be made sufficiently clear and simple that proving the equivalence of the two programs demonstrates that the original program performs as intended. In general it is also necessary to provide a virtual program describing the intended effect of each loop in the program (for the same reason that Q or I are needed - to make the induction hypothesis strong enough).

The equivalence of the two programs is shown by symbolically executing each of them in turn, and checking that they return the same results. Symbolically executing the virtual program is straightforward as it is always defined such that it has a finite execution tree. When symbolically executing the actual program, however, we use the following induction principle:

Whenever we return to the beginning of the program (or loop), instead of continuing to execute the actual program, we use the induction hypothesis that the two programs are equivalent and continue from the current state by executing the corresponding virtual program.

This principle is just induction on the number of steps remaining in the computation, as is the principle used in subgoal induction.

As an example, in the greatest common divisor program of Fig. 2, the virtual program specification is simply the assignment "x := gcd(x,y)". Symbolically executing the actual program involves just three paths: $x=y$, $x > y$, and $x < y$. Consider the second of these: after passing once through the loop the (symbolic) value of the indentifier x (which was "x") is now "x - y", after executing the virtual program from that state the final value of x is "gcd(x-y,y)". We now equate the final values of x obtained by executing the two programs, taking the corresponding path conditions into account, and obtain the following verification conditions to be proved:

1) $x = y \Rightarrow x = \gcd(x,y)$

2) $x > y \Rightarrow \gcd(x-y,y) = \gcd(x,y)$

3) $x < y \Rightarrow \gcd(x,y-x) = \gcd(x,y)$.

These formulae are slightly easier to prove from our knowledge of greatest common divisors than those arising from subgoal induction or inductive assertions.

As a second example, consider the following recursive function definition:

$$f(x) \Leftarrow \underline{if}\ x > 100\ \underline{then}\ x - 10\ \underline{else}\ f(f(x+11))\ \underline{fi}$$

We wish to show that the (actual) program, "f(x)", is equivalent to the virtual program.

$$\text{"}\underline{if}\ x > 100\ \underline{then}\ x-10\ \underline{else}\ 91\ \underline{fi}\text{"}$$

Fig. 4 shows the tree of computation paths generated by symbolically executing the actual and virtual programs. Except for the top-level call in the actual program, whenever f is called the corresponding virtual program is executed instead (such virtual calls are shown with dotted lines). Note that one impossible branch, 91 > 100, has been cut off during symbolic execution. Then by considering all pairs in the cartesian product of the sets of states at the tips of the trees, and equating the final values obtained, the verification conditions shown in Fig. 5 are obtained. The problem has been automatically broken down into simple cases during symbolic execution, and the resulting formulae are all trivial to prove.

Let us now consider the relationship between subgoal induction and continuation induction in the special case that $Q(x\ ;\ x_f)$ represents a function, $x_f = k(x)$, as in the above gcd example. Suppose we have proved Q holds for the program in Fig. 1 by subgoal induction. The corresponding specification for a proof by continuation induction is the virtual program 'x:=k(x)'. This specification yields the verification conditions

1) $P(x) \Rightarrow h(x) = k(x)$, and

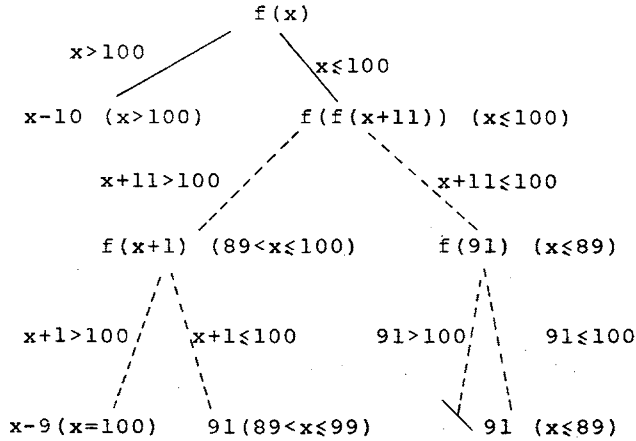
2) $\sim P(x) \Rightarrow k(g(x)) = k(x)$,

which follows easily from M1 and M2. Thus, in this case, proofs using subgoal induction can be translated into proofs using continuation induction; the translation in the other direction is equally simple.

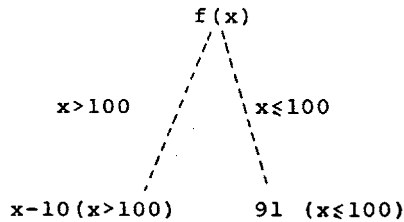
Continuation induction can also be applied when the corresponding relation Q does not represent a function. It does this by introducing Hilbert's epsilon operator, a non-deterministic construct written $\varepsilon z.\ Q(x\ ;\ z)$ which denotes any z such that $Q(x\ ;\ z)$ holds, and using statements such as "$x := \varepsilon z.\ \ Q(x\ ;\ z)$" as virtual programs. For example, the outer loop of a sorting program might be equivalent to

$$\text{"}A := \varepsilon A'\ .\ \text{perm}\ (A[1:i],\ A'[1:i]):\ \wedge\ \text{ordered}\ (A'[1:i])$$
$$\wedge\ A[i+1:n] = A'[i+1:n]\text{"}\ .$$

It is worth noting at this point that both subgoal induction and continuation induction only prove partial correctness: they do not guarantee termination. With continuation induction we are really proving that the actual program is



4a: Actual Program



4b: Virtual Program

Fig. 4

$x>100 \wedge x>100$	$\Rightarrow$	$x - 10 = x - 10$
$x>100 \wedge x\leq100$	$\Rightarrow$	$x - 10 = 91$
$x=100 \wedge x>100$	$\Rightarrow$	$x - 9 = 91$
$x=100 \wedge x\leq100$	$\Rightarrow$	$x - 9 = 91$
$89<x\leq99 \wedge x>100$	$\Rightarrow$	$91 = x - 10$
$89<x\leq99 \wedge x\leq100$	$\Rightarrow$	$91 = 91$
$x\leq89 \wedge x>100$	$\Rightarrow$	$91 = x - 10$
$x\leq89 \wedge x\leq100$	$\Rightarrow$	$91 = 91$

Fig. 5

"included in" the virtual program, i.e. whenever the actual program terminates it returns the same results as the virtual program.

To cope with these two properties, a set of inference rules is used which allows the mechanical simplification of verification conditions until they are formulae in first-order logic. Lack of space precludes a description of these rules, but in general one can translate a proof using subgoal induction to a proof using continuation induction by replacing the subgoal $Q(x ; x_f)$ by the virtual program "$x := \varepsilon z. Q(x ; z)$"

Although proofs by continuation induction can be complex, the method shares many of the advantages of subgoal induction. Moreover, it allows recursive, iterative and non-deterministic programs to be treated uniformly; it handles escapes and procedures with side-effects; and it is specially convenient for proving properties of certain recursive programs. Complete descriptions of the method are given in [11,12]; [11] also contains a description of an interactive program verifier based on continuation induction. A proof of the validity of the method can be found in [3].

4. PREDICATE TRANSFORMERS

The above methods only prove partial correctness of programs, as does the method of inductive assertions, though separate proofs of termination can easily be given. In an attempt to include termination proofs within Hoare's axiomatic approach, Dijkstra [4] has introduced the concept of the "weakest precondition": if S is some command and R is a condition on the state of the system, then wp(S,R) denotes the weakest precondition on the initial state of the system such that the activation of S will properly terminate in a state satisfying the postcondition R. Such a wp is called a "predicate transformer" because it associates a precondition with a postcondition R. The semantics of a simple programming language are defined by the following axioms:

D1) wp ("skip", R) = R

D2) wp ("x:=E", R) = $R_{E \to x}$

where $R_{E \to x}$ denotes a copy of R with each occurrence of x replaced by (E).

D3) wp ("$S_1;S_2$", R) = wp(S_1, wp(S_2, R))

D4) wp (IF,R) = BB $(\forall i | 1 \leq i \leq n)(B_i \Rightarrow \text{wp}(SL_i, R))$

where IF ≡ <u>if</u> $B_1 \to SL_1$ [] ... [] $B_n \to SL_n$ <u>fi</u>

and BB = $(\exists i | 1 \leq i \leq n)\ B_i$

D5) wp (DO,R) = $(\exists k | k \geq 0)\ H_k(R)$

where DO ≡ <u>do</u> $B_1 \to SL_1$ [] ... [] $B_n \to SL_n$ <u>od</u>,

and $H_o(R) = R \wedge \sim BB$,

$$H_k(R) = wp\ (IF,\ H_{k-1}(R)) \wedge H_o(R),\ k > 0\ ,$$

where IF is the alternative statement obtained by replacing do od by if fi in DO.

The two fundamental theorems for alternative and repetitive constructs respectively are:

T1) If $(\forall i | 1 \leq i \leq n)(Q \wedge B_i \Rightarrow wp(S_i, R))$ for all states,

then $Q \wedge BB \Rightarrow wp(IF,R)$ holds for all states

T2) If $(P \wedge BB) \Rightarrow (wp(IF,P) \wedge wdec(IF,t) \wedge t \geq 0)$

holds for all states, then $P \Rightarrow wp(DO, P \wedge \sim BB)$ holds for all states. (The expressions involving t guarantee termination, see [4]).

Although Dijkstra's main concern is to provide tools to aid in the systematic development of programs, when used to verify existing programs his method turns out to be equivalent to the inductive assertion method. We indicate this by considering again the program in Fig. 2. The postcondition R is $x = gcd(x_o, y_o)$ and the invariant P is

$$P \equiv gcd(x,y) = gcd(x_o, y_o) \wedge x>0 \wedge y>0.$$

Verifying this program, denoted DO, requires us to show that $P \Rightarrow wp(DO,R)$. (P is assumed true initially). Since

$$P \wedge x=y \Rightarrow R \qquad (V1)$$

the problem is reduced to showing $P \Rightarrow wp(DO, P \wedge x=y)$. Ignoring termination conditions (with apologies), this follows from T2 provided

$$P \wedge (x>y \wedge x<y) \Rightarrow wp(IF,P)$$

where $IF \equiv$ <u>if</u> $x>y \rightarrow x:=x-y\ []\ x<y \rightarrow y:=y-x$ <u>fi</u>

But, in turn, this last statement follows from T1 provided

$$P \wedge x>y \Rightarrow wp("x:=x-y",\ P) \qquad (V2)$$

and

$$P \wedge x<y \Rightarrow wp("y:=y-x",\ P) \qquad (V3)$$

But after applying the definition of the assignment statement (D2), formulae V1, V2 and V3 simply become the verification conditions obtained in a proof by inductive assertions (working forwards).

As a tool for developing programs, predicate transformers are very useful. For verifying existing programs, however, the method is more restricted than inductive assertions in allowing only concatenation, alteration and repetition as control structures.

5. INTERMITTENT ASSERTIONS

This method also allows the termination and correctness of a program to be shown in a single proof. It involves attaching comments to points in the program such that control will sometimes pass through the point and satisfy the attached assertion. At other times control may pass through the point without satisfying the assertion; therefore the comments are called intermittent assertions. If the output specification is an intermittent assertion at the program's exit and can be proved (to sometimes hold), then the program's total correctness has been established.

```
Start:  stack := empty; count := 0;

Loop:   if tree is a tip

           then count := count + 1;

           Test: if stack = empty then go to Finish fi;

                 Pop tree from stack;

                 tree := right(tree); go to Loop

           else  Push tree onto stack;

                 tree := left(tree); go to Loop

        fi;

Finish:
```

Fig. 6

Consider the program shown in Fig. 6 which counts the number of tips in a binary tree. The program is a translation of the obvious recursive function.

```
tips(tree) <= if tree is a tip then 1

                 else tips (left(tree)) + tips(right
                                                (tree))

              fi
```

Using intermittent assertions, we can express the total correctness of this program in the following theorem:

Theorem: if sometimes tree = t at Start

then sometimes count = tips(t) at Finish.

(Here, "sometimes count = tips(t) at Finish" is the inter-

mittent assertion). The proof of the theorem follows easily from the following lemma, which describes the behaviour of the program in more detail.

Lemma: if sometimes tree = t, count = c and stack = s at Loop then sometimes count = c + tips(t) and stack=s at Test.

The proof of this lemma is by complete induction on the structure of tree. We suppose control is at loop with tree = t, count = c and stack = s, and symbolically execute the program trying to achieve a state in which the lemma's conclusion holds. Our inductive assumption is that the lemma holds whenever tree = t', where t' is any sub-tree of t. Details of the proof are given in [2] and [8]. It is worth noting that the lemma is applied inductively twice: once for tree = left(t) and once for tree = right(t), reflecting the recursive origin of the program. (The invariant at Loop used to verify this program by inductive assertions is

$$\text{tips}(\text{tree}_o) = \text{count} + \text{tips}(\text{tree}) + \sum_{s \in \text{stack}} \text{tips}(s) \text{ .)}$$

The advantage of using intermittent assertions are that total correctness can be shown in just one proof; that the resulting proofs often reflect our intuitive understanding of why programs work, and that they can be used to prove validity of program transformations and the correctness of continuously operating programs. The method was first explicitly described by Burstall in [2], and further examples, applications and its relationship to other methods are described in [8].

6. CONCLUSIONS

Each of the methods discussed has its own characteristics. The inductive assertion method is the best known, but in its basic form is only applicable to flowchart programs; it can be used, however, for most programs. Subgoal induction is effectively a dual to inductive assertions and can be used with both functional and imperative programs. It allows easier generalization of output specifications to subgoals (invariants) in many cases. Continuation induction also has this advantage and is specially convenient when the input-output relation of a program can be expressed as a function. Predicate transformers are only defined for a simple imperative language but do lead to termination proofs; this appears to be the most suitable technique to aid program development. The intermittent assertion method is generally applicable, also proves total correctness, and best reflects our intuitive understanding of programs. A more formal comparison of some of these methods is given in [10].

The problem of verifying programs, however, remains a difficult one, and none of the above methods is a panacea. Inductive proofs are always necessary, and finding appropriate inductive statements (be they assertions, programs or lemmas) for existing programs can always present difficulties. As has been said many times before, the simultaneous construction of programs and their correctness proofs, followed by the

application of correctness-preserving transformations, is the only practical way to construct large programs which are likely to be correct.

REFERENCES

[1] R.M. Burstall, Proving properties of programs by structural induction, Computer J. 12, 1, February 1969, 41-48.

[2] R.M. Burstall, Program proving as hand simulation with a little induction, Proc. IFIP Cong. 1974, North-Holland, 308-312.

[3] R.M. Burstall, A note on program proof by a continuation method, DAI Working paper 7, University of Edinburgh, February 1975.

[4] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM 18, 8, August 1975, 453-457.

[5] R.W. Floyd, Assigning meanings to programs, Proc. Symp. Appl. Math., Vol. 19, (1967), AMS, 19-32.

[6] C.A.R. Hoare, An Axiomatic basis for computer programming, Comm. ACM 12, 10, October 1969, 576-580, 583.

[7] Z. Manna, S. Ness and J. Vuillemin, Inductive methods for proving properties of programs, Comm. ACM 16, 8, August 1973, 491-502.

[8] Z. Manna and R. Waldinger, Is "sometime" sometime better than "always"? Intermittent assertions in proving program correctness, Report No. STAN-CS-76-558, June 1976, Stanford University.

[9] J.H. Morris, Jr. and B. Wegbreit, Program Verification by subgoal induction, Memo September 1975, Xerox Research Center, Palo Alto, Ca. Also Comm. ACM (to appear).

[10] C. Reynolds and R.T. Yeh, Induction as the basis for program verification, IEEE Trans. on Software Engineering 2, 4, December 1976, 244-252.

[11] R.W. Topor, Interactive program verification using virtual programs, Ph.D. thesis, July 1975, University of Edinburgh.

[12] R.W. Topor, Continuation induction: program verification using virtual programs, Memo in preparation.

16

Computer Language Systems and Cognitive Processes

Garth L. Wolfendale

ABSTRACT

This paper explores the literature on the semantics of natural languages from a linguistic and psycho-linguistic point of view. The evidence points strongly towards a system that processes linguistic information without recourse to the complications of lexical searches, dictionary look-ups, deep structure and transformations - the apparatus of generative semantics. The evidence is for a multistage process which eventually generates statements and implications (truth sets) in some internal 'cognitive' language leading to an 'understanding' that is partial, being completed later, "off-line" or as a background process. This model is considered for its implications for computer language systems design, as an example of a system based on those principles as described.

1. COGNITIVE PROCESS RESEARCH RELEVANT TO COMPUTER LANGUAGE SYSTEM THEORY AND DESIGN

First of all I wish to briefly discuss the evidence and arguments for the hypothesis that; for human beings:-

> *"There has to be a private, internal cognitive language. In fact in the history of philosophical and psychological investigation into cognitive processes .... there is only one kind of theory that has ever been proposed for concept learning - indeed, there would seem to be only one kind of theory that is conceivable - and this theory is incoherent unless there is a language of thought"*
>
> (Fodor [7], p.36)

A simple argument for this at a philosophical level comes from Wittgenstein [19]. Consider the statement that *"I have a mild tickle"*.

(a) There is no public evidence for this that can lead to verification by others.

(b) The 'private' evidence could be essentially random in that there is nothing to show that *"mild tickle"* is properly applied to sensations like those I am having. If there is no such evidence then there is no difference between getting the use of the term correct or getting it wrong: no difference between obeying the conventions for the use of the term or failing to obey them if there are any. But a convention for which adherance or not is irrelevant, is no convention at all. A term that can be used at random is no term at all, and a language without terms is no language at all - and could not be a private language. However a system that responded randomly to its input could not show such

properties as learning (which is essentially anti-entropy) and therefore we have a contradiction. If there is no private language there is no order. Since we at least have order (of a kind) then there must be a private language of a kind.

So-called *'natural'* language is the means by which constructs in the private system are made public and vice versa.

2. WHAT MUST THE PRIVATE LANGUAGE BE LIKE?

I would like to consider a powerful statement made by Fodor [7]:- Consider a language L that some human (or system) is learning. Then we can make the following disjunctions:-

(a) It is false that learning L is learning its truth definition (the conditions under which its predictates are true or false, i.e. truth conditions), or

(b) It is false that learning a truth definition for L involves projecting and confirming hypotheses about the truth conditions upon the predicates of L, or

(c) No one learns L unless he already knows some language different from L but rich enough to express the extensions of the predicates of L.

Given the current state of theorizing about language and learning - only the third approach seems possible. It follows immediately:-

> *"that not all the languages which one knows are languages one has learned, and that at least one of the languages which one knows without learning is as powerful as any language that one can ever learn."*

A consequence of this is that so-called *"natural language"* is not the language of thought. It has a role in the process we normally call thinking but it is not the foundational cognitive language. The latter is more powerful than natural language, enabling intelligent behaviour to proceed in terms not expressible in natural language, and thus not communicable by natural language.

Thus the drive by some workers to make computer languages closer and closer to natural language is ill founded, since so much of cognitive activity is carried out in other languages (e.g. consider how much information is processed, understood, extrapolated over such media as animation and music in which natural language plays a very minor role).

As Fodor points out:-

> *"Nothing can be expressed in a natural language that can't be expressed in the language of thought. For if something could, we couldn't learn the natural language formula that expresses it."*

Bryant [3] also makes this point in his discussion of language learning in children.

Assuming that we have established the points just made, what can be said about this language and the system which embodies it at a functional and structural level.

I am going to go into some detail on linguistic and psychological evidence, since the system that emerges will be used as a model for computer systems, providing powerful programming language subsystems. Also the linguistic arguments are relevant for all language systems and pertinent for work on language understanding and representation systems (e.g. [1]).

3. LANGUAGE AND COMMUNICATION

Languages as communication devices are the means by which systems are extended beyond their private domains and impacted by other systems. Considering that physical media and different levels of language are involved, in linguistic terms we have, traditionally, phonetic, morphophonological, surface syntactic, deep syntactic and so on (according to Chomskyan models of language [5]), and in computer system terms we have physical, logical, resource-level protocols with the message as the basic unit. These levels are stripped off and processed progressively in the system, each giving its different type of information and errors, and incidently cutting down possibilities, leading to a message that has been processed ready for some further interpretation in terms of the internal private language.

For the purposes of a seminar on programming language systems, further discussion of the processing of low levels is not relevant - in fact syntax analysis, Chomskyan models of languages, parsers and so on have been discussed ad nauseum. However, in work on the design and development of computer network systems, the link between protocols and linguistic structure have proven personally insightful.

In summary, again from Fodor [7] - at the level we are discussing here:-

(a) Nothing can be an adequate representation of a message unless it can serve as input to a device capable of computing the structural description of those sentences which express that message; "structural description" is here taken in its technical linguistic sense.

(b) Nothing can be an adequate representation of a message unless it can be produced as output by a device requiring as input a structural description of a sentence which expresses the message.

To generalise somewhat, the structures that we identify with messages will have to provide appropriate domains for *whatever* cognitive operations apply to the information in the message at the level indicated by the structures (protocols).

For example, the things that we read and hear are often confirmed or otherwise by the things that we taste, touch,

smell, remember and so on. There must be computational procedures which can relate the information in the remark that *"it is sunny"* to the information gained by looking out of the window. An obvious way to achieve this is to translate the visual information and verbal information into a common language, and apply verification or other procedures at that level. There are many ways of achieving this, and some psychological evidence that there are marked individual differences in this process. For example, Clark and Chase [6], found evidence that some people seem to use a *"natural language"* translation from visual input, thus guaranteeing a common base between visual and linguistic information. Can you imagine a computer language system providing:-

If picture (x) *means-same-as* statement (y) *then* ....

Split-brain patients [17] showed difficulties in carrying out such comparisons.

So we can extend our structural analysis into the deep or semantic domain - where we hit a language which is now common to all higher-level channels, which I will call information channels - and, if you recall, is more powerful than any language that we can learn (or compile in a programming language system). This internal system can be thought of, to complicate the picture somewhat, as being made up of various types and levels of representation, and which one becomes assigned in the course of given operation is determined by a variety of variables including such factors as motivation, attention and the demand characteristics of a task.

In programming language system terms, this extends the picture from one of a single target language system to multiple target systems, managed and coordinated by the equivalent of a virtual machine monitor. Wolfendale [21] describes such an approach for data base management systems, where, as is described in the next section, the problems are identical to those in programming language systems.

The question now is what is the nature of the semantic structure and how does it impact the system that must process information at this level. There is one school of thought which treats semantic structure as a dictionary. This view, exemplified by Katz and Fodor [12], treats the processing of information as tracing of a dictionary like *'semantic net'* progressively replacing *"higher-order"* terms by *"lower order"* terms and connectives. For example *"he is a bachelor"* : becomes *"he is not a married man"* becomes *"this man is a man without a wife"* : becomes *"This man is a man without a married woman"* : becomes *etc. etc.* Progress in this search is determined by whether the expansion or conversion increases the semantic constituents of the information. That is *'bachelor* is replaced by *'not married'* and *'man'* because now we know two things:- we are talking about a man and he is not-married.

There has been a great deal of work in terms of this *'semantic tree'* model in linguistics, psychology, artificial intelligence and computer science (e.g. information and data base systems), even though the theory is ill-formed (what are the stopping rules for example?) and provides no insight into the relationships

between tokens in the dictionary and what is allowable in *"the real world"*. The theory is really vacuous unless something can be said about what may appear in the dictionary, i.e. something that specifies the form and content of possible definitions. Katz [11] attempted to formulate such constraints within the assumptions of *"interpretive"* semantics. Janet Fodor has produced an extensive review of this work and the attempts to constrain the theory to be a valid scientific hypothesis explaining some aspects of semantics.

The question now arises: is the dictionary approach necessary or desirable, or is there something better.

First of all, there is no a priori reason why conditions upon well-formedness in the internal (or object) language should mirror conditions upon well-formedness in the surface language L. In particular there is no a priori reason why the definitions of terms in L should be expressible in L. In fact this is mistaken since although such a mapping looks alright (e.g. bachelor and unmarried man), when we consider other than nouns, (e.g. *'kill'* and *'cause to die'*) we get into trouble:- *"A killed B'* is well formed but *"A caused to die B'* is not.

Thus the approach outlined so far, in order to cope with such problems, has resorted to transformations, applying structural information to semantic mappings in order to maintain well formedness. Here, *'A caused to die B'* would then be transformed to *'A caused B to die'*. Now there are as many transformations as there are problematical dictionary mappings. This approach will be referred to as generative semantics.

Fodor [7], chapter 3, gives detailed examples of where the transformational approach breaks down. In fact Fodor concludes that for the data under consideration there is no syntactic or transformational process of definition, and *that there is no process for definition at all;* i.e. both the defined expression and its definition appear as items in the primitive vocabulary of the representational system.

If entailments that derive from terms in the vocabulary of L do not depend on the process of definition, how are they determined?

A standard proposal since Carnap [4] is that, if we want F in L to entail G then the statement *'F entails G'* is added to the inference rules. These have become referred to as *'meaning postulates'*, and these are proposed as doing the work that definitions are supposed to do. Fodor et al.[8] provide extensive coverage of the role of meaning postulates in the semantic analysis of natural languages.

In summary, the alternative approach to the generative semantic model proposes that there is a target language L*, which is extensible by addition of meaning postulates. When a message in L is processed it generates terms in L* plus related meaning postulates which provide "entailments". There is no progressive transformation and dictionary look-up during the processing of L messages, they are translated into L*

terms pretty directly, plus entailments, to generate a map of the message plus a truth set of statements in L*. Let us call this T*. When T* is generated the L message is treated as accepted and understood in an on-line sense. After this, other L messages can be processed, and the T* sets can be further processed, *"off line"* as background tasks.

Thus there are two levels of understanding - an immediate level which accepts a message as meaningful and a longer term level that processes the processed message and its truth or implication set. This certainly corresponds with experience and there is significant psychological evidence for it (e.g. [13]), showing that there is some information about the content of linguistic material available within 250 msec of its reception in human beings, and that after a longer period the information increases significantly. This is counter evidence to the generative semantic approach.

From a programming language system point of view (information and database management systems also), the latter approach places the computational load where it is most easily accommodated: on off-line or background processes. This is also where most searching or large quantities of interrelated data (driven by T*) needs to be done, *not* in the foreground, as is predicted by the generative semantic model.

A consequence of this approach, also, is that the closer L and L* are in structure, the simpler their mapping and the faster understanding and T* generation can proceed. Over the millions of years of evolution, one might expect this, so that humans learn natural languages extremely easily since the languages map simply onto their internal languages, L*.

I will not go into the psychological work involved in discovering the semantic processes of man since it is extensive. The interested reader should read [8,10,14,15,16, 18].

Since the human being is the most sophisticated language processing system we know, then perhaps, given our current state of knowledge we could benefit by embodying some of the cleaner principles that have evolved from cognitive studies in programming language systems, or computer language systems in general. The next section explores such application.

4. IMPLICATIONS FOR COMPUTER LANGUAGE SYSTEMS

First of all, I wish to argue that we should extend the field of discourse from just programming language systems to computer language systems covering:-

- Programming language systems as we know them
- Information and data base management systems
- Communication language systems
- Modelling language systems
- Control language systems

to name but a few.

I believe that the problems of structure, efficiency, semantics and power are common to all of them, the only differences being the *stress* placed on particular aspects.

For example - programming systems stress the representation and support of classes of algorithms - however COBOL algorithms are worlds apart from PASCAL algorithms in terms of the natural expressiveness of each. Information systems stress the semantic processing and retrieval of human-consumable data, database management systems stress the management of logically related and possibly complex data in general, communication systems stress the reliable transmission and translation of information, modelling systems stress control and data flow aspects with events being the basic units, and control language systems stress the interrelationships between the user and the system which he wishes to control (cause to do some action). They all can coexist on the same computer system, representing a varied array of activities, types of processing, system demands, data-requirements and so on.

I argue that it is impossible at the present time to develop one target language system that is optimum for all classes of computer language system, but we can use the virtual machine concepts at our disposal to support multiple language systems on the same host, each being optimised for the type of language in hand. If L demands efficient stack and vector processing we set up a virtual machine optimised for L.

5. AN IMPLEMENTATION OF A DATA LANGUAGE SYSTEM

Five years ago I was faced with the problem of producing a data language and manipulation system that was to service data for - serial files, complex lattice and index files, graphical displays and input, CODASYL data base structures and manipulations and a host of other data structures. The question was - how was the target language system to be organized?

The literature in the field provided no information of any relevance, so I resolved to a study of human information processing, in particular, the work on languages and semantics, which I have described (and updated) in the previous section.

The approach that was recommended from looking at the computer and artificial intelligence literature was the dictionary look-up and transformational approach (generative semantics). However, considering how long the system would have spent dictionary searching during operations, in order to support the data languages required for the different data applications, another approach was sought. Instead of compiling each data-dialect independently into run-time data structures and procedures, an intermediate data language was developed which, it was hoped, was more powerful than any of the individual specific languages. This internal language,

DDL*, was incomplete in that its "terminal" strings were not terminal, but variables. When a data language was defined, it was linked to the internal language by means of these terminal variables. The necessary rules defining a language, DDL (i), say, were then added to the original rules of the internal language, thus extending it. This approach is not exactly new, since it underlies the compiler-compiler of Brooker et al [2] and similar work, but it had not been generalised to other than programming languages. Wolfendale [20] describes DDL* in detail, and only a brief sketch will be given here.

6. DESCRIPTION OF DDL*

The main construct in DDL* is the schema, which is broken down into four divisions .. data structure, data-control, data-selection and data- organisation. DDL* is a language onto which data-description languages can be mapped so that those languages need not be sectioned into divisions. However, the use of dynamic terminal variables enables the appropriate transformations to be made as will be shown later.

A general statement in DDL* is defined by:-

(s) lhs = rhs

where s is an integer (statement number).

The '=' sign indicates transformation from rhs to lhs.

Terminal variables in DDL* are described by vectors t(s,j) where s is the number of the statement in which the terminal variable appears and j is an identifier of the vector within the statement. This identifier is necessary because more than one terminal vector may appear in a statement. The i th element in a terminal vector is referred to by t(s,j,i) and the constructs associated with such elements represent choices for the language writer or generator. In order to define a language, say DDL(n), a subset of the terminal variables in DDL* (t-variables) are associated with constructs in DDL(n) according to the following format:-

(s) t(s*,j,i) = rhs

where s* is the statement number in which vector t(s*,j) occurs on the rhs in DDL* or DDL(n), j is the identifier of the vector in s*, and i is the identifier of an element in t(s*,j). The rhs is often a source format in DDL(n), a software-module name or a parameter for code-generation by the DDL*+DDL(n) compiler. The rhs in a t-variable assignment may also be a further addition to the syntax of DDL*.

Transformations may be required between source formats in DDL(n) and DDL*. For example, an active construction in DDL* may be represented by a passive construction in DDL(n).

In the definitions of DDL(n) it will be required to allow entity declarations, entity-relation-declarations, entity-

mappings etc. to be identified in some way. For example in order to specify the formats for data-item specifications and level numbers in records and data structures in PL/I, COBOL and DBTG-DDL, and *'map'* them into DDL*, it is necessary to define a right-hand side thus:-

```
[level-no] [entity-name(1)] [entity-
definition(1)/] [* [level-no+inc)
[entity-name(j)] [entity-definition(j)/]]
```

It is to be noted that entity-name is followed by an integer in brackets. This is used to distinguish constructs or terms. If the number of terms or constructs in a production is a variable, then the notion (j) is used to indicate this, where j is an identifier which can be left undefined on specification in a rewrite rule (e.g. (s) (j) = (1:20))

The symbols '[' and ']' are used as brackets within syntax rules. Thus the clause x = [a/b]c means the same as x = ac/bc . The construction [a/] means *'an 'a' or nothing'*. A construct may be repeated an indefinite number of times. The symbol '*' is used to express this. The asterisk may appear once within a pair of brackets and then denotes that:-

(i) the contents of the brackets may be repeated any number of times, but

(ii) the last repetition ends at the point where the asterisk is written.

```
Thus:-          x  =  Y [a *b]

means           x  =  ya/yaba/yababa/ ......

and             x  =  y [*a] b

means           x  =  yb/yab/yaab/ ....
```

7. SOME MAJOR CONSTRUCTS AND RULES IN DDL*

```
(1) schema = t(1,1) [data-structure-division
                    [data-selection/][data-control-division/]
                    [data-organisation-division/]*] t(1,2)

(2) data-structure-division
                    = t (2,1) [[entity-specification *]
                    [* entity-relation-specification]*]t (2,2)

(50) data-selection-division  =  t(50,1)
                    [t(50,2)selection-heuristic*]t(50,3)

(52) data-control-division  =  t(52,1) [t(52,2)
                    error-action/t(52,3) privacy-
```

mechanism/t(52,4) integrity-
mechanism *]t (52,5)

(73) data-organisation-division = t(73,1)[*entity-mapping]
[*function-specification] t(73,2)

At present 89 such rules have been identified, to define DDL*

8. AN EXAMPLE OF A HYPOTHETICAL DATA-DESCRIPTION LANGUAGE (DDL (1))

Since the total language occurrence is defined by DDL* + DDL(1), the statement numbers in the definition of DDL(1) must not overlap with those of DDL*, nor must the lhs of any statement in DDL(1) be the same as any lhs in DDL*. Underlined terms are source strings. x occurring in t(s,j,x) signifies that each element in the vector is set equal to values on rhs, in the order in which they are given, and is generally used to refer to the selection of any element in the vector t(s,j).

(100) t(1,1,1) = FILE-DESCRIPTION

(101) t(1,2,1) = END-OF-DESCRIPTION

(102) t(2,1,1) = DATA-STRUCTURE

(103) t(2,2,1) = END-OF-STRUCTURE

(104) t(4,1,1) = RECORD

(105) t(4,1,2) = FRECORD

(106) t(4,1,3) = FILE

(107) t(4,1,4) = USER-LABEL

(108) t(5,1,1) = t(12,1,x) *t(13,1,x)

(109) t(7,1,1) = X

(110) t(7,1,2) = 9

(111) t(7,1,3) = BYTES

(112) t(11,1,x) = 0,1,2,..........

(113) t(7,1,3)t(11,1,x) = t(11,1,x)t(7,1,3)

(114) t(12,1,x) = A,B,C, ...7

(115) t(13,1,x) = t(12,1,x)/t(11,1,x)

(116) t(15,1,1) = SET-TYPE

(117) t(15,1,2) = RELATED-DATA

(118) t(16,1,x) = t(12,1,x)*t(13,1,x)

(119) t(17,1,1) = OWNS

(120) t(17,1,2) = IS-A-MEMBER-OF

(121) t(17,1,3) = HAS

(122) t(17,1,4) = IS-PART-OF

(123) t(73,1,1) = FILE-MANAGEMENT-SPECS

(124)	t(73,2,1)	=	END-OF-MANAGEMENT-SPECS
(125)	t(74,1,1)	=	MAP
(126)	t(74,2,1)	=	ONTO
(127)	t(7,1,4)	=	BLOCK
(128)	t(78,1,1)	=	CALLING
(129)	t(81,1,1)	=	ON
(130)	t(81,2,1)	=	DO
(131)	t(82,1,1)	=	FIND
(132)	t(84,1,1)	=	SERIAL-SEARCH
(133)	t(83,1,1)	=	VIA
(134)	t(80,1,1)	=	CHAINED-BLOCKS
(135)	t(7,1,5)	=	STANDARD-LABEL

9. THE MEANING OF THE t-VECTORS

(a) t(1,1), t(1,2), t(2,1), t(2,2), t(52,1), t(52,2), t(52,3), t(52,4), t(52,5), t(73,1), t(73,2) are shown in the rules given earlier.

(b) t(4,1) is a vector associated with data-entity-type specifications in DDL*.

(c) t(5,1) is associated with naming rules and can be traced via rules (114) and (115).

(d) t(7,1) is associated with entity - definitions.

(e) t(15,1) is a vector associated with entity-entity relationship types in DDL*.
t(17,1) is associated with logical relations within relationship types.

(f) t(74,1) and t(74,2) are associated with entity-entity transformations and mappings. (So is t(75,1) but this does not occur in the example).

(g) t(78,1), t(81,1), t(81,2), t(84,1), t(83,1) and t(80,1) are all associated with definition of the object machine that manages the other constructs in DDL* + DDL(1). They are associated with software - module and procedure names and code-generation parameters.

A sample DDL(1) program is as follows:-

```
FILE-DESCRIPTION
DATA-STRUCTURE
RECORD MYRECORD 300 BYTES
FRECORD PHYSICALRECORD
FILE LOGICALDATAFILE
FILE PHYSICALPRIMEDATAFILE
USER-LABEL  UL  STANDARD-LABEL
```

```
BLOCK DATABLOCK 1200 BYTES
UL OWNS MYRECORD SET-TYPE BASICSET
MYRECORD IS-A-MEMBER-OF SET-TYPE BASICSET
END-OF-STRUCTURE
FILE-MANAGEMENT-SPECS
MAP  MYRECORD  ONTO  PHYSICALRECORD
CALLING GENERALMAPPROC (MYRECORD, PHYSICALRECORD)
MAP PHYSICALRECORD ONTO DATABLOCK
CALLING PHYSICALMAPPROC (PHYSICALRECORD, DATABLOCK)
MAP DATABLOCK ONTO PHYSICALPRIMEDATAFILE CHAINED-BLOCKS
MAP LOGICDATAFILE ONTO PHYSICALPRIMEDATAFILE
    CALLING FILEMANAGER
ON FILE DO SERIAL-SEARCH VIA LOGICALDATAFILE
END-OF-MANAGEMENT-SPECS
END-OF-DESCRIPTION
```

This program is parsed following the t-vector definition rules and those of DDL*. For example, the statement:-

```
ON FIND DO SERIAL-SEARCH VIA LOGICALDATAFILE
```

leads to the recognition of a '*function-specification*' in DDL*.

10. THE PROBLEM OF MAPPING EXISTING DATA-DESCRIPTION LANGUAGES AND FACILITIES ONTO DDL*

I will concern myself with the problems of mapping PL/1, COBOL and DBTG(1971) record descriptions onto DDL*, and those of mapping the SET declarations of DBTG ('71) onto DDL*. They pose interesting problems because:-

(a) The COBOL, PL/1 and DBTG record descriptions contain data entity definitions, logical relationships in the form of level numbers and physical order specifications.

(b) The DBTG SET - OWNSHIP and MEMBERSHIP declarations are in a passive voice whereas they are in an active voice in DDL*.

11. MAPPING OF RECORD-DESCRIPTION ONTO DDL* CONSTRUCTS

The important factor in this mapping is that a record declaration expresses the fact that a group of entities are related to each other and provides a name for that relationship. A record is to be thought of as a type of set of related entities rather than a type of data entity in the data environment.

The necessary rules to transform a COBOL, PL/1 and DBTG - like format to DDL* are:-

```
(100)    t(17,1,1)          =    has-as-component-field
(101)    t(15,1,1)          =    RECORD
(102)    substructure(k)    =    [level-no] [entity-name(1)]
                                 [entity-definition(1)]
                                 [   [*level-no+inc]
                                     [entity-name(j)]
                                     [entity-definition(j)/]]

(103)    [*entity-name(1) has-as-component-field entity-name(j)]
         [entity-definition(1)][*entity-definition(j)]   =
          substructure(k)
(104)    [*RECORD t(16,1,1) has-as-component-field substruct-
          ure(k)]
         =  RECORD alpha [*alphanumeric][*substructure(k)]
(105)    level-no=1,2,3, . . .
(106)    inc = 1,2,3, . . .
```

Another aspect of COBOL, PL/1, DBTG-like record declarations is that the order of declarations of items within a record represents the physical order of the items as fields in the physical embodiment of the record.

This is expressed relatively simply in DDL* thus:-

replace (103) above by

```
(103)    [*entity-name(1) has-as-component-field
         entity-name(j)][entity-definition(1)/]
         [*entity-definition(j)][map entity-name(1) onto
         physical-record after substructure (k-1)]
         [*map entity-name(j+1) onto physical-record after
          entity-name(j)]
         =  substructure (k)
```

adding:-

```
(107)    t(4,1,1)     =    physical-record
(108)    t(74,1,1)    =    map
(109)    t(74,2,1)    =    onto
(110)    t(75,1,1)    =    after
(111)    at physical-record after substructure (0)
(112)    t(75,1,2)    =    at
```

where <u>map</u>, <u>onto</u> <u>at</u> and <u>after</u> are not source strings but codes interpreted by the DDL(n)+DDL* compiler or interpreter. They are underlined here to aid comprehension.

12. MAPPING OF DBTG SET-OWNERSHIP AND SET-MEMBERSHIP DECLARATIONS ONTO DDL*

The rules are straightforward and are given below.

(100) *<u>SET</u> [alpha][*alphanumeric][entity-name-1 <u>OWNS</u>
entity-name(j)][entity-name(j)
<u>IS-OWNED-BY</u> entity-name(1) <u>SET</u> [alpha][*alphanumeric]]
= <u>SET</u> [alpha][*alphanumeric][OWNER [IS/]
entity-name(1)][intermediate-clauses]
[*[MEMBER[IS/] entity-name(j)][intermediate-clauses]]

(101) t(15,1,1) = <u>SET</u>

(102) t(16,1,1) = [alpha][*alphanumeric]

(103) t(17,1,1) = <u>OWNS</u>

(104) t(17,1,2) = <u>IS-OWNED-BY</u>

Note that in the rules above, <u>SET</u>, <u>OWNS</u> and <u>IS-OWNED-BY</u> are not strings in DBTG-DDL but codes for the compiler or interpreter of DDL*+DDL(n). 'Intermediate-clauses' appearing on the rhs of (100) is shorthand for allowable constructs that can appear in DBTG-DDL in the positions indicated. Rule (100) is a transformation rule rather than a 'parsing' rule.

13. IMPLEMENTATION OF DDL*

There are two aspects to implementation that must be considered:

- Language DDL(n) definition
- Compiler generation from the definition

Language definition and compiler generation in the DDL* scheme can be implemented by the same process, since DDL* + DDL(n) define both the syntax and semantics of the data description language.

The function specifications in the data-organization-division define the details of the compiler's functions, the entity mappings in the same division defining the transformations made to the data as it progresses from one part of the data environment to another, or from one phase of processing to another.

As a first implementation, an interpretive scheme is considered, so that no code generation is involved. The DDL(n) compiler is therefore an interpreter of the production of DDL* + DDL(n) translation.

Thus language definition only need be considered in this case. Combined DDL* rewrite rules and those defining DDL(n) are input to the DDL* + DDL(n) translator to generate code for the interpreter. When the interpreter is operational it has two input streams:

(a) The current DDL(n) program.

(b) The DDL* + DDL(n) translator output.

For example, the language, DDL(1) specified earlier produces the following Polish productions for the program of the example:

```
t(1,1,1)"FILE-DESCRIPTION"t(2,1,1)
"DATA-STRUCTURE"t(4,1,1)"MYRECORD"
t(7,1,3)t(11,1,300)t(4,1,2)
"PHYSICAL RECORD"t(4,1,3) "PHYSICAL
PRIMEDATAFILE"t(4,1,4)"UL"...etc...
t(73,2,1)"END-OF-MANAGEMENT-SPECS"
t(1,2,1)"END-OF-DESCRIPTION"
```

The interpreter pops off this string (t(1,1,1) is at the top), the t variables acting as switches to the appropriate code which then pops off the operand(s) and processes accordingly thus:

(a) POP X

(b) <u>if</u> X = t(1,1,1) <u>then</u> START

<u>else if</u> X = t(1,2,1) <u>then</u>

FINISH <u>else</u> etc.

This system was designed to run transactionally, batched or interactively (but compiled).

The transaction mode was most similar to the process of receiving messages in L, understanding them, and processing the generated T* asynchronously (the user could fire off multiple transactions).

The system, at run time, had an internal language consisting of DDL* + DDL(1) + ····· + DDL(n). The user issued a message (transaction command) in DDL(i), which if passed for well-formedness, consistency and other pre-transactional levels, was finally structurally packed as described earlier. This packet was sent to the central transaction processor which then unpacked the message (equivalent to T*) and processed it. The user was informed that the transaction had been initiated as soon as the transaction processor set a flag (global) indicating that it had received the packed message from the DDL* + DDL(1) etc. processor.

The compiled system also interfaced with the transaction processor in the same way, and the user notification was done by a transaction/audit trail mechanism, or optionally interactively.

14. CURRENT WORK IN PROGRAMMING LANGUAGES

A system is being developed for the definition and running of experiments in Artificial Intelligence. A *"powerful"* language system provides the features required for a wide class of experiments - providing an integrated special purpose

operating system for AI research combined with a language to drive it and have access to its complete range of functions. Based on this common level (equivalent to L* in previous sections), languages are defined which are designed to model and specify environments, hypothesis formation methods, motivational systems and so on, for the purposes of the experiment in mind.

These languages, like the DDL(i) of the previous section are defined by linking to the common underlying language system. This system is described in [21,22].

In general, common mechanisms and system facilities, are required in different mixes by different applications. At the moment, a user is locked into either FORTRAN, COBOL, SIMULA, DMS1100, APL, ALGOL, LISP or whatever other language is available on a system, whereas, if a user is involved in serious programming, he requires features from each of them at different times in different combinations, and demands efficient support for those mixes.

My thesis is that, as is likely in the case of natural language systems, a common powerful language level (L*) is required, along with mechanisms for defining an L language and associated mappings between L and L*, with added entailments if required. Further to this a compiled/linked virtual machine system is required to provide the efficiency for such a system. Perhaps then we could have access to a wide variety of mechanisms now hardly available in current languages, and also perhaps stop language bandwagonning. Combined with common data description at the L* level, we would also achieve an independence of compiled programs from the language that defined them, and provide a high degree of portability.

REFERENCES

[1] G.B. Bobrow and A. Collins, '<u>Representation and Understanding: Studies in Cognitive Science</u>', Academic Press, New York, 1975.

[2] '<u>The Compiler-compiler</u>', Annual Review in Automatic Programming 3, 1963, 229-275.

[3] P.E. Bryant, '<u>Perception and Understanding in Young Children</u>', Basic Books, N.Y. 1974.

[4] R. Carnap, '<u>Meaning and Necessity</u>', University of Chicago Press, Chicago: 1956.

[5] N. Chomsky, '<u>Syntactic Structures</u>', Mouton: The Hague, 1957.

[6] H.H. Clark and W.G. Chase, '<u>On the process of comparing sentences against pictures</u>', Cognitive Psychology, 3, 1972, 472-517.

[7] J.A. Fodor, <u>The Language of Thought</u>. Crowell: N.Y., 1975.

[8] J.A. Fodor, T. Bever and M. Garrett, '<u>The Psychology of Language</u>', McGraw-Hill, N.Y., 1974.

[9] J.D. Fodor, Semantics: To be published.

[10] J. Greene, 'Psycholinguistics: Chomsky and Psychology' 1972, Penguin, Harmondsworth, U.K.

[11] J.J. Katz, 'Semantic Theory', Harper, N.Y., 1972.

[12] J.J. Katz and J.A. Fodor, 'The structure of semantic theory', Language, 39, 1963, 170-210.

[13] N. Marslin-Wilson, 'Speech shadowing and speech perception'. Ph.D. Dissertation, M.I.T., Cambridge, Mass. 1973.

[14] G.A. Miller and P.N. Johnson-Laird, 'Perception and Language'. to be published.

[15] K.H. Pribram, 'Languages of the Brain', 1971, Prentice-Hall, Englewood Cliffs, N.Y.

[16] S. Rosenberg, Modelling semantic memory: effects of presenting semantic information in different modalities. Ph.D. Dissertation, Carnegie-Mellon, 1974.

[17] R.W. Sperry, 'The eye and the brain' Scientific American, May, 1956, 48-52.

[18] P.C. Wason and P.N. Johnson-Laird, 'Psychology of Reasoning: Structure and Content'. 1972, Harvard U.P., Cambridge, Mass.

[19] L. Wittgenstein, 'Philosophical Investigations;, Blackwell, Oxford, 1953.

[20] G.L. Wolfendale, 'A system for the definition of the syntax and semantics of data description languages. In Gunther et al (eds), International Computing Symposium, 1973; North Holland, Hague, 1974, 517-525.

[21] G.L. Wolfendale, 'A Survey of Recent Work and Ideas in Data Base Management' in Data Base Management Systems, G.L. Wolfendale (ed), ANU Press, pp.1-21.

[22] G.L. Wolfendale, 'PLEXIS User's and Reference Manual - VERSION I' ANU Computer Centre, Technical Report, 51, May 1976.

[23] G.L. Wolfendale, 'PCM: The Plexis control machine', ANU Computer Centre, Technical Report, 56, November, 1976.

17

An Entomological Approach to the Design of Programming Language Systems

Garth L. Wolfendale

'No one ever travels so high as he who knows not where he is going'

Oliver Cromwell

1. THE REASONS UNDERLYING THE ENTOMOLOGICAL APPROACH

With the advent of cheap microprocessors and memory, we are faced with the exciting prospect of changing the way we tackle problems. Instead of painstakingly analysing problems to be solved and then writing programs to solve them, I believe that we have the technology to raise the level of 'combat' for the user from 'hand-to-hand' struggle with the problem to that of tactical and strategic planning and warfare.

The old-time programmer was a Samurai whereas the programmer of the future will be a General.

Of course we are not there yet. We do not know how to build or design systems consisting of many (even thousands) of processors (Kuck [1]; Sullivan and Bashkow [3]); but this present paper is a light hearted attempt to provide a coherent framework for future developments.

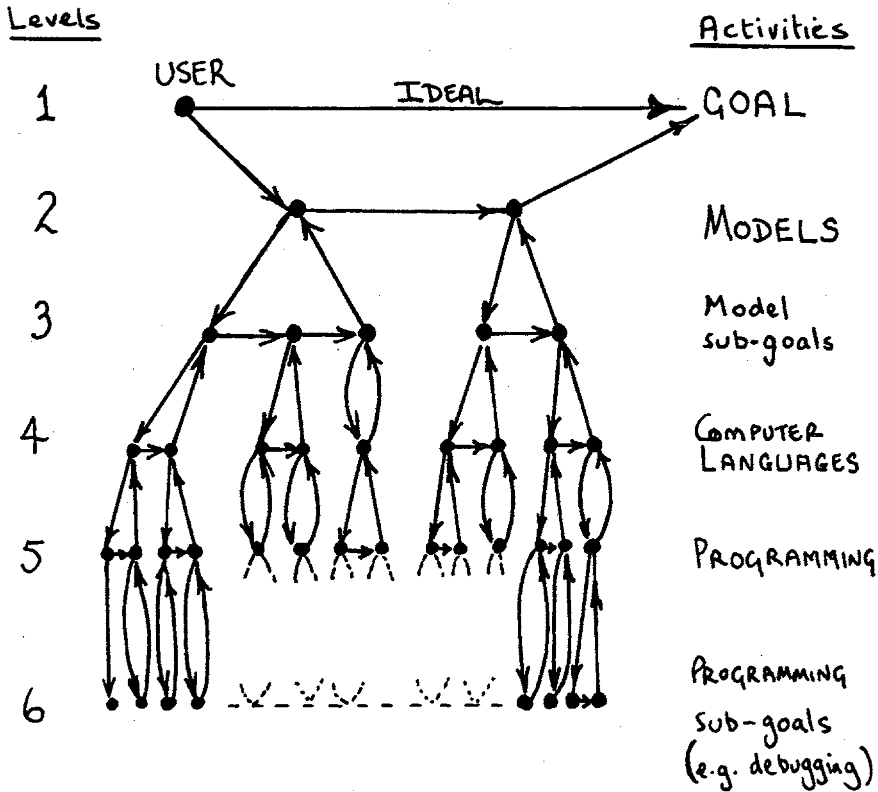
The model used was the ant-hill, which has solved the complex problems of survival for millions of years by means of cooperating micro-elements or processors (ants), specialisation (queen, workers, etc.) and communications.

The paper published in these proceedings [5] compares the work on human language processing with programming language systems and postulates lessons that can be learned by such a study. However, during the necessary incubation and study period leading up to the paper, I was hit by certain facts:-

(a) Most users of computers are being forced to write unsatisfactory programs in languages ill-suited for their requirements. In fact the question was raised:-

> *'why should most users have to program as we know it, and why are they faced with a clutter of inadequate languages and commands?'*

If you consider the average user who 'appears' to be a programmer, in fact he has become far removed from the goals that are of greatest relevance to him, and far removed from his field of competence. Fig. 1 illustrates just how far removed programming can be from the main activities and goals of a user.



The devious paths a user must take
in order to achieve relevant goals.

Fig. 1

(b) Most research on cognitive processes in human beings seems to create *more* complexity and uncertainty rather than leading to a clearer understanding.

(c) Psychological studies on animals and insects reveals highly organised and sophisticated individual and group systems. Perhaps this is where insights are more likely to come from.

(d) There are ad hoc operating systems and ad hoc language implementations, packages and command language systems, but

not what can be discerned as *coherent* programming language systems.

However, this heap of sticks was placed into a pile that constitutes my previous mentioned paper.

Recently and in my past I have been engaged in the design, development and implementation of database management systems, operating systems and communications networks. There are asynchrony, uncertainty, resource limitations, unsolved error handling problems by the ton in such systems: so what impact do such systems have on our attempts to order our programming language system world? Perhaps we are attempting to order the unorderable (like *ordering* human society). So by the time came to present my other paper in these proceedings; my pile of sticks had become tinder.

Finally, I attended the talks of Professor Dijkstra. I heard the following:-

(i) 'Programming is the province of hard engineering science.'

He meant that programmers and users as we know them would be left *even more* without adequate resources to enable them to solve their problems and achieve their goals. Surely a computer system should provide a *rich* environment for users to work creatively in. More restrictions and formalism *reduces* that richness and accessibility of computer based resources.

(ii) The word 'invariant' was mentioned many times. I began to think about the word and the thought came to me - 'I wonder if there are many other *ants* that are fundamental to programming semantics?'

Now the pile of sticks went up in flames. I realised that these 'invariants' reflected the *ideal* world of the formal programmer, but when compared with the world in which I, and many others are working they are totally inadequate to deal with the complexity and uncertainty of the systems which we must develop and implement.

To reiterate somewhat:

We require:-

(a) Rich environments and systems to enable users to carry out their work on computers creatively and as close as possible to the level at which they prefer to work,

(b) Supportive systems and semantics that reflect the complexities of real systems, with their inherent uncertainties.

Society functions well even though there is a high level of uncertainty, which is not due to *errors* but due to inherent variability in humans and human organisations. Suppose that we design systems in such a way that they

reflect such naturally occuring highly successful systems, with entities in them that can be described functionally and precisely, as can their interactions. Such systems would allow the user to muster and control forces and act as strategist in order to solve his problems or produce a desired system.

Thus the *ant* community came to mind, leading to an analysis presented in the next section.

2. A SEMANTIC SYSTEM MODELLED ON ANTS

The world of the ant

To the ant, it is considered, there exist things natural and things superstitious. Things natural consist of other ants and foreign ants, and also structures in the physical world, referred to here as cells, cell structures and sets. Also there are classes of entity which will remain, undefined referred to as 'things' and 'conglomerates'.

Fig. 2 tabulates the ant's world (natural).

In the superstitious world, things happen to change the natural world affecting both ants and things. To an ant, for example, being trodden on is an act of a 'superstitious' level entity. These are briefly listed in fig. 3.

In general terms, ants function to maintain order, whereas demons and other entities in the 'superstitious' world function to destroy order. Thus these latter are *included* in any system description and are not treated as merely random or unfortunate errors.

Classes of ants

Fig. 4 gives the main classes of ant, in terms of their overall functions. A more detailed breakdown is given in figs. 5 to 13. Thus we have a complete army of ants and vast range of functions at our disposal

Classes of 'Demon'

As was stated earlier, forces of disorder are treated as an essential part of the system being described.

Fig. 13 appropriately provides a demonology, showing the different kinds of disruptive entity in the system.

The figures are self explanatory, providing very broad outlines of the system.

THE ANTS' WORLD (NATURAL)

- CELLS
- CELL STRUCTURES/SETS
- THINGS AND CONGLOMORATES
- FOREIGN ANTS

A GENERIC WORLD OBJECT IS REFERRED TO AS AN ENTITY OR WORLD ENTITY.

Fig. 2

THE ANTS' WORLD (SUPERSTITIOUS)

.NOTE.

TO ANTS HUMAN BEINGS WITH BIG BOOTS, DDT SPRAYS ETC. ARE ESSENTIALLY SUPERSTITIOUS OBJECTS.

- DEMONS
- GREMLINS
- IMPS
- OBSERVERS .... HUMAN OR OTHER USERS OF ANT SYSTEMS.

Fig. 3

CLASSES OF ANTS

OFFICERS

MESSENGERS

INFORMATION GATHERERS/INTELLIGENCE

ENGINEERS

DEFENCE/SUPPLY/SUPPORT

INFANTRY/WORKERS

ADMINISTRATION/RECORDS

DEGENERATES

Fig. 4

ANTHOLOGY

OFFICERS

ANT	OBJECT	FUNCTION
COMMANDANT	STRATEGY	IMPLEMENT/TRIGGER
SERGEANT	TACTICS	IMPLEMENT/CARRY OUT
LIEUTENANT	MOVES/ MANOEUVERS	CARRY OUT
FORMANT	FORMATIONS	CREATE & MAINTAIN
DOMINANT	DEVIANTS OR PEDANTS	APPLIES REPELLANTS TO DEVIANTS OR PEDANTS OR POSSIBLY DISINFECTANTS.

Fig. 5

MESSENGERS

ANT	OBJECT	FUNCTION
COMMUNICANT	ANTS(INFORMANTS AND RECEPTANTS	FORMS AND MAINTAINS CHANNELS BETWEEN ANTS
INFORMANT	COMMUNICANTS	SENDS INFORMATION TO RECEPTANTS VIA COMMUNICANTS
RECEPTANTS	COMMUNICANTS	RECEIVES INFO. FROM INFORMANTS VIA COMMUNICANTS

Fig. 6

INFORMATION AND

INTELLIGENCE

ANT	OBJECT	FUNCTION
ASPIRANT	ENTITY-PREDICATES IN WORLD	.IF TRUE (TRUANT) .IF FALSE (DEVIANT)
OBSERVANT	ANTS?ENTITIES	REPORTS ON STATES OF ENTITIES OR ANTS
DILETTANT(E)	WORLD	SEEKS OUT INFORMATION ABOUT WORLD
COGNISANT	ENTITY PREDICATES ANT PREDICATES	ACTS AS A RESOURCE OF TRUE OR FALSE STATEMENTS ABOUT WORLD AND ANTS. ANY ANT CAN ASK A COGNISANT ANYTHING AND IT CAN TELL THE ANT.
EXPECTANT	ENTITY AND ANT PREDICATES	WAITS FOR STATEMENTS ABOUT ANTS AND THE WORLD TO BECOME TRUE OR FALSE. DOES NOT BECOME A DEVIANT AS AN ASPIRANT DOES
PEDANT	ENTITY AND ANT	CONSISTENTLY MAINTAINS THAT PREDICATES ARE TRUE OR FALSE, HARD TO CHANGE - ONLY BY REPELLANTS OR DISINFECTANTS.

Fig. 7

ENGINEERS,

ANT	OBJECT	FUNCTION
SUPERINTENDANTS	VARIANTS AND CONSULTANTS	CARRIED OUT PROJECTS USING VARIANTS.GUIDED BY CONSULTANTS AT ENORMOUS FEES.
CONSULTANTS	SUPERINTENDANTS	DESIGN AND DIRECT PROJECTS
VARIANTS	WORLD ENTITIES	CAUSE CHANGES IN WORLD, CONSTRUCT THINGS ETC.

Fig. 8

DEFENCE/SUPPORT/SUPPLY

ANT	OBJECT	FUNCTION
SUPPLIANT	WORLD ENTITIES/ANTS	REQUESTS THE SOURCE OF ALL THINGS AND ANTS FOR RESOURCES AND SUPPLIES.
INVARIANT	ENTITY AND ANT PREDICATES	APPLIES CONTROLS AND DEFENSIVE ACTIONS TO MAINTAIN TRUTH OR FALSITY OF PREDICATES
DEFENDANT	ENTITIES OR ANTS	PROTECTS ENTITIES AND ANTS THEMSELVES FROM DESTRUCTION OR DEVIATION
DISINFECTANT	ANTS	KILLS ANTS. THEY BECOME MORDANTS
REPELLANT	ANTS	DEVIATES ANTS FROM NORMAL FUNCTIONS. THEY BECOME DEVIANTS

Fig. 9

INFANTRY/WORKERS

ANT	OBJECT	FUNCTION
OPERANTS	WORLD, ANTS	GENERAL FUNCTIONS. DO WHATEVER THEY ARE TOLD.
SERVANTS	WORLD, ANTS	TOTALLY DISPENSIBLE OPERANTS
ANTAGONISTS	ANTS(FOREIGN), GREMLINS, IMPS	FRONT LINE ATTACK OR DEFENCE AGAINST EVIL FORCES

Fig. 10

ADMINISTRATION/RECORDS

ANT	OBJECT	FUNCTION
DESKANTS	ENTITY PREDICATES ANT PREDICATES	STORES INFORMATION GIVES TO COGNISANTS. MADE AVAILABLE TO OBSERVERS
ACCOUNTANTS	ANT, WORLD RESOURCES	BALANCE ALL BOOKS KEEP TABS ON REQUESTS FROM SUPPLIANTS

Fig. 11

DEGENERATES

ANT	OBJECT	FUNCTION
DEVIANTS	ANTS	ANTS WHOSE FUNCTION HAS BEEN HINDERED OR INTERFERED WITH OR BEHAVIOUR HAS BECOME UNPRED-ICTABLE
DECAD(A)NTS	ANTS, ENTITIES	ABSORB RESOURCES OCCUPY CELLS, CAUSE ANTS TO BECOME DEVIANTS OR DECADANTS
MIGRANT	ANT	AN ANT THAT IS INVOLVED IN A WAVE OF ANTS THAT MOVES IN WORLD
MORDANT	ANT	A DEAD ANT
ELEGANT	ANT	A NICE LOOKING BUT RATHER USELESS ANT. OF INTEREST TO OBSERVERS.
RESULTANT	ANT	AN ANT OF GENERAL INTEREST TO AN OBSERVER BUT NOT TO ANTS.

Fig. 12

DEMONOLOGY

DEMON	GREMLINS	ALL POWERFUL DESTRUCTIVE OR DISRUPTIVE FORCE
GREMLIN	IMPS, ANTS WORLD.	. ATTACKS ANTS . CHANGES OR . DESTROYS ENTITIES . WIDE RANGE OF FUNCTIONS . VERY POWERFUL
IMPS	GREMLINS, ANTS ENTITIES	. A MORE SPECIFIC AND LIMITED GREMLIN . TENDS TO HAVE LIMITED POWER AND LIFE.
OBSERVERS	WORLD AND ANT SYSTEM	THESE ARE USUALLY HUMAN BEINGS. MORE DESTRUCTIVE THAN DEMONS

Fig. 13

3. SOME ASPECTS OF THE SYSTEM

Sullivan and Bashkow [3] recently explored necessary functions (e.g. broadcasting) for the support of large numbers (thousands) or processors involved in the processing of parallel but interdependent tasks.

There has been continual interest in independent but cooperating automata (e.g. Von Neumann [4]), but so far there has been little impact of the theoretical work on 'real' programming language systems. However, we are at the stage where we can build, cheaply, the hardware to support large numbers of processes, and it is now a ripe time to consider approaching systems in this way.

The ant system just described is not proposed as a serious system as it stands since so many constructs need precise definition. However, it is intended to point out that such a system is both desirable and necessary if programming is to evolve from the spanner and screwdriver level that it is at, presently (no matter how formally sound it may be).

That the ant system is useful, even as it stands can easily be demonstrated.

Consider the simple task of finding a cell A[i] in a set of cells A[1],....,A[n] containing the value X. A *format* of *aspirants* can be set up using a *formant*. Each aspirant looks for a cell in range A[1], ..., A[n], and its *predicate* to test is A[i] = X. If found, it becomes an *informant* telling its *lieutenant* of the truth of the predicate. No serial searching, all done in one *tactic* of 2 *moves*.

Also, this is trivial to program in an ALGOL-like language. However, consider the same problem but involving an *IMP* with its function of moving X in and out of cells in A[1], ... A[n] in a *random* fashion.

Now program this in an ALGOL or FORTRAN-like language. In the ant system, the solution would still be a 2-move tactic. In fact the previous tactic could be improved to use *expectants* instead of *aspirants* to increase the robustness in the face of the IMP.

In the network system that we have implemented at ANU, we find that many of the aspects of the system have direct counterparts in the ant system. A few are listed below:

Network Entities	Ant entities
Line handlers	Communicants/Informants
Message processors, parsers	Aspirants
Diagnostics, buffer, line, node, state monitors	Dilettants
Event flag waiters	Expectants
Configuration parameters	Pedants
Buffer pool management	Variants/Suppliants
Protection mechanisms	Invariants/Defendants
Hard, soft error actions	Disinfectants/Repellants
Service Routines	Operants
Monitoring, traffic measurement	Deskants/Accountants

At run-time, as we de-bug, we find many IMPS, GREMLINS and DEMONS creating deviants, decadents and mordants. NO elegants.

REFERENCES

[1] D. Kuck, 'Parallel Processor Architecture - A Survey'. 1975 Sagamore Conference on Parallel Processing.

[2] O.G. Selfridge, 'Pandemonium: A Paradigm for Learning' in Mechanisation of Thought Processes, NPL, Symposium 10, 1956, 511-526.

[3] H. Sullivan and T.R. Bashkow, 'A Large Scale, Homogeneous, Fully Distributed Parallel Machine' in 4th Annual Symposium on Computer Architecture, IEEE Computer Society, 1977, pp.105-124.

[4] J. von Neumann, 'Theory of Self-Reproducing Automata', A.W. Burke (Ed), U. of Illinois, 1966.

[5] G.L. Wolfendale, 'Computer Language Systems and Cognitive Processes'. These proceedings.

Contributor's Affiliations

L. Allison,
Dept. of Computer Science,
University of Melbourne,
Parkville, VIC 3052.

C. J. Barter,
Dept. of Computing Science,
University of Adelaide,
Adelaide, SA 5001.

J. A. Campbell,
Dept. of Mathematics,
Newcastle University,
NSW 2308.

E. W. Dijkstra,
Professor Extraordinarius,
Technological University,
Eindhoven, The Netherlands.

K. R. Elz,
Dept. of Computer Science,
University of Melbourne,
Parkville, VIC 3052.

J. B. Hext,
Dept. of Computer Science,
University of Sydney,
Sydney, NSW 2006.

C. B. Mason,
Aust. Atomic Energy Commission,
Research Establishment,
Lucas Heights, NSW 2232.

P. W. Milne,
Divn. of Computing Research,
C.S.I.R.O.
Canberra, ACT 2601.

M. C. Newey,
Computer Centre,
Australian National University,
Canberra, ACT 2600.

P. C. Poole,
Dept. of Computer Science,
University of Melbourne,
Parkville, VIC 3052.

P. A. Pritchard,
Computer Centre,
Australian National University,
Canberra, ACT 2600.

K. A. Robinson,
Department of Computing Science,
University of New South Wales,
Kensington, NSW 2033.

J. S. Rohl,
Dept. of Computer Science,
University of Western Australia,
Nedlands, WA 6009.

R. W. Topor,
Dept. of Computing Science,
Monash University,
Clayton, VIC 3168.

W. D. Wilde,
Swinburne College of Technology,
Hawthorn, VIC 3122.

TITLE: PROGRAMMING LANGUAGE SYSTEMS

AUTHORS: ALLISON, ELZ, POOLE: Melbourne University; BARTER, KIDMAN: University of Adelaide; CAMPBELL: Newcastle University; DIJKSTRA: Technological University, Eindhoven, The Netherlands; HEXT: Sydney University; MASON: Australian Atomic Energy Commission; MILNE: Divn. of Computing Research, CSIRO; NEWEY, PRITCHARD, WOLFENDALE: Australian National University; ROBINSON: New South Wales University; ROHL: University of Western Australia; TOPOR: Monash University; WILDE: Swinburne College of Technology.

KEYWORDS: Abstraction, Abstract machines, Algebraic simplification, **Algol**, Ants, Axiomatization, Backtracking, **Cobol**, Cognitive processes, Command languages, Communication, Compact representations, Compiling, Concurrency, Continuation Induction, Control structures, Correctness, Co-routines, Data abstraction, Data description languages, Data structures, Data transformations, Data types, Declarations, Denotational semantics, Diagnostics, Dynamic arrays, Efficiency, Expression languages, Formal semantics, **Fortran**, Guarded commands, Hierarchical structure, Hoare semantics, Inductive assertions, Intermittent assertions, Invariants, Iteration, **Janus**, Job control languages, Lattices, **LCF**, Linear search theorem, Linguistics, **Lisp**, Methodology, Nondeterminism, Operators, Optimisation, Parameter passing, Parsing, **Pascal**, Phrase Structures, **Planner**, Portability, Predicate logic, Predicate Transformers, Preprocessing, Private language, Problem solving, Problem transformation, Program editors, Program structure, Program transformations, Programming, Programming languages, Programming systems, Programs, Protection, Proof rules, Psychology, Queues, Recursion, Reliability, Scott logics, Semantics, **Simula 67**, Software tools, Stacks, Standards, Structured programming, Subgoal induction, Symbolic computation, Syntax, Syntax graphs, Translation, **Uncol**, Variables, Verification, Wirth sequences.

Australian National University Press Canberra 1978

ISBN 0 7081 0493 2