Programming Denotational Semantics

Lloyd Allison
Department of Computer Science, University of Western Australia, Nedlands 6009, Western Australia

The denotational semantics of a simple language which includes jumps are programmed in Pascal to give an interpreter. By concentrating on the final state of a program the semantics are directly coded in Pascal with only slight modification to the semantic equations. The interpreter was produced as easily as the formal definition of the language and makes a reference implementation and development testbed. By using a widespread metalanguage such as Pascal this definition can be widely understood and executed.

INTRODUCTION

The denotational semantics definition of a language provides a method of calculating the high order function denoted by a program. In this paper the definition of a simple language 'contlang' is coded directly into Pascal; the result allows the denoted functions to be applied—it is an interpreter for contlang. As the name suggests, the semantics include continuations as the meaning of labels and other sequencers.

Mosses[1] recognized that a notation for denotational semantics which was itself formally defined was amenable to computer processing. He produced the Semantics Implementation System (SIS) which directly implements the semantics of a programming language by translation to a lambda-calculus based language and interpretation. Bodwin *et al.*[2] describe some experiences in using SIS. Efficiency is improved if a formal definition is used to generate a true compiler. Such systems—true compiler-compilers—have been produced by Paulson,[3] Raskovsky[4] and Sethi.[5]

The use of a conventional programming language as a meta-language which is certainly amenable to computer processing, in particular Algol 68, has been proposed by Pagan–both for a VDL style definition[6] and a denotational semantics definition.[7]

In the latter paper the *direct* semantics of a language 'loop' are coded in an extended Algol 68 which includes partial parameterization. Pagan argues that an executable definition comparable to a formal definition is not achievable 'in the case of strict Algol 68 (and certainly not in the case of other mainstream languages)'.

Here it is shown how to avoid the difficulties raised by Pagan using Pascal—a more modest language than Algol 68. The technique is also applied to coding *continuation* semantics handling full jumps.

The advantages of an executable semantic definition are obvious. Such a definition is compiled and checked automatically and it can be run and tested. It provides a reference system—albeit a possibly inefficient one. The language can be run as it is being developed! Changes to the semantic equations can be tried, and program examples run directly. The advantage of using a widely available language like Pascal as the meta-language is precisely that it is widely available, and most Pascal compilers are ruthless in their type-checking and give good error messages.

Finally there is a resistance among programmers to the mathematics of, say, denotational semantics and it is hoped that the runnable mathematics might seduce them.

It is assumed that the reader is familiar with the aims of denotational semantics. A certain amount of its notation is used in the paper but as the theme is to translate it into Pascal, any strange looking formula probably reappears in a more familiar notation nearby. Gordon[8] provides a readable introduction to denotational semantics; Milne and Strachey[9] is a reference work.

CONTLANG

Contlang is not a very useful programming language but it contains some of the more difficult language features—**goto**, **valof** ⟨*statement*⟩, **resultis** ⟨*exp*⟩. It is loosely based on the language used by Strachey and Wadsworth[10] to introduce continuations. This paper is neutral about the desirability of various language features and it does not extend denotational theory; it is solely concerned with implementing formal definitions.

Contlang operates only on integers but 0 can be used for false and 1 for true; a reasonable set of operators is provided.

The structure of contlang programs is specified by abstract syntax given in BNF in Fig. 1. This grammar happens to be ambiguous if used to parse linear strings, representing programs but it gives the shape of parse trees and, as is usual, the semantics are given in terms of it. In order to parse linear strings, concrete syntax for

⟨*statement*⟩ ::= ⟨*label*⟩⟨*stat*⟩ | ⟨*stat*⟩
⟨*stat*⟩ ::= ⟨*id*⟩ := ⟨*exp*⟩ |
if ⟨*exp*⟩ **then** ⟨*statement*⟩ **else** ⟨*statement*⟩ |
while ⟨*exp*⟩ **do** ⟨*statement*⟩ |
skip |
⟨*statement*⟩; ⟨*statement*⟩ |
(⟨*statement*⟩) |
resultis ⟨*exp*⟩ |
goto ⟨*label*⟩
⟨*exp*⟩ ::= ⟨*exp*⟩ ⟨*op*⟩ ⟨*exp*⟩ |
⟨*id*⟩ |
⟨*int*⟩ |
valof ⟨*statement*⟩
⟨*id*⟩ ::= *A* | *B* | *C* | . . .
⟨*label*⟩ ::= ⟨*int*⟩
⟨*op*⟩ ::= + | − | * | < | ≤ | > | ≥ | = | ≠

Figure 1. Abstract syntax.

CCC-0010-4620/83/0026-0164 $06.00

⟨*statement*⟩ ::= ⟨*label*⟩ : ⟨*stat*⟩ | ⟨*stat*⟩

⟨*stat*⟩ ::= ⟨*id*⟩ := ⟨*exp*⟩ |
if ⟨*exp*⟩ **then** ⟨*statement*⟩ **else** ⟨*statement*⟩ |
while ⟨*exp*⟩ **do** ⟨*statement*⟩ |
skip |
(⟨*statementlist*⟩) |
resultis ⟨*exp*⟩ |
goto ⟨*label*⟩

⟨*statementlist*⟩ ::= ⟨*statementlist*⟩ ; ⟨*statement*⟩ |
⟨*statement*⟩

⟨*exp*⟩ ::= ⟨*sexp*⟩ ⟨*relop*⟩ ⟨*sexp*⟩ | ⟨*sexp*⟩
⟨*sexp*⟩ ::= ⟨*sexp*⟩ ⟨*addop*⟩ ⟨*factor*⟩ | ⟨*factor*⟩
⟨*factor*⟩ ::= ⟨*factor*⟩ * ⟨*unit*⟩ | ⟨*unit*⟩
⟨*unit*⟩ ::= − ⟨*unit*⟩ | ⟨*opd*⟩
⟨*opd*⟩ ::= ⟨*id*⟩ | ⟨*int*⟩ |
(⟨*exp*⟩) | **valof** ⟨*statement*⟩

⟨*id*⟩ ::= *A* | *B* | *C* | . . .
⟨*label*⟩ ::= ⟨*int*⟩
⟨*addop*⟩ ::= + | −
⟨*relop*⟩ ::= < | ≤ | > | ≥ | = | ≠

Figure 2. Concrete syntax.

contlang is given in Fig. 2; the interpreter includes a recursive descent parser based on this latter grammar. The parser builds a tree corresponding to the abstract syntax.

The rather few context sensitive restrictions of contlang, such as the correct declaration of labels, are presumed to be satisfied, although a production interpreter would have to check them.

SEMANTICS

The semantic equations of contlang are modelled as closely as possible on those in Ref. 10. Contlang has no notion of a location or a reference so a *state* is a mapping of identifiers to current values; e.g.

$$\sigma \in state = id \to value$$

so that $\sigma(\text{'}x\text{'})$ is the current value of x. In assignment, it is necessary to *update* the state function for the new value of an identifier

$$\sigma[v/i]\,(y) = \textbf{if } y = i \textbf{ then } v \textbf{ else } \sigma(y)$$

An *environment* provides an interpretation of objects which have *scope*. Only labels have scope in contlang, so an environment maps a label to its meaning. The meaning of a label is a continuation, that is to say a state transformation, or a computation from the label until the program stops, e.g.

$$\rho \in env = label \to cont$$
$$\rho(99) \in cont = state \to state$$
$$(\rho(99))\ \sigma = \sigma' \in state$$

For ease of parsing, labels are Pascal-style integer labels.

The notion of a continuation is not as mysterious as is often suggested—it is simply a more general variety of composition. Where for composition $(f \circ g)(x)$, f is applied to the result of g applied to x, we can have $(g'(f))(x)$ in which f is a parameter to g'; f is given as a continuation to g'. The way that continuations are used, we can think of g' acting on x and then normally calling f, but it has the option not to, and this escape is used when modelling jumps. A return address to a procedure is a form of continuation—normally the procedure will execute or carry on from the return address when finished, but it has the option to **goto** somewhere else.

An expression produces a value and possibly a change of state so the idea of an expression kontinuation (*sic*) is introduced to specify what to do with the value, e.g.

$$\kappa \in Kont = value \to cont$$

this is particularly clear in structured statements where a controlling expression selects a continuation out of alternatives.

The meaning of statements is given by

$$\mathbb{P}\text{: } Cmd \to env \to cont \to state \to state$$

which can be read as the meaning of a command, given an environment to interpret labels, and given a continuation to carry out afterwards, is a state transformation. The meaning of expressions is given by

$$\mathscr{E}\text{: } Exp \to env \to kont \to state \to state$$

or, the meaning of an expression, given an environment, and given a kontinuation which will use the value of the expression and finish the program, is a state transformation.

The various function domains are summarized in Fig. 3.

$\sigma \in$ $state = id \to value$
$\theta \in$ continuation $= state \to state$
$\kappa \in expression$ kontinuation $= value \to state \to state$
$\rho \in$ environment $= (label \to cont) \times kont$
$\mathscr{E}$ $: exp \to env \to kont \to state \to state$
$\mathbb{P}$ $: cmd \to env \to cont \to state \to state$

Figure 3. Semantic domains.

The semantic equations for expressions are given in Fig. 4. As an example, the meaning of an identifier expression given an environment, a kontinuation and a state is the given kontinuation applied to the value of the identifier in the state.

$$\mathscr{E}[\![\langle int\rangle]\!]\rho\kappa = \kappa(value(\langle int\rangle))$$
$$\mathscr{E}[\![\langle id\rangle]\!]\rho\kappa\sigma = \kappa(\sigma(\langle id\rangle))\sigma$$
$$\mathscr{E}[\![\textbf{valof } s]\!]\rho\kappa = \mathbb{P}[\![s]\!]\rho[\kappa/reslabel]\{fail\}$$
$$\mathscr{E}[\![e1 + e2]\!]\rho\kappa = \mathscr{E}[\![e1]\!]\rho\ \{\lambda x, \sigma \cdot \mathscr{E}[\![e2]\!]\rho\{\lambda y, \sigma' \cdot \kappa(x + y)\sigma'\}\sigma\}$$

Figure 4. Expression semantics.

The equations for statements are given in Fig. 5. To understand the equation for the **if** statement, note that

$$\mathbb{P}[\![id := E]\!]\rho\theta\sigma = \mathscr{E}[\![E]\!]\rho\{\lambda rhs, \sigma \cdot \theta(\sigma[rhs/id])\}\sigma$$
$$\mathbb{P}[\![\textbf{if } E \textbf{ then } s1 \textbf{ else } s2]\!]\rho\theta = \mathscr{E}[\![E]\!]\rho\{cond\ (\mathbb{P}[\![s1]\!]\rho\theta, \mathbb{P}[\![s2]\!]\rho\theta)\}$$
$$\mathbb{P}[\![\textbf{while } E \textbf{ do } s]\!]\rho\theta = Y(\lambda\theta'.\ \mathscr{E}[\![E]\!]\rho\{cond\ (\mathbb{P}[\![s]\!]\rho\theta', \theta)\})$$
$$\mathbb{P}[\![\textbf{skip}]\!]\rho\theta = \theta$$
$$\mathbb{P}[\![s1\,;\,s2]\!]\rho\theta = \mathbb{P}[\![s1]\!]\rho\{\mathbb{P}[\![s2]\!]\rho\theta\}$$
$$\mathbb{P}[\![(lab_1 : s_1\,;\, lab_2 : s_2\,;\, \ldots;\, lab_N : s_N S)]\!]\rho\theta = \theta_1$$
where $(\theta_1, \theta_2, \ldots, \theta_N) =$
$$Y\lambda(\theta_1, \theta_2, \ldots, \theta_N).\quad \theta_1 = \mathbb{P}[\![s_1]\!]\rho'\theta_2$$
$$\ldots$$
$$\theta_N = \mathbb{P}[\![s_N]\!]\rho'\theta$$
$$\rho' = \rho[\theta_1 \ldots \theta_N/lab_1 \ldots lab_N]$$
$$\mathbb{P}[\![\textbf{resultis } E]\!]\rho\theta = \mathscr{E}[\![E]\!]\rho\{\rho(reslabel)\}$$
$$\mathbb{P}[\![\textbf{goto } l]\!]\rho\theta = \rho(l)$$

Figure 5. Statement semantics.

COND builds a pair ⟨meaning of $s1$, meaning of $s2$⟩ which is the kontinuation for the expression E. The value of E causes one of the pair to be selected—one can think of executed.

Note that a compound statement containing labels causes the environment to be updated, as does $\mathscr{E}[\![\textbf{valof } statement]\!]$. $\mathbb{P}[\![\textbf{goto } L]\!]$ drops the given continuation and is the meaning of L in the given environment.

SEMANTICS TO INTERPRETER

Strictly speaking, when a denotational definition is applied to a particular program one calculates the function that the program denotes. One can then apply that function to an initial environment, continuation and state, but the meaning of the program is the function not its application. However, with an interpreter it is the result of application of a program/function that is of interest.

A stack-based language such as Algol 68 does not allow a procedure p to return a procedure q as a result if q depends on local objects of p. As pointed out by Pagan[7] this prevents a direct coding of the semantics. Pascal is even more restrictive in that p cannot return any procedure result at all. Only in a language with very general scope rules can an 'interpreter' P be written to return the function denoted by a statement.

Note that when

$$f: A \to B \to C$$

and

$$g: (A \times B) \to C$$

are related by $(f(a))(b) = g(a, b)$, f is called the curried version of g. The functions in denotational semantics are curried to make the equations simpler—$f(a)$: $B \to C$ is meaningful but $g(a,?)$ is not. $f(a)$ can be considered as g partially parameterized and Pagan's final coding of semantics[7] requires Algol 68 extended in this way.

Because an interpreter is to apply the meaning of a program the approach taken here is to *uncurry* the denotational functions and concentrate on the final state of the program. This enables semantics to be coded in a very ordinary language such as Pascal or Algol 68.

This partly solves the problems of lack of partial parameterization and of the limited scope of procedure results. Remaining instances of partial parameterization can be solved by writing new (small) procedures. This is illustrated in the next section.

As a last resort a data-structure can be introduced to program around partial parameterization or scope problems. In the interpreter presented in the appendix a data-structure is used to represent a state. It is this state returned by P that is the result of the interpreter applied to a contlang program. It is in principle possible to remove this last data-structure as discussed later.

THE INTERPRETER

Brief examples will illustrate the general construction of the interpreter.

The meaning of statements (Fig. 5) is given by

$$\mathbb{P}: cmd \to env \to cont \to state \to state$$

this can be uncurried to

$$\mathbb{P}: (cmd \times env \times cont \times state) \to state$$

or closer to Pascal

function *P*(*cmd*; **function** *env*(. . .) . . .;
function *cont*(. . .) . . .;
state): *state*

P forms the main routine of the interpreter. The body of P has one case for each form of statement, for example $S1$; $S2$. The denotational equation for $S1$; $S2$ is

$$\mathbb{P}[\![S1; S2]\!]\rho\theta = \mathbb{P}[\![S1]\!]\rho\{\mathbb{P}[\![S2]\!]\rho\theta\}$$
$$\rho \in env = lab \to cont$$
$$\theta \in cont = state \to state$$

that is to say the meaning of $S1$; $S2$ given an environment ρ and continuation θ is the meaning of $S1$ with environment ρ and continuation {the meaning of $S2$ with environment ρ and continuation θ}. The state has been cancelled out as it is the same on both sides of the equation.

Only the given environment is used but a new continuation, let us call it '$s2c$' is constructed. This continuation can be programmed as

function *s2c*(*s*: *state*): *state*;
begin *s2c* := *P*(*s2*, *env*, *cont*, *s*) **end**

and the meaning of $s1$; $s2$ is

P(*s1*, *env*, *s2c*, *s*)

Note that *env*, *cont* and *s* are parameters to P and are accessible to $s2c$ which is a local function of P.

The final Pascal is only slightly bigger in having full parameters and types, in accessing the program tree and in Pascal's way of returning a result

$$P := P(cmd\uparrow \cdot left, env, s2c, s).$$

In fact $s2c$ is not in final form. Due to the **resultis** statement for returning a value from a **valof**⟨*statement*⟩ expression, the environment must contain an expression kontinuation

$$\kappa: value \to cont$$

for the **resultis** to invoke. Conventionally

$$env = (label \to cont) \times kont$$

In Pascal it is necessary to turn continuations into expression kontinuations which ignore the value.

We now program

$$cont = kont = (value \times state) \to state$$

The lazy approach from Ref. 10 of having a hidden **resultis**-label in the environment is adopted

$$env = (label \times value \times state) \to state$$

The rest of the semantics are programmed in this way. Even the construction of a new environment for a block is a direct translation of the semantic equations. If a label is to be evaluated in the new environment, we look at the last label in the block; if that does not match the rest of the block is searched. If there is no match at all, the old or outer environment is tried. The complete parser and interpreter are given in the appendix.

COMMENTS

As an example to support the claim that the semantics can easily be changed and run, the changes necessary to implement the BCPL-style **break** and **continue** statements are shown in Fig. 6.

```
Example:
    while  B      do
          (. . . . . .break; . . . . . .; continue; . . . . . .)

Interpreter:
    function ℙ(cmd, env, c, s): astate;
             . . . . . .
             function whileenv(L: labtype; v: value; s: astate): astate;
             begin if L = contlab then
                           whileenv := again(v, s)  { again: copy that
                                                    { in loop
                   else if L = breaklab then
                           whileenv := c(v, s)
                   else    whileenv := env(l, v, s)
             end;
             . . .
    begin {ℙ}
          . . .
          continue: P := env(contlab, ∅, s);
          break:    P := env(breaklab, ∅, s);
          while:    P := E(cmd↑·left, whileenv, loop, s);
          . . .
    end   {ℙ}
```

Figure 6. BCPL-style break & continue.

```
                        state: id → value
Semantics:      ℙ: cmd → env → cont → state → state
Uncurried for
interpreter:    ℙ: (cmd × env × cont × state) → state
Further
uncurried:      ℙ: (cmd × env × cont × state × id) → value

function    startstate(I: idtype): value;
begin  if I = 'output' then startstate := ''
       else startstate := undefined
end;

function ℙ(cmd;
         function env(lab; value; function state(. . .). . .; id): value;
         function cont(value; function state(. . .). . .; id): value;
         function state(id): value;
         id): value;
    function update(vv: value;
                    function ss(id): value;
                    id): value;
             function newstate(id): value
             begin if id = cmd↑·left↑·ident then
                           newstate := vv
                   else newstate := ss(id)
             end;
    begin   {update}
            update := cont(0, newstate, id)
    end;
    . . .
    begin {ℙ}
          . . .
          assign: P := E(cmd↑·right, env, update, state, id);
          . . .
    end   {ℙ}

To parse and run:
    print (ℙ(statement, nilenv, nilcontin, startstate, 'output'))
```

Figure 7. Removal of 'astate' datastructure.

It is possible to remove the state data-structure to more closely model

$$\sigma \in state = id \rightarrow value$$

To do this it is necessary to further uncurry ℙ (Fig. 7), & and so forth. A function representing the final state cannot be returned—scope rules forbid it in Algol 68 and it is out of the question in Pascal—but an identifier can be passed *in* and its final value can be passed *out*. This would imply 'running' a program once to inquire the final value of each identifier! This is only sensible if there is a special component of the state for 'output'—possibly with a rather complex value. In passing, note the similarity between updating the state on assignment and the environment on block entry. On balance it seems reasonable to keep the state data-structure; in this form $P(\langle statement \rangle)$ is reminiscent of the single state transition per major computation of Backus' AST systems.[11]

The interpreter P is an entirely functional program. This is no surprise as denotational semantics is concerned with the functions denoted by programs. (The only variable in P is to get around assigning to fields in a record that **function** update returns a pointer to.)

One irresistibly compares the result with the definition of Lisp.[12] Similarities are due to the common influence of λ-calculus. It is not clear if this is a little more evidence that Lisp is just right or that denotational semantics is just right.

The interpreter bears a similar relationship to the semantics of contlang as does the recursive descent parser to the syntax—maybe it is recursive descent semantics.

The interpreter is not the fastest but this is not an objective. In particular the environment within a block may be reevaluated each time that the block is executed. To prevent this it is necessary to (be able to) store a procedure in the program data-structure or to represent continuations by a data-structure. It is not clear to the author what should be done with such static semantic parts.

The interpreter is not so slow as to be totally hopeless and it could find a place as more than just a definition of contlang. It slows execution by 20 to 100 times over compiled code whereas conventional interpreters are reckoned to lower speed by 5 to 10 times; the state lookup is one easily reprogrammed inefficiency. The lack of side-effects in the interpreter may make it a good candidate for program transformation, or for execution by a data-flow machine.

THE METALANGUAGE

The major drawback in the use of Pascal as a metalanguage is certain weaknesses and inconsistencies in its datatypes and function parameters and results. A datatype must be named before a parameter of that type can be specified but **function** and **procedure** are not datatypes so procedure formal parameters must be specified in full. Algol 68 is more concise here but not essentially more powerful. Pascal's lack of a disjoint union of types except through variant records is sometimes inconvenient.

Pascal seems powerful enough to program the semantics of procedures; the meaning of a procedure is not

dissimilar to that of a label. However to handle variables of type procedure or type label it is necessary to store procedures in the state data-structure, if there is one, and Algol 68 would have the advantage here.

CONCLUSIONS

Pascal, a very ordinary and widespread language, has been used to code and run the continuation-denotational-semantics of a simple programming language without much difficulty. It would be reasonable for a language designer to work with such a definition for experimental purposes and for distribution as a reference. The definition, expressed in Pascal, has been checked by the Pascal compiler in a sense more rigorously than the original denotational semantics. In fact this has already uncovered minor slips in early versions of the semantics of the simple language.

The call *P* (statement, –, –, –) turns any program/statement into a functional program '*P*' applied to some data '*statement*'. Although the interpreter is not efficient—that is not an objective—it might be a good candidate for program transformation or for execution on a data-flow machine.

REFERENCES

1. P. D. Mosses, Compiler generation using denotational semantics, in *Mathematical Foundations of Computer Science 1976*, p. 436. Springer Verlag Lecture Notes in Computer Science **45**, Springer Verlag, Berlin (1976).
2. J. Bodwin, L. Bradley, K. Kanda, D. Litle and U. Pleban, Experience with an experimental compiler generator based on denotational semantics, in *Proceedings 1982 Sigplan Symposium on Compiler Construction*, p. 216.
3. L. Paulson, A semantics-directed compiler generator, *9th Annual Symposium on Principles of Programming Languages 1982*, p. 224.
4. M. R. Raskovsky, Denotational semantics as a specification of code generators, *Proceedings 1982 Sigplan Symposium on Compiler Construction*, p. 230.
5. R. Sethi, Control flow aspects of semantics directed compiling, *Proceedings 1982 Sigplan Symposium on Compiler Construction*, p. 245.
6. F. G. Pagan, On interpreter-oriented definitions of programming languages. *The Computer Journal* **19**(2), 151 (1976).
7. F. G. Pagan, Algol 68 as a metalanguage for denotational semantics. *The Computer Journal* **22**(1), 63 (1979).
8. M. J. C. Gordon, *The Denotational Description of Programming Languages*, Springer Verlag, Berlin (1979).
9. R. Milne and C. Strachey, *A Theory of Programming Language Semantics*, Chapman & Hall, London (1976).
10. C. Strachey and C. P. Wadsworth, *Continuations: a mathematical semantics for handling full jumps*, Oxford University PRG-11.
11. J. Backus, Can programming be liberated from the von Neumann style? a functional style and its algebra of programs, *Communications of the ACM* **21**(8), 613 (1978).
12. J. McCarthy, J. Abrahams, D. Edwards, T. Hart and M. Levin, *Lisp 1.5 Programmer's Manual*, 2nd Edn, MIT Press (1965).

Received September 1982

APPENDIX: CONTLANGUAGE PROGRAM

```
program contlanguage( input, output );
label 99; {error "recovery"}
const   nolabel=-1;
        reslabel=-2;
type    value=integer;
        idtype=char;
        labtype=integer;
        alfa=packed array[1..10] of char;
        nodetype =(id,int,
                ne,eq,le,lt,ge,gt,
                plus,minus,
                times,
                valof,
                g0t0,resultis,assign,iff,wile,skip,semi,block);
        tree=^anode;
        anode=record lab:labtype;
                case t:nodetype of
                id:(ident:idtype);
                int:(v:value);
                skip:();
                g0t0, resultis, valof, block :(son:tree);
                ne,eq,le,lt,ge,gt,
                plus,minus,times,assign,wile,semi:
                   (left,right:tree);
                iff:(s1,s2,s3:tree)
              end;
```

```
        astate=^avar;
        avar=record ident:idtype;
                v:value;
                next:astate
             end;
var     prog:array[1..200]of char;
        ptr:integer;
{-----------------------------------------------------------------}
procedure error(message:alfa);
begin writeln('*** error ', message); goto 99
end;

procedure show( s:astate );
begin   if s<>nil then
        begin   writeln( s^.ident, ' = ', s^.v:3 );
                show( s^.next )
end     end;

function buildleaf(tt:nodetype; ii:idtype; vv:value):tree;
  var p:tree;
begin   new(p); buildleaf:=p;
        with p^ do
        begin   t:=tt;
                if tt=id then ident:=ii else v:=vv
end     end;

function buildnode(tt:nodetype;  ss1,ss2,ss3:tree):tree;
  var p:tree;
begin   new(p); buildnode:=p;
        with p^ do
        begin   t:=tt;
                s1:=ss1; s2:=ss2; s3:=ss3
end     end;

function word(s:alfa):boolean;
  var i,j:integer;
begin   while prog[ptr]=' ' do ptr:=ptr+1;
        j:=ptr; i:=1;
        while (s[i]=prog[j]) and (s[i]<>' ') do
        begin i:=i+1; j:=j+1 end;
        if (s[i]=' ') and ((prog[j]<'a')or(prog[j]>'z'))
                and((prog[j]<'0')or(prog[j]>'9')) then
          begin while prog[j]=' ' do j:=j+1;
                ptr:=j; word:=true
          end
        else word:=false
end;

function matchstring(s:alfa):boolean;
  var i,j:integer;
begin   while prog[ptr]=' ' do ptr:=ptr+1;
        j:=ptr; i:=1;
        while (s[i]=prog[j]) and (s[i]<>' ') do
          begin i:=i+1; j:=j+1 end;
        while prog[j]=' ' do j:=j+1;
        if (s[i]=' ') then
          begin ptr:=j; matchstring:=true end
        else matchstring:=false
end;

{-----------------------------------------------------------------}
                                                {expression syntax}
function statement:tree; forward;
function exp:tree; forward;
```

```
function identifier:tree;
  var ch:idtype;
begin   ch:=prog[ptr];
        if (ch<'a')or(ch>'z') then error('ident-----');
        ptr:=ptr+1;
        identifier:=buildleaf(id, ch, 0)
end;

function nteger:tree;
  var ch:char; i:integer;
begin   ch:=prog[ptr];
        if (ch<'0')or(ch>'9') then error('int-------');
        i:=ord(ch)-ord('0'); ptr:=ptr+1;
        while (prog[ptr]>='0') and (prog[ptr]<='9') do
        begin   i:=i*10+ord(prog[ptr])-ord('0');
                ptr:=ptr+1
        end;
        nteger:=buildleaf(int, '?', i)
end;

function opd:tree;
  var ch:char;
begin   ch:=prog[ptr];
        if matchstring('(          ') then
        begin   opd:=exp;
                if not matchstring(')          ') then error('exp )      ')
        end
        else if word('valof      ') then
                opd:=buildnode(valof, statement, nil, nil)
        else if (ch>='a') and (ch <='z') then
                opd:=identifier
        else opd:=nteger
end;

function unit:tree;
begin   if matchstring('-          ') then
                unit:=buildnode(minus,buildleaf(int,'?',0),unit,nil)
        else unit:=opd
end;

function factor:tree;
  var p:tree;
begin   p:=unit;
        while matchstring('*          ') do
                p:=buildnode(times, p, unit, nil);
        factor:=p
end;

function sexp:tree;
  var p:tree; ch:char;
begin   p:=factor; ch:=prog[ptr];
        while (ch='+') or (ch='-') do
        begin   ptr:=ptr+1;
                if ch='+' then
                  p:=buildnode(plus, p, factor, nil)
                else
                  p:=buildnode(minus, p, factor, nil);
                ch:=prog[ptr]
        end;
        sexp:=p
end;

function exp {:tree    forward-ed};
  var p:tree; tt:nodetype;
```

```
begin   p:=sexp; exp:=p; tt:=int{a sort of null};
        if matchstring('=          ') then tt:=eq
        else if matchstring('<>         ') then tt:=ne
        else if matchstring('<=         ') then tt:=le
        else if matchstring('<          ') then tt:=lt
        else if matchstring('>=         ') then tt:=ge
        else if matchstring('>          ') then tt:=gt;
        if tt<>int then
          exp:=buildnode(tt, p, sexp, nil)
end;

{-------------------------------------------------------------------}
                                                    {statement syntax}
function assignment:tree;
  var p:tree;
begin   p:=identifier;
        if matchstring(':=         ') then
          assignment:=buildnode(assign, p, exp, nil)
        else error(':=         ')
end;

function ifstatement:tree;
  var p1, p2 :tree;
begin   p1:=exp;
        if word('then       ') then
        begin   p2:=statement;
                if word('else       ') then
                  ifstatement:=buildnode(iff, p1, p2, statement)
                else error('else------')
        end
        else error('then------')
end;

function whilestatement:tree;
  var p1:tree;
begin   p1:=exp;
        if word('do         ') then
          whilestatement:=buildnode(wile, p1, statement, nil)
        else error('do--------')
end;

function compoundstatement:tree;
  var p:tree;
begin   p:=statement;
        while matchstring(';          ') do
          p:=buildnode(semi, p, statement, nil);
        if not matchstring(')          ') then error('cpmd stat)');
        compoundstatement := buildnode( block, p, nil, nil )
end;

function gotostatement: tree;
begin gotostatement:=buildnode(g0t0, nteger, nil, nil)
end;

function statement {:tree   forward-ed};
  var p:tree; l:labtype;
begin   while prog[ptr]=' ' do ptr:=ptr+1;
        if (prog[ptr]>='0') and (prog[ptr]<='9') then
        begin p:=nteger; l:=p^.v {Pascal!!!};
                if not matchstring(':          ') then error('label :    ')
        end
        else l:=nolabel;
        if word('if         ') then p:=ifstatement
        else if word('while      ') then p:=whilestatement
        else if word('goto       ') then p:=gotostatement
```

```
        else if word('resultis  ') then p:=buildnode(resultis,exp,nil,nil)
        else if word('skip      ') then p:=buildnode(skip,nil,nil,nil)
        else if matchstring('(          ') then p:=compoundstatement
        else p:=assignment;
        p^.lab:=l;
        statement:=p
end;

{-------------------------------------------------------------------}
                                                          {semantics}
function undefined:value;
begin error('undef val.')
end;

function applystate( s:astate; ii:idtype):value; {(apply)state: id->value}
  var found:boolean;
begin   found:=false;
        while (s<>nil) and not found do
                if s^.ident=ii then
                begin found :=true; applystate:=s^.v end
                else s:=s^.next;
        if s=nil then applystate:=undefined
end;

function nilcontin(v:value; s:astate):astate;
        begin nilcontin:=s end;

function emptyenv(l:labtype; v:value; s:astate):astate;
        begin error('empty env.') end;

function p( cmd:tree;
                function env(ll:labtype; vv:value; ss:astate):astate;
                function   c(vv:value; ss:astate):astate;
                s:astate):astate; forward;


function e(exp:tree; function env(l:labtype;vv:value;ss:astate):astate;
                function k(vv:value;ss:astate):astate;
                s:astate):astate;
        function fail(v:value; s:astate):astate;
        begin error('valof.....')
        end;
        function resenv(thelab:labtype; vv:value; s:astate):astate;
        begin   if thelab=reslabel then resenv:=k(vv,s)
                else                    resenv:=env(thelab, vv, s)
        end;

        function opd2(v1:value; s:astate):astate;
          function opr(v2:value; s:astate):astate;
          begin case exp^.t of
                  plus: opr:=k(v1+v2, s);
                  minus:opr:=k(v1-v2, s);
                  times:opr:=k(v1*v2, s);
                  eq:   if v1=v2 then opr:=k(1, s) else opr:=k(0, s);
                  ne:   if v1=v2 then opr:=k(0, s) else opr:=k(1, s);
                  lt:   if v1<v2 then opr:=k(1, s) else opr:=k(0, s);
                  ge:   if v1<v2 then opr:=k(0, s) else opr:=k(1, s);
                  gt:   if v1>v2 then opr:=k(1, s) else opr:=k(0, s);
                  le:   if v1>v2 then opr:=k(0, s) else opr:=k(1, s)
                end{case}
          end{opr};
        begin{opd2}
          opd2 := e( exp^.right, env, opr, s)
        end{opd2};
```

```
begin {e: exp->env->kontinuation->state->state}
  case exp^.t of
    id:          e:=k(applystate(s,exp^.ident), s);
    int:         e:=k(exp^.v, s);
    valof:       e:=p(exp^.son, resenv, fail{if drop out},  s);
    ne, eq, lt, le, gt, ge, plus, minus, times:
                 e:=e(exp^.left, env, opd2, s)
  end{case}
end;

function p; { proc(cmd, env, cont, state)state      forward-ed }
        function s2c(vv:value; s:astate):astate;
        begin s2c:=p(cmd^.right, env, c, s)
        end;
        function update(vv:value; s:astate):astate;
          var p:astate;
        begin   new(p);
                with p^ do
                begin   ident:=cmd^.left^.ident; v:=vv; next:=s
                end;
                update:=c(0, p)
        end;
        function cond(vv:value; s:astate):astate;
        begin if vv=1 then      cond:=p(cmd^.s2, env, c, s)
                else            cond:=p(cmd^.s3, env, c, s)
        end;
        function loop(vv:value; s:astate):astate;
          function again(vv:value; s:astate):astate;
          begin again:=p(cmd, env, c, s)
          end;
        begin   if vv=0 then    loop:=c(0, s)
                else            loop:=p(cmd^.right, env, again, s)
        end;
        function rescontin(vv:value; s:astate):astate;
        begin rescontin:=env(reslabel, vv, s)
        end;

        function newenv{block}(thelab:labtype; vv:value; s:astate):astate;
          function search(cmd:tree;
                          function c(vv:value; ss:astate):astate;
                          s:astate):astate;
            function cat(vv:value; s:astate):astate;
            begin cat:=p(cmd^.right, newenv, c, s)
            end;
          begin {search}
            if cmd^.t=semi then
              if cmd^.right^.lab=thelab then
                search:=p(cmd^.right, newenv, c, s)
              else search:=search(cmd^.left, cat, s)
            else
              if cmd^.lab=thelab then
                search:=p(cmd, newenv, c, s)
              else search:= {old}env(thelab, vv{resultis}, s)
          end{search};
        begin newenv:=search(cmd^.son, c, s)
        end;

begin { p: cmd->env->continuation->state->state }
    case cmd^.t of
      assign:   p:=e(cmd^.right, env, update, s);
      iff:      p:=e(cmd^.s1,    env, cond,   s);
      wile:     p:=e(cmd^.left,  env, loop,   s);
      skip:     p:=c(0, s);
      semi:     p:=p(cmd^.left,  env, s2c,    s);
      block:    p:=p(cmd^.son,newenv,c,       s);
```

```
      resultis: p:=e(cmd^.son, env, rescontin, s);
      goto:     p:=env(cmd^.son^.v,  0,  s)
    end{case}
end{p};



{-------------------------------------------------------------------}

begin
  ptr:=0;
  while not eof do
  begin ptr:=ptr+1; read(prog[ptr])
  end;
  prog[ptr+1]:='.';
  ptr:=1;

  show( p( statement, emptyenv, nilcontin, nil{empty state}) );

  99:
end.
```