Programming Denotational Semantics II

LLOYD ALLISON
Department of Computer Science, University of Western Australia, Nedlands 6009, Western Australia

The Denotational Semantics of a small programming language is coded into Algol-68 to give an interpreter. The Semantics incorporates many of the notions of Standard Semantics including declarations, declaration continuations, final answers and stores or memory which are used to define block structuring, output and parameterless procedures. This extends work previously reported[1]. The coding in Algol-68 is quite straightforward and the result is a type checked and executable semantics which is as readable as the original semantics to one familiar with Algol-68. No special software is needed other than a compiler for the general purpose language Algol-68.

1. INTRODUCTION

Any given technique of language definition, here Denotational Semantics, can be expressed in various notations. Pagan[2] suggested the use of Algol-68 as a metalanguage in which to write the Denotational Semantics definition of a programming language, but he came to the conclusion that Algol-68 would have to be extended with partial parameterisation to be used for this purpose. The present author[1] showed that quite a good job could be done, at least on the definition of a very small language, by uncurrying the denotational equations and expressing them directly in a modest metalanguage such as Pascal, and hence in Algol-68 as it is. Here the technique is extended to the so called Standard Semantics[3] of a bigger language which includes procedures and block-structuring. The semantics incorporates declarations, declaration continuations, final answers and stores or memory. The fit between these Standard Semantics and Algol-68 is better than that obtained before[1]. The results suggest that the technique is applicable to more than just toy languages.

As before, the semantics coded in Algol-68 constitute an interpreter for the defined language in the same way that coding syntax gives a recursive-descent parser. The interpreter is not efficient but this was not an objective. It is mechanically type-checked and executable. It is claimed that the Algol-68 version of the semantics is as readable to one who knows Algol-68 as the conventional denotational version is to one who knows lambda calculus.

Ideally in some sense, one would directly execute the definition of a programming language to give an efficient interpreter or compiler. Parser-generators[4] have long made this possible for the syntax of programming languages. Nevertheless recursive-descent parsing remains an important technique in practice as it requires no special software nor the learning of any new language above the chosen recursive programming language. The development of (true) semantic compiler-compilers[5,6,7] may make it commonplace to formally define a new language and to derive implementations automatically from the definition. Another approach is to use a functional programming language as the metalanguage, and ML[8] is particularly suitable.

The purpose of this note is to investigate how far it is possible to use a conventional block-structured language, with its benefits of compilation, strong typing, wide availability and readability, in expressing Denotational Semantics. The technique requires no special software above a recursive programming language with a reasonable type system, here Algol-68.

2. BASICS

It is assumed that the reader is familiar with the aims, if not the details, of Denotational Semantics; Gordon[3] provides a readable introduction and Milne and Strachey[9] is a reference work.

This paper discusses the coding of definitions in Algol-68[10], although it applies to any block-structured language with a reasonable type mechanism and strong type checking. In fact an Algol-68S system[11] was used, the important omissions being **union** modes and **flex** and the lack of multiples in structures. Full Algol-68 is used in the text but the final interpreter, which is given in an appendix, is written in Algol-68S. The interpreter has been compiled and run; this is felt to be very important as it is otherwise easy to make errors in the semantic equations, particularly when using lambda calculus rather informally, confident in the knowledge that it will not be subject to mechanical scrutiny.

Denotational Semantics is usually written in a typed lambda calculus making much use of high-order functions. The mathematical foundations will not be treated here at all. Given domains A and B, the domain of functions from A to B is written $A \rightarrow B$, or in Algol-68 **proc(A)B**. The domain $A \rightarrow B \rightarrow C$ or $A \rightarrow (B \rightarrow C)$ can be written as **proc(A)proc(B)C** except as pointed out by Pagan[2] a procedure cannot return as a result a **proc** which depends on a local object because this would violate scope rules. However as is well known, given $f: A \rightarrow B \rightarrow C$, there is an $f': A \times B \rightarrow C$ defined by $f'(a,b) = f(a)(b)$ and called the *uncurried* version of f. Provided that it is the final result in C that is of interest, f′ is as good as f, and this turns out to be the case in the semantic equations given here, so that **proc(A,B)C** can be used.

In addition, the disjoint sum of domains $A + B$ can be written **union(A,B)** and the product $A \times B$ can be written **struct(A,B)** in Algol-68. Certain recursive types such as list = empty + (int × list) would directly translate as

mode list = union(empty,struct(int,list))

This is illegal but it is easily recast as

mode cell = struct(int,list); mode list = ref cell

nil represents the empty list.

```
<program>     ::= <statement>
<statement>   ::= begin <stat list> end |
                  if <exp> then <statement> else <statement> |
                  while <exp> do <statement> |
                  <ident> := <exp> |
                  <ident> |
                  <dec>
<dec>         ::= var <id list> |
                  proc <ident> = <statement>
<id list>     ::= <ident> | <ident>, <id list>
<stat list>   ::= <statement> | <statement>; <stat list>
<exp>         ::= <sexp> <relop> <sexp> | <sexp>
<sexp>        ::= <sexp> <addop> <term> | <term>
<term>        ::= <term> * <opd> | <opd>
<opd>         ::= ( <exp> ) | <ident> | <integer>
<relop>       ::= = | <> | < | <= | > | >=
<addop>       ::= + | -
```

Figure 1. Concrete Syntax

3. SUBJECT LANGUAGE

The concrete syntax of the language to be defined is given in figure 1. It includes **if** and **while** statements and is Algol-like in general flavour. Blocks are delimited by **begin** and **end**. Declarations and statements can be interleaved; the precise semantics of this depends on the Denotational Semantics to be given, but informally the effect of a declaration runs from the declaration to the end of the block.

There is only one data-type, integer, but 1 doubles as true and 0 as false. Because of this and for conciseness, the only aspect of context-sensitive semantics defined here concerns the scope of identifiers.

4. STORES AND ANSWERS

In what follows some of the well known concepts of Standard Semantics (see Chapter 5 of reference 3) are introduced in conventional notation and in Algol-68.

To define block structuring, it is usual to introduce locations and stores. The environment at a point in a program is a mapping of declared variable identifiers to locations, and the store maps locations to values (for procedures see later)

e:env = ident→location

s:store = location→value

or

mode env = proc(ident)location,

store = proc(location)value

In fact some storage may not be bound to any identifier and some bound locations may not be initialised

store = location→value + {undefined value, unbound}

or

mode store = proc(location)union(value,undef,unbound)

In early considerations of semantics it is natural to think of computations as store transformations store →store, **proc(store)store**. However the machine store is not available to a Pascal programmer (say) at the end of a run, only the final output file is. To model this final answers are introduced, answer = empty + value × answer. An answer is a sequence of output values, **mode anscell = struct(value,answer); mode answer = ref anscell.** Computations now map stores to answers store→answer, **proc(store)answer**.

One particular sort of computation is a continuation. A continuation c:cont can be thought of as something else to do. The return address for a procedure is a continuation; it is something else to activate when the procedure has done its own work.

To understand a command one requires an environment to interpret local names, a continuation to do after the command, and a store to act upon before a final answer can be produced. The meaning of commands is given by

cc:cmd→env→cont→store→answer

this can be uncurried to

cc:cmd × env × cont × store → answer

or

proc cc = (tree cmd, **env** e, **cont** c, **store** s)**answer**: ...

The introduction of answers brings a hidden bonus in that the result of cc is automatically a data-structure.

5. PROCEDURES

The language being defined includes parameterless procedures. The environment must map identifiers onto denotable values, here locations and procedures

e:env = ident → location + prok

prok = cont→store→answer

The type prok requires a continuation as the *return address* for each procedure and a store for that procedure to produce a final answer; it can be coded as **proc(cont,store)answer**.

Environments could be coded as

mode env = proc(ident)union(location,proc(cont,store)answer)

except that the scope rules prohibit the procedure result again. The **mode env** cannot be uncurried as it stands because of the **union** in the result but the environment can be split into two

env = (ident→location) × (ident→cont→store→answer)

for variables and for procedures. The second component can now be uncurried. The two bits could be left as

components of a structure but it is more convenient to program them as two separate environments

```
mode env = proc(ident)location;
mode pnv = proc(ident,cont,store)answer
```

The choice, always present, of whether or not to allow a name to denote both a variable and a procedure and to use context to distinguish as in

```
begin proc x=...; var x;... x:=... x;... end
```

in our language is now more obvious. The easy way out of leaving this to context-sensitive syntax or static semantics, which is not treated here, is taken.

If jumps and labels were included in the language, the environment would have to include a mapping for them. If labels were syntactically distinct from identifiers, as in Pascal, the following would do

```
env = as before × (label→cont).
```

Taking the previous approach would give a three component environment for variables, procedures and labels

```
env = (ident→location) × (ident→prok) × (label→cont).
```

Note that in the coding to Algol-68, both **prok** and **cont** are procedure **modes** and can be uncurried. Jumps are treated in reference 1.

If identifiers could denote other simple objects, perhaps constants, there would be no problem in recoding the first component of the environment

```
mode env = proc(ident)union(location, value, ...);
mode pnv = proc(ident,cont,store)answer
```

but if further procedure results were needed, it would be necessary to split it again as above.

6. SEMANTICS

The semantics of the subject language is given in figure 2. The equations are still in curried form but use a notation closer to programming languages than the usual lambda calculus. The most important section is the definition of the functions cc, ee and dd which give the meaning of commands, expressions and declarations. They are defined on a case by case analysis. In the interpreter,procedures cc, ee and dd form the driving routines.

There are three sorts of continuation in the semantics. Command continuations cont=store→answer, or **proc (store)answer**, represent some computation, typically the rest of the program. An expression continuation k:kont=value→store→answer, is very similar except that k must also absorb or use the value that is the result of an expression. For example in evaluating an **if** statement the controlling expression is evaluated with an expression continuation called cond which uses the value to pick one of the arms of the **if** statement to be evaluated. Kont can be uncurried to value × store→answer or **proc(value,store)answer**. A declaration produces a new environment rather than a value so a declaration continuation dc:dcont=env→store→answer, must use that new environment and a store to produce an answer. Dcont can be coded as **proc(env,store)answer**.

As an example, to evaluate the program

```
begin var x; x:=99; output x end
```

the semantics dictate that the declaration is evaluated followed by a declaration continuation which consists of evaluating the two statements in a new environment. The new environment binds the first free location in the store to x. The declaration continuation is now invoked, firstly the assignment is evaluated with a continuation which will evaluate output x. The assignment requires that 99 be evaluated with an expression continuation which updates the location bound to x to the value, 99. Finally output x is evaluated; the expression x is evaluated with an expression continuation to do the output operation. Evaluating x involves asking the environment what location x is bound to and then asking the store what is in that location. The expression continuation then appends the value, 99, to the answer.

The semantic equations were directly coded into Algol-68 to produce an interpreter. By way of illustration, the 'classical' rule for a procedure call (figure 1) is

```
cc '<ident>' e c s = e '<ident>' c s
```

that is to say, given a call on procedure ident, an environment e to interpret names, a continuation c to follow and a store s, execute the prok denoted by ident in e passing it c to return to and store s. Recall that in the interpreter the environment is split into a variable environment e and a procedure environment p; only the latter takes part in this rule which is coded as

```
p(op of cmd, c, s)
```

where op **of** cmd pulls the procedure identifier out of the call.

The classical rule for procedure declaration is

```
dd 'proc <ident> = <statement>' e dc s
 = dc newenv s
  where newenv = (identifier i) union(location,prok):
   (cont c, store s)answer:
   if i='<ident>' then cc '<statement>' newenv c s
   else e i c s
```

that is to say execute the declaration continuation with an updated environment; the new environment maps the ident onto the prok which is the meaning of procedure body. Note that the prok is interpreted in the new environment to make recursion possible. Within the interpreter for the case of a procedure declaration we have dc(e, new pnv, s). The updated (procedure) environment is

```
pnv new pnv = (alfa id, cont ret addr, store s)answer:
  if eq(op of s1 of dec, id) then
    cc(s2 of dec, e, new pnv, ret addr, s)
  else p(id, ret addr, s)
  fi
```

which is the old procedure environment changed only at the declared procedure's name. This is slightly longer than the original in accessing the program data-structure and in having type declarations. The latter should not be looked upon as a draw-back for they enable the compiler to rigorously check the Algol-68 form of the semantic equations. The remaining semantic equations were coded in the same way and with the same ease.

Due to the lack of **union**, the **mode value** could not be coded in the preferred way in the interpreter given here

```
      answer          = empty + (value × answer)
      location        = {1, 2, 3, ... }
 s:   store           = location → {value + undefined value + unbound}
 e:   env             = ident → location + prok
      prok            = cont → store → answer
 c:   cont            = store → answer
 dc:  dcont           = env → store → answer
 k:   kont            = value → store → answer

  cc:   statement → env → cont → store → answer

  ee:   exp → env → kont → store → answer

  dd:   dec → env → dcont → store → answer

    cc 'begin <statement> end' e c s = cc '<statement>' e c s

    cc '<s1>;<s2>;...' e c s  = cc '<s1>' e s2c s
      where s2c = (store s)answer:
        cc '<s2>;...' e c s

    cc '<dec>;<s2>;...' e c s = dd '<dec>' e statpart s
      where statpart = (env e, store s)answer:
        cc '<s2>;...' e c s

    cc 'if <exp> then <s1> else <s2>' e c s
    = ee '<exp>' e cond s
      where cond = (value v, store s)answer:
        if v=1 then cc '<s1>' e c s else cc '<s2>' e c s

    cc 'while <exp> do <statement>' e c s
    = ee '<exp>' e loop s
      where loop = (value v, store s)answer:
        if v=1 then cc '<statement>' e again s
          where again = (store s)answer:
            cc 'while <exp> do <statement>' e c s

    cc '<ident> := <exp>' e c s = ee '<exp>' e update s
    where update = (value v, store s)answer: c news
        where news=(location l)value:
          if l=e(<ident>) then v else s(l)

    cc 'output <exp>' e c s = ee '<exp>' e doio s
    where doio = (value v, store s)answer: (v, c(s))

    cc '<ident>' e c s = e '<ident>' c s

    dd 'var <ident1>,<ident2>,...'l e dc s
    = dd 'var <ident1>' e otherdecs s
      where otherdecs = (env e, store s)answer:
        dd 'var <ident2>,...' e dc s

    dd 'var <ident>' e dc s = dc newenv news
    where newenv=(identifier i) union(location,prok):
      if i='<ident>' then new(s) else e(i)
    and   news =(location l)value:
          if l=new(s) then undefined value else s(l)

    dd 'proc <ident> = <statement>' e dc s
     = dc newenv s
      where newenv=(identifier i) union(location,prok):
        (cont c, store s)answer:
        if i='<ident>' then cc '<statement>' newenv c s
        else e i c s

    ee '( <exp> )'   e k s      = ee '<exp>' e k s

    ee '<ident>'   e k s      = k( s(e'<ident>')), s)

    ee '<integer>'   e k s      = k( value('<integer>'), s)

    ee '<exp1> + <exp2>'   e k s      = ee '<exp1>' e rhs s
    where rhs= (value v1, store s)answer:
      ee '<exp2>' e op s
      where op = (value v2, store s)answer:
          k(v1+v2, s)
    other operators similarly
```

```
new = (store s)location:
  (int l:=0; while s(l) /= unbound do l+:=1 od; l)

to execute a program '<statement>':
cc '<statement>' emptyenv finish emptystore
  where  emptyenv=(identifier i)union(location,prok):undefined id
  and    finish =(store s)answer:empty
  and    emptystore=(location l)value:unbound
```

Figure 2. Semantic Equations.

and a cludge of using two unlikely integer values as unbound and as undefined value was adopted. The lack of **union** can be programmed around by means of a structure but this is rather heavy handed if correct.

The syntax of the language was also coded into a simple recursive descent parser. This builds a tree which the semantic interpreter walks. The interpreter is given in an appendix.

7. DISCUSSION

Algol-68 can be used to code the given semantics easily with only a little distortion where the restricted scope of procedure results makes it necessary to uncurry an equation. Whether the definition of a significant language such as Pascal or Algol-68 would require so much distortion as to make the method not worthwhile is an open question at the moment, but there does not seem to be any great obstacle in the definition of Pascal or a substantial subset of it. Perhaps of more interest, a semantics of Prolog[12] has been written, and run, in this way. The metalanguage requires a type mechanism of the power of Algol-68's to handle the high-order functions of Standard Semantics. Note that although Algol-68 **passes** parameters by-value, **proc** is an ordinary **mode** allowing the effect of by-name parameters.

It is very easy to make type errors in semantic equations such as forgetting one of the parameters for a highly curried function or selecting an inappropriate continuation type. The Algol-68 compiler catches all such errors. More significant errors such as forgetting to apply a continuation are quickly shown up by running the semantics – by interpreting small test programs. Such errors are so easy to make that it seems essential that a formal definition be mechanically checked.

As indicated in the earlier paper[1] it is easy to change the semantics of the language and try example programs.

The aim here was to produce a formal definition of a language that is automatically checked and is executable without regard to efficiency. However note that the interpreter presented is very inefficient in its use of the Algol-68 stack. The heavy use of continuations means that when it finally stops interpreting a program there is an activation record in the stack for every evaluation of every operand, operator and control structure in the program. As pointed out by the referee this is a function of the lack of tail-recursion optimisation, which is a property of implementations of Algol-68 not of its own semantics.

8. CONCLUSION

The Denotational Semantics definition of a small language has been given and coded directly into Algol-68. The definition includes the answers, continuations, stores and locations of Standard Semantics. It defines block-structuring, declarations, output and parameterless procedures. The result suggests that the technique is applicable to more than just toy languages; a semantics of Prolog[12] has been written in this way. The coding is almost mechanical and gives a formal definition that is mechanically checked and executable without the need for any other software tools. For one familiar with Algol-68, the interpreter is as easy to work with as the original definition.

Such interpreters are quick and easy to write and are useful in language experimentation and development. Any recursive programming language with a reasonable type mechanism could be used in this way as the metalanguage. The use of Algol-68 enables the derived definition to be executed on a wide range of machines[13] and to be widely understood.

REFERENCES

1. L. Allison, Programming Denotational Semantics, *Computer Journal* V26 No2 (1983) p164–174.
2. F.G. Pagan, Algol–68 as a metalanguage for Denotational Semantics, *Computer Journal* V22 No1 (Feb 1979) p63–66.
3. M.J.C. Gordon, *The Denotational Description of Programming Languages* Springer Verlag 1979.
4. R.A. Brooker, D. Morris, A General Translation Program for Phrase Structure Languages, *JACM* V9 No1 (Jan 1962) p1–10.
5. L.Paulson A Semantics Directed Compiler Generator *9th Annual Symposium on Principles of Programming Languages* (Jan 1982) p224–233.
6. M.R. Raskovsky, Denotational Semantics as a Specification of Code Generators, *Proc*' 1982 *Sigplan Conference on Compiler Construction* (June 1982) p230–244.
7. R. Sethi, Control Flow aspects of Semantics Directed Compiling, *Proc*' 1982 *Sigplan Conference on Compiler Construction* (June 1982) p245–260.
8. M.J. Gordon, A.J. Milner, C.P. Wadsworth, *Edinburgh LCF* Springer Verlag, Lecture Notes in C.S. V78 1979.
9. R. Milne, C. Strachey, *A Theory of Programming Language Semantics*, Chapman Hall 1976.
10. C.H. Lindsey, S.G. van der Meulen, *Informal Introduction to Algol*–68, North Holland (revised) 1977.
11. C.H. Lindsey, Algol–68S system, Dept.Computer Science, University of Manchester.
12. W.F. Clocksin, C.S. Mellish, *Programming in Prolog*, Springer–Verlag 1981.
13. Survey of Viable Algol–68 Implementations, *Algol Bulletin* no 47, (Aug 1981), p15.

APPENDIX.

```
( # Semantic Interpreter, Dept Computer Science U.W.A. 1983 #
 mode alfa = [1:10] char;
 proc eq = (alfa x,y)bool:
   (loc bool b:=true; loc int i := lwb x;
   while b and i <= upb x do
     b:=x[i]=y[i]; i+:=1
   od;
   b
  );
 mode node = struct( ref node s1, s2, s3, ref alfa op, int i);
 mode tree = ref node;
 # error and input routine omitted #
#-------------------------------------------------------------#
                                                  # syntax #
proc program = tree:
begin
   # parse a program, body omitted #
end # program #;
#-------------------------------------------------------------#
                                                  # semantics #
 mode value = int, location = int;
 mode anscell = struct(value v, ref anscell next);
 mode answer = ref anscell;
 mode store = proc(location)value,
      env = proc(alfa)location;
 mode cont = proc(store)answer,
      kont = proc(value,store)answer;
 mode pnv = proc(alfa,cont,store)answer;
 mode dcont = proc(env,pnv,store)answer;

 value unbound = -max int, undefined value = -(max int -1);
 proc dump = (answer s)void:
  if s isnt nil then
     print((newline, v of s)); dump(next of s)
  fi;
 proc new = (store s)location:
 ( loc location l := 1;
  while s(l) /= unbound do
    l +:= 1
  od;
  l
 );

 # cc: cmd→(env × pnv)→cont→store→answer #
 proc cc = (tree cmd, env e, pnv p, cont c, store s)answer:
 begin

 # dd:dec→(env × pnv)→dcont→store→answer #
 proc dd = (tree dec, env e, pnv p, dcont dc, store s) answer:
 begin
  env new env = (alfa id) location:
     if eq(op of dec, id) then new(s)
     else e(id)
     fi;
  pnv new pnv = (alfa id, cont ret addr, store s)answer:
     if eq(id, op of s1 of dec) then
        cc( s2 of dec, e, new pnv  # recursion! #, ret addr, s)
     else p(id, ret addr, s)
     fi;
  dcont other decs = (env e, pnv p, store s)answer:
     dd(s2 of dec, e, p, dc, s);
  if dec is nil then dc(e, p, s)
  elif eq(op of dec, 'var      ') then
     dd(s1 of dec, e, p, dc, s)
  elif eq(op of dec, 'proc     ') then
     dc(e, new pnv, s)
  elif eq(op of dec, ',        ') then # dec1, dec2, ... #
     dd(s1 of dec, e, p, other decs, s)
  else # var id #
     store new s = (location l)value:
        if l=new(s) then undefined value
        else s(l)
        fi;
     dc( new env, p, new s )
  fi
 end # dd #;
```

```
# ee : exp→(env × pnv)→kont→store→answer #
proc ee = (tree exp, env e, pnv p, kont k, store s)answer:
begin
  kont rhs = (value v1, store s)answer:
   ( kont operator = (value v2, store s)answer:
       k((alfa opr = op of exp;
          if eq(opr,'=          ') then
             if v1=v2 then 1 else 0 fi
          elif eq(opr,'< >          ') then
             if v1/=v2 then 1 else 0 fi
          elif eq(opr,'<          ') then
             if v1<v2 then 1 else 0 fi
          elif eq(opr,'<=          ') then
             if v1<=v2 then 1 else 0 fi
          elif eq(opr,'>          ') then
             if v1>v2 then 1 else 0 fi
          elif eq(opr,'>=          ') then
             if v1>=v2 then 1 else 0 fi
          elif eq(opr,'+          ') then v1+v2
          elif eq(opr,'-          ') then v1-v2
          elif eq(opr,'*          ') then v1*v2
          else error(' undef operator in ee'); skip
          fi),
          s
         );
     ee(s2 of exp, e, p, operator, s)
   ) # rhs # ;

  if eq(op of exp, '-integer ') then
     k(i of exp, s)
  elif (op of exp)[1]>='a' and (op of exp)[1]<='z' then
     value v = s( e( op of exp ) );
     if v=undefined value then error(' undefined variable');skip
     else k( v, s)
     fi
  else
     ee(s1 of exp, e, p, rhs, s)
  fi
end # of ee # ;

# the body of cc(cmd,env,pnv,cont,store)answer #
  dcont stat part = (env e, pnv p, store s)answer:
     cc(s2 of cmd, e, p, c, s);
  kont cond = (value v, store s)answer:
     cc(if v=1 then s2 of cmd else s3 of cmd fi, e, p, c, s);
  cont again = (store s)answer:cc(cmd,e,p,c,s);
  kont loop = (value v, store s)answer:
     if v=1 then cc(s2 of cmd, e, p, again, s) else c(s) fi;
  cont s2c = (store s)answer: cc(s2 of cmd, e, p, c, s);
  kont update = (value v, store s)answer:
     c( (location l)value:
           if l = e(op of s1 of cmd) then v else s(l) fi
      );
  kont do i o = (value v, store s)answer:
     heap anscell := (v, c(s));

  if cmd is nil then c(s)
  elif eq(op of cmd, 'begin     ') then
     cc(s1 of cmd, e, p, c, s)
  elif eq(op of cmd, ';          ') then
     if eq(op of s1 of cmd, 'var          ')
        or eq(op of s1 of cmd, 'proc       ') then # dec; stats #
        dd(s1 of cmd, e, p, stat part, s)
     else # stat; statlist #
        cc(s1 of cmd, e, p, s2c, s)
     fi
  elif eq(op of cmd, 'if          ') then
     ee(s1 of cmd, e, p, cond, s)
  elif eq(op of cmd, 'while     ') then
     ee(s1 of cmd, e, p, loop, s)
  elif eq(op of cmd, ':=          ') then
     ee(s2 of cmd, e, p, update, s)
  elif eq(op of cmd, 'output    ') then
     ee(s1 of cmd, e, p, do i o, s)
  else # identifier : call on a proc #
     p(op of cmd, c, s)
  fi
end # of cc # ;
#-------------------------------------------------------#
  dump( cc( show( program),
          (alfa id)location:(error(' undeclared id'); skip),
          (alfa id, cont ra, store s)answer:
             (error(' undeclared proc'); skip),
          (store s)answer:nil,
          (location l)value:unbound    ))
)
```