CHAPTER 4

MODELLING CONCEPTS OF PROGRAMMING LANGUAGES

The design of the meta-language (known as "META-IV") used in VDM is motivated here by considering programming language concepts which are to be formally defined. In particular, the combinators which are introduced in chapter 2 are shown here to provide readable definitions of language concepts like block structure and goto statements. Although programming language definition was the first use of META-IV, part III shows that most of the meta-language applies to the specification of other systems. The individual language concepts are collected together to form a mini-language whose complete definition is given in section 4.10. Alternative ways of handling exception constructs are discussed in the next chapter.

This chapter is a rewritten version of [Jones 78c]. Similar motivations for denotational semantic definition techniques are given in [Tennent 80a] and [Gordon 79a].

CONTENTS

4.1 INTRODUCTION

This chapter introduces many of the basic concepts of programming languages and, for each such concept, shows how it can be defined using denotational semantics. The emphasis is on the underlying model required although the metalanguage used to present these models is that used throughout this book. The concepts have been selected so as to motivate features of the metalanguage. For this reason, it might appear that each new language feature requires some extension of the metalanguage. In the definition of a full language (e.g. chapter 6) the features of the metalanguage are used far more intensively.

4.2 EXPRESSIONS

The denotational method defines semantics by mapping the objects to be defined to some known objects. The objects which are the target of this mapping are referred to as "denotations" and are assumed to be already understood. The essence of this chapter is to show the different denotations used to define various concepts of programming languages. A language whose constructs are expressions requires only simple denotations and will provide a useful introduction to the denotational method.

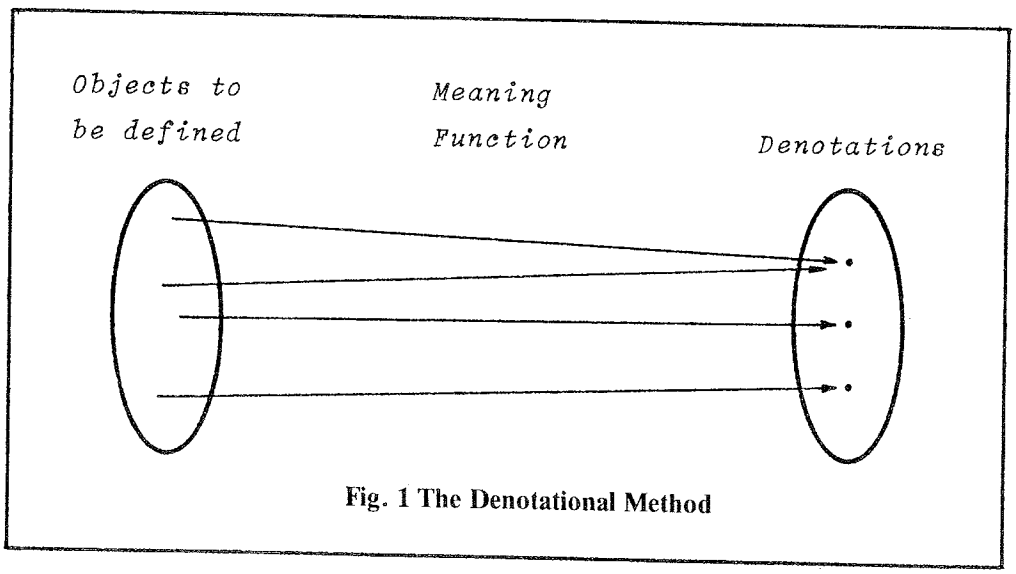

Fig. 1 The Denotational Method

Fig. 1 illustrates the denotational approach to defining semantics. The first task is to fix the class of objects whose semantics are to be defined. For a small finite language, it is possible to do this by enumeration.

Fig. 2 enumerates all of the Boolean expressions which can be constructed with an implication sign and the Boolean constants. The next part of the general method requires choosing a set of objects to use as denotations. For the language of Fig. 2 this can be the Boolean values and the mapping of expressions to denotations is given in the same figure by enumeration.

Language	Denotation
false ⊃ *false*	*true*
false ⊃ *true*	*true*
true ⊃ *false*	*false*
true ⊃ *true*	*true*

Fig. 2 Semantics of Implication with Constants

For infinite, or even very large, languages it is not possible to define the class of objects of interest by enumeration. The strings of such a language are normally defined by syntax rules and the structure created by such a syntax is used in fixing the semantics. Fig. 3 gives an *abstract syntax* of an (infinite) language of logical expressions. The notation used is that for describing objects discussed in chapter 2. The valid expressions are any finite objects matching this recursive definition.

```
Boolexpr      =  Boolinfixexpr | Negation | Boolconst
Boolinfixexpr :: Boolexpr  Boolop  Boolexpr
Negation      :: Boolexpr
Boolconst     =  true | false
Boolop        =  AND|OR|IMPL|EQUIV
```

Fig. 3 Abstract Syntax of Logical Expressions

What denotations are to be used for the logical expression language? The same set of Boolean values will suffice:

$$Bool = \{true, false\}$$

The mapping from the language to the denotations can be given by a function of type:

M: Boolexpr → Bool

There are different ways of presenting such functions. Because the language involved is small, it will be clearest to provide a definition for each case separately. Thus:

```
M[mk-Boolinfixexpr(e1,op,e2)] Δ
  let v1 = M[e1] in
  let v2 = M[e2] in
  cases op:
    AND   → if v1 then v2 else false,
    OR    → if v1 then true else v2,
    IMPL  → if v1 then v2 else true,
    EQUIV → if v1 then v2 else (if v2 then false else true)

M[mk-Negation(e)] Δ let v = M[e] in if v then false else true
```

The mapping for the constants is the identity. This actually shows that the objects contained in the expressions of Fig.3 are, in part, semantic objects. This could easily be avoided by choosing some syntactic representation for the semantic objects.

Looking at the definition of *M* it is clear that the semantics are built up over the structure of the syntax. This corresponds to a rule of the denotational method that the semantics of a compound construct should be derived (only) from the semantics of its components. It is perhaps this rule more than anything else which distinguishes between denotational and operational semantics. The point could be further emphasized by redefining:

```
M[mk-Boolinfixexpr(e1,op,e2)] Δ M[op](M[e1],M[e2])

M[AND] Δ λv1,v2.(if v1 then v2 else false)
```

etc. Notice that, although *M* is defined recursively, there is no need to introduce domains (cf. chapter 3): since no fixed-points are taken; a set of defined values is adequate to define the denotations.

Fig.3 contains an *abstract syntax*. This term is used to distinguish it from the normal form of (concrete) syntax which is concerned with the strings of symbols of the language. The advantage of an abstract syntax is that it can provide a smaller class of objects which are more clearly structured as a basis for the semantic definition. Objects corresponding to an abstract syntax can often be represented in many ways by character strings. Although this becomes more apparent in languages with richer sets of abbreviations and defaults, the points can already be seen in the language of logical expressions. A Backus-Naur Form (BNF) definition would have to show the possibility of bracketing subexpressions as a way of defining priority of operators. The abstract syntax also makes it possible to avoid issues of representation; the BNF definition would have to choose a specific way of showing a negation. The correspondence between (concrete) strings and (abstract) trees is also part of the definition of the language. Although this book tends to focus on semantics, it is clear that both BNF and the concrete/abstract relationship are part of a full definition of a system.

Expressions in programs are not usually restricted to constants and operators. It is normal to have some form of reference to variables. Thus the definition of *Boolexpr* might be extended with:

Boolexpr = ... | *Varref*
Varref :: *Id*

It is clear that an expression of this extended language does not denote a simple Boolean value. What is denoted will depend, in general, on the values of the various variables. A first step in choosing the denotations is to recognize the concept of a store which associates identifiers with values. Using a mapping, this can be written as:

$STORE = Id \xrightarrow{m} Bool$

This leads naturally to the idea of a denotation which is a function from stores to values. Thus:

$M: Boolexpr \to (STORE \to Bool)$

$M[mk\text{-}Boolinfixexpr(e1,op,e2)](\sigma) \triangleq$
  let $v1 = M[e1](\sigma)$ *in*
  let $v2 = M[e2](\sigma)$ *in* $M[op](v1,v2)$

$M[mk\text{-}Negation(e)](\sigma) \triangleq \underline{let}\ v = M[e](\sigma)\ \underline{in}\ \underline{if}\ v\ \underline{then}\ \underline{false}\ \underline{else}\ \underline{true}$

$M[mk\text{-}Varref(id)](\sigma) \triangleq \sigma(id)$

(The problem of ensuring that the stores have values for all necessary variables is postponed until more realistic languages are considered.) Writing the store parameter everywhere can become tedious and it is necessary below to find a way of avoiding having to do so.

4.3 STORE CHANGES

The dependence of the value of an expression on the values of variables has been defined by using functions as denotations. The next step is to consider language constructs which change the store and here again functional denotations will be required. The assignment statement is the obvious way of changing the store. (In fact, the read/write nature of the von Neumann computer architecture has influenced much more than just programming languages. Operating systems and data management systems are built around the notion of a changeable store. Only the enthusiasts of a functional programming style resist this notion (e.g. [Burge 75a, Backus 78a, Henderson 80a]).)

The abstract syntax of assignment statements is defined by:

$Assign :: Id\ \ Expr$

The denotation of such statements is to be a store to store function:

$M: Assign \rightarrow (STORE \rightarrow STORE)$

$M[mk\text{-}Assign(id,e)](\sigma) \triangleq \underline{let}\ v = M[e](\sigma)\ \underline{in}\ \sigma + [id \mapsto v]$

It is interesting to notice how the different uses of identifiers are brought out by the definition. In the source language assignment:

```
x := x + 1
```

the occurrence of 'x' on the right hand side of the assignment denotes the value of the variable whereas the occurrence on the left denotes the "address" which has to be modified. This is seen by comparing the uses

of the *id* in the semantic function for *Assign* and in that given above for *Varref*. The introduction of *locations* below will provide a mechanism for making the distinction between left and right values of variables more explicit.

The use of functional denotations is not only a natural result of the denotational method. It has the further advantage that the functions are familiar objects with known methods of manipulation. The task of defining the meaning of a sequence of assignments can be performed by using functional composition of the denotations of the individual assignments. Thus:

$$Ml: Assign^* \rightarrow (STORE \rightarrow STORE)$$

$$Ml[\langle\rangle] \triangleq I_{STORE}$$
$$Ml[al] \triangleq Ml[\underline{tl}\,al] \circ M[\underline{hd}\,al]$$

Notice that the denotation of the empty list of statements is the identity function on stores. It would have been equivalent to write:

$$Ml[\langle\rangle](\sigma) = \sigma \qquad \text{or:} \qquad Ml[\langle\rangle] = \lambda\sigma.\sigma$$

The denotation for a non-empty list is given by composition: composing two functions of type Store to Store gives a denotation of the correct type. Composition is a "combinator" which makes it possible to build up expressions without having to write all of the arguments. A tasteful choice of combinators will do much to aid the readability of definitions. Expanding the meaning of the functional composition used above, gives:

$$Ml[al](\sigma) = Ml[\underline{tl}\,al](M[\underline{hd}\,al](\sigma))$$

The need to do one thing followed by another is familiar from programming. If a (semicolon) combinator is defined:

$$f1;f2 = \lambda\sigma.(f2(f1(\sigma)))$$

then the definition can be restated as:

$$Ml[al] \triangleq M[\underline{hd}\,al];Ml[\underline{tl}\,al]$$

This is more natural for a programmer to read. There is no danger of

circularity in the use of a combinator which might be the same as the (syntactic) symbol used in the language to be defined providing the combinators are themselves defined formally in terms of mathematical concepts.

It is also quite safe to define a *for* combinator which provides the obvious (static) expansion into a sequence of semicolon compositions. The definition of *Ml* can then be rewritten:

$$Ml[al] \triangleq \underline{for}\ i=1\ \underline{to}\ lenal\ \underline{do}\ M[al[i]]$$

Thus given:

$$M[x:=x+1] = \lambda\sigma.(\sigma + [x \mapsto \sigma(x)+1])$$
$$M[x:=x-2] = \lambda\sigma.(\sigma + [x \mapsto \sigma(x)-2])$$

then:

$$\begin{aligned}
&Ml[x:=x+1;x:=x-2;x:=x+1] \\
&= M[x:=x+1];M[x:=x-2];M[x:=x+1] \\
&= \lambda\sigma.(M[x:=x+1](\lambda\sigma'.M[x:=x-2](M[x:=x+1](\sigma'))))(\sigma) \\
&= \lambda\sigma.M[x:=x+1](M[x:=x-2](M[x:=x+1](\sigma))) \\
&= \lambda\sigma.M[x:=x+1](M[x:=x-2](\sigma + [x \mapsto \sigma(x)+1])) \\
&= \lambda\sigma.M[x:=x+1](\sigma + [x \mapsto \sigma(x)+1-2]) \\
&= \lambda\sigma.\sigma + [x \mapsto \sigma(x)-1+1] \\
&= \lambda\sigma.\sigma \\
&= I_{STORE}
\end{aligned}$$

The need for another combinator becomes apparent if the expression language is extended to permit function invocation. It should be clear that the possibility of side effects in such a language complicates the denotations of expressions. In general the evaluation of an expression yields both a changed store and a value. Thus, for such a language:

$$M: Expr \to (STORE \to STORE \times Bool)$$

Without combinators, it would be necessary to write:

$$\begin{aligned}
&M[mk\text{-}Assign(id,e)](\sigma) \triangleq \\
&\quad \underline{let}\ (\sigma', v) = M[e](\sigma)\ \underline{in} \\
&\quad \sigma' + [id \mapsto v]
\end{aligned}$$

$$M[mk\text{-}Boolinfixexpr(e1,op,e2)](\sigma) \triangleq$$
$$\quad \underline{let}\ (\sigma',v1) = M[e1](\sigma)\ \ \underline{in}$$
$$\quad \underline{let}\ (\sigma'',v2) = M[e2](\sigma')\ \underline{in}\ (\sigma'',M[op](v1,v2))$$

A *def* combinator can, however, be defined:

$$(\underline{def}\ v: f1;\ f2(\ldots v\ldots)) = \lambda\sigma.(\underline{let}\ (\sigma',v) = f1(\sigma)\ \underline{in}\ f2(\ldots v\ldots)(\sigma'))$$

Using this combinator the definitions can be written:

$$M[mk\text{-}Assign(id,e)] \triangleq \underline{def}\ v: M[e];\ assign(id,v)$$

$$assign(id,v)(\sigma) \triangleq \sigma + [id \mapsto v]$$

$$M[mk\text{-}Boolinfixexpr(e1,op,e2)] \triangleq$$
$$\quad \underline{def}\ v1: M[e1];$$
$$\quad \underline{def}\ v2: M[e2];$$
$$\quad \underline{return}(M[op](v1,v2))$$

The *return* combinator elevates a pure value to an object of appropriate type:

$$\underline{return}(v) = \lambda\sigma.(\sigma,v)$$

It should be noted in passing that the definitions given for expressions define the order of evaluation of sub-expressions.

4.4 COMPOSITE STATEMENTS

The sequential composition discussed above is the simplest of the techniques provided in most programming languages for building composite statements. Only slightly more complicated is the conditional statement. The abstract syntax might be written:

$$If :: Expr\ Stmt\ Stmt$$

The semantics which are to be defined are firstly to evaluate the expression to a Boolean value and then to use the function corresponding to the denotation of the appropriate statement. Given a value $v \in BOOL$ a combinator can be defined:

$$\underline{if}\ v\ \underline{then}\ f1\ \underline{else}\ f2 = \lambda\sigma.(\underline{if}\ v\ \underline{then}\ f1(\sigma)\ \underline{else}\ f2(\sigma))$$

This combinator combines two functions from stores to stores to yield a function of the same type. The semantics of the conditional construct in the language can then be defined:

$$M[mk\text{-}If(e,th,el)] \triangleq \underline{def}\ v: M[e];\ \underline{if}\ v\ \underline{then}\ M[th]\ \underline{else}\ M[el]$$

(The reader should try expanding the combinator definitions and check that the resulting semantics of the conditional is a function from stores to stores.) Once again a combinator *(if)* has been used which is similar to a construct of the language to be defined. The use with a value rather than an expression was simpler than the feature being defined and the same combinator is now used to define a more involved construct.

One iterative construct found in programming languages is the "while statement". Its syntax is:

$$While :: Expr\ Stmt$$

Its semantics can be given by:

$$M[mk\text{-}While(e,s)] \triangleq \underline{let}\ L = (\underline{def}\ v: M[e];\ \underline{if}\ v\ \underline{then}\ (M[s];L)\ \underline{else}\ I_{STORE})\ \underline{in}\ L$$

The definition of L is recursive. Assuming L to be of the appropriate type one might write:

$M[e]$:	$STORE \to STORE \times Bool$
$M[s]$:	$STORE \to STORE$
$(M[s];L)$:	$STORE \to STORE$
I_{STORE}	$STORE \to STORE$
$(\underline{def}\ v: M[e];\ \underline{if}\ v\ \underline{then}\ (M[s];L)\ \underline{else}\ I_{STORE})$:	$STORE \to STORE$

The recursion in the definition of L is exactly the sort which has been explained in chapter 3: it is here that the denotations must be domains. The definition of L denotes the least fixed point of the recursive equation. There is, however, a factor to be considered which has not occurred above. Such functions will only be partial. To see this it is only necessary to ask what denotation to associate with a loop which (for some

starting stores) fails to terminate. Thus the type of the semantic function for the iterative construct is:

$$M: While \rightarrow (STORE \overset{\sim}{\rightarrow} STORE)$$

The syntactic definitions given above permit arbitrary nesting of the composite statements. Thus the syntax rule for statement is:

$$Stmt = If \mid While \mid Assign$$

A consequence of this is that the partial nature of the denotations inherit, so that:

$$M: Stmt \rightarrow (STORE \overset{\sim}{\rightarrow} STORE)$$

4.5 SCOPE

Block-structured programming languages employ the notion of the scope of a variable. Many other systems have ways of binding different values to names at different times so the way of handling scope is of general interest. Furthermore, the need to restrict the class of objects to be defined, which is handled here by "context conditions", occurs in nearly all definitions.

Suppose that the programs of some language fit the following abstract syntax:

$$\begin{array}{lll}
Program & :: & Stmt \\
Stmt & = & Block \mid Call \mid Assign \\
Block & :: & s\text{-}vars{:}Id\text{-}set \;\; s\text{-}procm{:}(Id \overset{m}{\rightarrow} Proc) \;\; s\text{-}body{:}Stmt^* \\
Proc & :: & s\text{-}parml{:}Id^* \;\; s\text{-}body{:}Stmt \\
Call & :: & s\text{-}proc{:}Id \;\; s\text{-}argl{:}Varref^* \\
Assign & :: & Varref \;\; Boolexpr \\
Varref & :: & Id \\
Boolexpr & = & Boolinfixexpr \mid Rhsref \mid Boolconst \\
Rhsref & :: & Varref
\end{array}$$

The language still has only one type of variable (say Boolean), but variables are now either declared in the *s-vars* part of a block or are names of parameters within procedures. The statements which comprise the bodies

of blocks or procedures should use only identifiers which have been declared. It is well-known that context-free syntax rules cannot capture such constraints. The abstract syntax definition is essentially context-free and some way of defining which programs are valid is required. One possibility is to insert suitable tests into the definition of the semantic functions. Since this would lengthen that part of a definition which is anyway long, this proposal is rejected. Instead the class of objects of interest will be restricted as a separate task. Rather than adopt one of the exotic syntax schemes like two-level grammars (cf.[van Wijngaarden 75a]), the restriction is defined here by a predicate *(WF)*. Those objects which satisfy the "context conditions" are said to be "well-formed" or "valid".

The intent, then, is to define a predicate:

$$WF: Program \to Bool$$

This is defined using a (recursive) sub-function:

$$WF: Stmt \to Staticenv \to Bool$$

The static environment is a mapping which contains information about declared names:

$$\begin{aligned} Staticenv &= Id \xrightarrow{m} Attribute \\ Attribute &= \underline{BOOL} \mid Proctype \\ Proctype &:: Nat0 \end{aligned}$$

With only one variable type, the context conditions are easy to define. In particular, the only information required about a procedure is the number of its arguments.

$$WF[mk\text{-}Program(s)] \triangleq WF[s]([])$$

$$\begin{aligned} &WF[mk\text{-}Block(vars,procm,sl)](\rho) \triangleq \\ &\quad vars \cap \underline{dom}procm = \{\} \ \wedge \\ &\quad (\forall p \in \underline{rng}procm)(WF[p](\rho)) \ \wedge \\ &\quad (\underline{let}\ \rho' = \rho + ([id \mapsto \underline{BOOL} \mid id \in vars] \cup \\ &\qquad\qquad [id \mapsto ATTR[procm(id)] \mid id \in \underline{dom}procm])\ \underline{in} \\ &\quad (\forall s \in \underline{elems}sl)(WF[s](\rho'))) \end{aligned}$$

$$WF[mk\text{-}Proc(pl,b)](\rho) \triangleq (\forall i,j \in \underline{inds}\, pl)(pl[i]=pl[j] \supset i=j) \wedge (WF[b](\rho+[pl[i] \mapsto \underline{BOOL} \mid i \in \underline{inds}\, pl])$$

$$ATTR[mk\text{-}Proc(pl,b)] \triangleq mk\text{-}Proctype(\underline{len}\, pl)$$

$$WF[mk\text{-}Call(pid,al)](\rho) \triangleq pid \in \underline{dom}\,\rho \wedge \rho(pid) \in Proctype \wedge (\underline{let}\ mk\text{-}Proctype(n) = \rho(pid)\ \underline{in}\ \underline{len}\, al = n) \wedge (\forall i \in \underline{inds}\, pl)(\rho(pl[i]) : \underline{Bool})$$

(These context conditions prohibit recursion - this restriction is removed below.)

In practice, some of the rules can be mechanically generated and need not be written out. For example, there is no need to write that the wellformedness of the assignment statement depends (with the same environment) on that of the variable reference and expression. This leaves, then, only:

$$WF[mk\text{-}Varref(id)](\rho) \triangleq id \in \underline{dom}\,\rho \wedge \rho(id) = \underline{BOOL}$$

It is now necessary to turn to the question of providing semantics for the well-formed programs of the language. So far the store has directly associated values with identifiers. But now it is possible to have programs like:

```
begin Boolean a,b,c;
      begin Boolean a; ... end;
      ...
      begin Boolean b; ... end
end
```

The scope rules ensure that not only do the two declarations of 'a' introduce different variables, but also that execution of the first inner block cannot affect the value of the outer variable named 'a'. In this case it would be possible to define the semantics in terms of the simple store by systematically changing identifiers in the program so as to avoid name clashes. This is, however, a patch and, as so often with patches, will not work in the general case. Consider the following program fragment in which parameters are assumed to be passed by reference (or location):

```
begin Boolean a;
      procedure p(x,y); x:=a+y;
         ... ;p(a,a); ...
end
```

Within the procedure 'p', all of the identifiers 'a', 'x', 'y' refer to the same entity. The necessary model introduces an abstraction of the machine address which the implementation would associate with the entity. This abstraction is called here a (scalar) location. The problem of having different uses of the same identifier can now be solved by associating different locations with the identifier at different times. Sharing can be defined by associating different identifiers with the same location as in fig. 4. The association between identifiers and locations is recorded in an environment and *STORE* now associates values with locations:

$$ENV = Id \xrightarrow{m} SCALARLOC$$
$$STORE = SCALARLOC \xrightarrow{m} Bool$$

It is now necessary to decide how the environment is to be handled in semantics. Placing it in the store causes a number of problems and it serves the purposes of the definition far better to treat the environment as a parameter used in determining the (store to store) denotation. Thus:

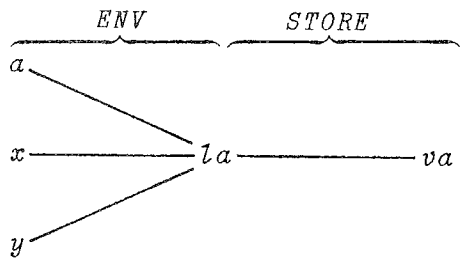

Fig. 4: The Use of Locations

$$M : Stmt \to ENV \to STORE \xrightarrow{\sim} STORE$$
$$ENV = Id \xrightarrow{m} DEN$$
$$DEN = SCALARLOC \mid \ldots$$
$$STORE = SCALARLOC \xrightarrow{m} [Bool]$$

The semantic functions can now distinguish clearly between the places where a left hand denotation (i.e. location) and a right hand denotation

(i.e. value) are required.

$$M[mk\text{-}Assign(vr,e)](\rho) \triangleq \underline{def}\ l: Mloc[vr](\rho);\ \underline{def}\ v: M[e](\rho);\ assign(l,v)$$

$$Mloc: Varref \to ENV \to SCALARLOC$$
$$Mloc[mk\text{-}Varref(id)](\rho) \triangleq \rho(id)$$

$$assign(l,v) \triangleq \lambda\sigma.\sigma + [l \mapsto v]$$

The parts of the expression semantics all require the extra environment argument. The interesting case is:

$$M[mk\text{-}Rhsref(vr)](\rho) \triangleq \underline{def}\ l: Mloc[vr](\rho);\ contents(l)$$

$$contents(l) \triangleq \lambda\sigma.\sigma(l)$$

New locations must be associated with all declared identifiers in a block. Locations are considered to be new if they are not in the domain of the store. Given this technique it is necessary for the *newloc* function to reserve an identifier even before it is assigned a value. This is achieved by associating the identifier with the *nil* object. Subsequent development of an implementation (cf. chapter 8) has to show that some particular way of choosing locations satisfies the specification being constructed here. For this reason the constraints on, among other things, locations should be minimized. The set *SCALARLOC* is an arbitrary infinite set of objects. The function *newloc* chooses an arbitrary free location. In order to show that a stack implementation is possible (but not to prescribe it) all locations corresponding to local variables should be deleted from the store after the block semantics have been determined. Thus:

$$M[mk\text{-}Block(vars,procm,sl)](\rho) \triangleq$$
$$\underline{def}\ \rho' : \rho + ([id \mapsto newloc() \mid id \in vars] \cup [id \mapsto \ldots \mid id \in \underline{dom} procm]);$$
$$\underline{for}\ i=1\ \underline{to}\ lensl\ \underline{do}\ M[sl[i]](\rho');$$
$$epilogue(\underline{rng}(\rho' | vars))$$

$$newloc()(\sigma)=(\sigma',l) \supset \neg(l \in \underline{dom}\sigma) \wedge \sigma'=\sigma \cup [l \mapsto \underline{nil}]$$

$epilogue(ls) = \lambda\sigma.\sigma\backslash ls$

The environment is a map (i.e. a restricted function). Chapter 3 shows that passing such maps as arguments to semantic functions is quite safe. This provides the basis of the definition method. It may, however, be useful to give a less mathematical view of environments before turning to the subject of procedures. The *for* combinator can be viewed as a technique for defining a static or macro-expansion of the text. Essentially it provides a way of generating an expression with semicolon combinators which one would naturally associate with a program containing a sequence of statements. The environment can be viewed in a similar way. Consider the following expansion:

```
M⟦begin integer a,b; a:=a+b end⟧([])
  = def la: newloc();
    def lb: newloc();
    M⟦a:=a+b⟧([a ↦ la, b ↦lb]);
    epilogue({la,lb})

  = def la: ... ;          def lb: ... ;
    def va: contents(la); def vb: contents(lb);
    assign(la,(va+vb));
    epilogue ...
```

The environment argument has been completely eliminated in the expansion. The final expression is the one which one might write down for the program in giving an informal explanation. The role of the environment is to avoid the problem of naming the locations for an unknown number of new identifiers.

The ellipsis points relating to procedures in the semantics for *Blocks* should now be replaced. It is clear that, to agree with what has been done above, the meaning of a call statement must be:

$M: Call \to ENV \to STORE \overset{\sim}{\to} STORE$

A procedure denotation must be generated, and recorded in the environment, which makes the derivation of the call statement semantics possible. It would be a mistake to store the text of the procedure because of the choice of most programming languages to bind names to the textual (or static) environment. Thus in:

```
begin Boolean a;
      procedure p ... a ... ;
      ...
      begin integer a; ... p ...  end; ...
end
```

the invocation of 'p' from the inner block must give rise to a reference to the Boolean variable 'a'. In order to bind the identifiers in the text of the procedure to the appropriate locations, the procedure denotation is made into a function as follows:

$$DEN = \ldots \mid PROCDEN$$
$$PROCDEN = SCALARLOC^* \rightarrow STORE \xrightarrow{\sim} STORE$$

The outer argument to this function is a list of locations. Thus it can be seen that parameter passing is to be by location (or reference). This gives rise to:

$$M[\![mk\text{-}Call(pid,al)]\!](\rho) \triangleq$$
$$\quad \underline{let}\ f = \rho(pid)\ \underline{in}$$
$$\quad \underline{let}\ locl = \langle Mloc[\![al[i]]\!](\rho) \mid i \in \underline{inds}\, al \rangle\ \underline{in}\ f(locl)$$

Procedure denotations are to be functional and their generation can be defined by:

$$M[\![mk\text{-}Proc(pl,s)]\!](\rho) \triangleq$$
$$\quad \underline{let}\ f(locl) = (\underline{let}\ \rho' = \rho + [pl[i] \mapsto locl[i] \mid i \in \underline{inds}\, pl]\ \underline{in}$$
$$\qquad\qquad M[\![s]\!](\rho'))\ \underline{in}$$
$$\quad f$$

This function is used in defining the denotation of a block:

$$\underline{def}\ \rho' : \rho + (\ \ldots \cup [id \mapsto M[\![proc m(id)]\!](\rho) \mid id \in \underline{dom}\, proc m]\ )$$

It should, by now, be clear that the use of an abstract syntax is simplifying the definition by avoiding representation details of a language. When faced with a large language or system, there is considerable scope for choice in "how abstract" the syntax should be. Clearly, minor syntactic variants (e.g. abbreviations or default values) should be subsumed. It is also obvious that constructs with different semantics cannot have the same abstract representation. These extremes do not, how-

ever, provide a clear rule for deciding when to abstract (e.g. the problem of factored declarations in Pascal).

4.6 RECURSIVE PROCEDURES

The definition of recursive procedures requires no new features in the meta-language but it does warrant some words of explanation. The context condition given above for blocks explicitly prohibits the direct call by a procedure of itself. All that is necessary to permit such calls, or to admit mutual recursion among procedures, is to use the intended environment in checking the well-formedness of procedures. Thus:

$$
\begin{aligned}
&WF[\![mk\text{-}Block(vars,procm,sl)]\!](\rho) \triangleq \\
&\quad vars \cap \underline{dom}\, procm = \{\} \ \wedge \\
&\quad (\underline{let}\ \rho' = \rho + ([id \mapsto \underline{\text{BOOL}} \mid id \in vars] \ \cup \\
&\qquad\qquad [id \mapsto ATTR[\![procm(id)]\!] \mid id \in \underline{dom}\, procm]\ )\ \underline{in} \\
&\qquad (\forall p \in \underline{rng}\, procm)(WF[\![p]\!](\rho')) \ \wedge \\
&\qquad (\forall s \in \underline{elems}\, sl)(WF[\![s]\!](\rho')))
\end{aligned}
$$

A similar change will suffice to extend the semantic equations so that the recursive case is covered:

$$
\begin{aligned}
&M[\![mk\text{-}Block(vars,procm,sl)]\!](\rho) \triangleq \\
&\quad \underline{def}\ \rho' : \rho + ([id \mapsto newloc()) \mid id \in vars] \ \cup \\
&\qquad\qquad [id \mapsto M[\![procm(id)]\!](\rho') \mid id \in \underline{dom}\, procm]\ ); \\
&\quad \ldots
\end{aligned}
$$

But this recursive use of the extended environment (in creating the extended environment) requires some comment. Mathematically, there is no difficulty: chapter 3 explains how the least fixed point of recursive equations can be found and the ordering on functions can be used to define one on environments. In programming terms, it is also easy to accept that the denotations will be available by the time they are called. It is, however, possible to provide another explanation using the macro-expansion view discussed above. Consider:

$$
\begin{aligned}
&M[\![\underline{\text{begin}}\ \underline{\text{procedure}}\ p1;\ldots p2\ldots; \\
&\qquad \underline{\text{procedure}}\ p2;\ldots p1\ldots p2\ldots; \\
&\quad \ldots p1\ldots\ \underline{\text{end}}]\!]([\,])
\end{aligned}
$$

$$= \underline{let}\ \rho' = [p1 \mapsto M[\![...p2...]\!](\rho'),$$
$$p2 \mapsto M[\![...p1...p2...]\!](\rho')\ ]\ \underline{in}$$
$$M[\![...p1...]\!](\rho')$$

If names are introduced to stand in place of the procedure denotations this becomes:

$$\underline{let}\ pden1 = \ldots\ pden2\ \ldots \qquad \underline{in}$$
$$\underline{let}\ pden2 = \ldots\ pden1\ \ldots\ pden2\ \ldots \qquad \underline{in}$$
$$\ldots\ pden1\ \ldots$$

The environment has been eliminated and has thus removed the recursion which was causing concern. There remains a recursion on the (names of) procedure denotations but this is the recursion on functions from stores to stores which has occurred elsewhere. The environment can therefore be viewed as a part of the process of expanding a program into an (expression for the) denotation of a program.

4.7 EXCEPTIONS

There are a number of concepts in systems which permit the definition of exceptional sequencing: the 'goto' statement in languages, exception traps or error handlers in both languages and other systems and the definition of the effect of some errors. There is often heated controversy about the wisdom of such features. It is not the intention here to enter into such debates. Providing a definition method is capable of defining such features, it becomes a tool with which one can compare alternative approaches. The purpose here is to show how the denotational approach can define various forms of 'goto' statements. Other chapters in this book apply the same model to other exception handling problems. It is possible to envisage 'goto' statements of varying degrees of generality and the implications on the definition are studied here. The simplest form serves to introduce the general idea. A program might restrict its use of 'goto' statements to defining the flow of control within a single phrase structure. Thus:

begin st1; lab1:st2; st3; goto lab2; st4; lab2:goto lab1 end

An abstract syntax which covers this case is:

```
Program    :: Namedstmt*
Namedstmt  :: s-lab:[Id] s-body:Stmt
Stmt       =  ... | Goto
Goto       :: Id
```

It is clear that some space of denotations is required for named statements, but what is this to be? The meaning of a simple named statement might be given by:

$M: Namedstmt \rightarrow ?$

However, the denotational approach requires that the denotation of a list of such statements be derived (only) from their individual denotations. The problem is that it is precisely the fact that a "goto" statement appoints a successor (which is not a component) that has to be captured as its semantics. There are two ways of providing a denotational definition: *exit*s and continuations. These approaches are compared in the next chapter, here the *exit* approach is described. The basis of the *exit* approach is to extend the transformations, which are used as denotations, so that they can carry an indication of abnormal termination. Thus:

$$TR = STORE \xrightarrow{\sim} STORE \times [ABNORMAL]$$

The effect is to elevate the abnormal sequencing to something which is anticipated; the cost is that denotations become more difficult to compose but combinators are provided below which ameliorate this problem. The second component of the range of a transformation is *nil* in the case of normal return; but will contain some indication of what is to be done next in the abnormal case. The choice of the *ABNORMAL* values depends on the system being defined. For the current language it will be sufficient to use the labels themselves. Thus:

$$TR = STORE \xrightarrow{\sim} STORE \times [Id]$$

The meaning of a 'goto' statement can be given by:

$$M[mk\text{-}Goto(id)] \triangleq \lambda\sigma.(\sigma, id)$$

The meaning of a 'goto' statement is to leave the store unchanged but to appoint an explicit successor. The meaning of, for example, an assignment statement is to change the store and to signal normal return:

$$M[mk\text{-}Assign(vr,e)] \triangleq$$
$$\lambda\sigma.(\underline{let}\ l = Mloc[vr]\sigma\ \underline{in}$$
$$\underline{let}\ v = M[e]\sigma\ \underline{in}$$
$$assign(l,v)(\sigma),\underline{nil})$$

The next problem to be resolved is the combination of two transformations to yield another of the same type. If the first transformation yields a $\underline{nil}$ abnormal component, then the second transformation is to be composed with the store to store part of the first. Alternatively, if the first transformation yields an abnormal return, this must be propagated. Thus, formally:

$$Ml: Namedstmt^* \rightarrow TR$$
$$Ml[<>] \triangleq \lambda\sigma.(\sigma,\underline{nil})$$
$$Ml[nsl] \triangleq (\lambda\sigma,a.\underline{if}\ a=\underline{nil}\ \underline{then}\ Ml[\underline{tl}nsl](\sigma)\ \underline{else}\ (\sigma,a))\circ$$
$$M[s\text{-}body(\underline{hd}nsl)]$$

For the program shown above:

$$M[\text{st4}] = \lambda\sigma.(f(\sigma),\underline{nil})$$
$$M[\text{lab2:}\underline{\text{goto}}\ \text{lab1}] = \lambda\sigma.(\sigma,\text{lab1})$$
$$Ml[\text{st4;lab2:}\underline{\text{goto}}\ \text{lab1}] = \lambda\sigma.(f(\sigma),\text{lab1})$$

etc. The final problem is to show how the abnormal return values are handled. The meaning of any label is the transformation determined by beginning execution at that label. If such a transformation ends with an abnormal exit, the meaning of another label must be composed with that transformation. Since '$\underline{\text{goto}}$' statements can be used to define loops, it is convenient to make this trap (see r below) recursive. Thus:

$$M[mk\text{-}Program(nsl)] \triangleq$$
$$let\ \rho = [id \mapsto Ml[sel(id,nsl)] \mid id \in dlabs(nsl)]\ \underline{in}$$
$$\underline{let}\ r = \lambda\sigma,a.\underline{if}\ a \in \underline{dom}\rho\ \underline{then}\ r(\rho(a)(\sigma))\ \underline{else}\ (\sigma,a)\ \underline{in}$$
$$r\circ Ml[nsl]$$

Where:

$$dlabs(nsl) \triangleq \{id \mid (\exists i \in \underline{inds}nsl)(s\text{-}lab(nsl[i])=id \wedge id \neq \underline{nil})\}$$

$sel(id,nsl)$ selects the subsequence of nsl whose first statement is labelled with id.

(It has been assumed here that context conditions have established that each label occurs as the name of at most one statement and that 'goto' statements refer only to valid labels.) It is interesting to note that the recursion in r can again be eliminated by macro expansion. The above definition is the one required but it is clouded by the explicit mention of stores etc. A group of combinators can be defined which greatly ease the task of understanding a definition. Thus:

$$exit\ a = \lambda\sigma.(\sigma,a)$$

An object, say t, of type *STORE* to *STORE* is automatically interpreted as a transformation if required by context:

$$t = \lambda\sigma.(t(\sigma),nil)$$

The 'semicolon' combinator is redefined to give the semantics used above:

$$t1;t2 = (\lambda\sigma,a.\underline{if}\ a=nil\ \underline{then}\ t2(\sigma)\ \underline{else}\ (\sigma,a))\circ t1$$

The "trapping" effect is achieved by the *tixe* (*exit* backwards) combinator:

$$(\underline{tixe}\ m\ \underline{in}\ t) \triangleq \underline{let}\ \rho = m\ \underline{in}\ \underline{let}\ r = (\lambda\sigma,a.\ \underline{if}\ a\in\underline{dom}\rho\ \underline{then}\ r(\rho(a)(\sigma))\ \underline{else}\ (\sigma,a))\ \underline{in}\ r\circ t$$

The use of these combinators permits the earlier definition to be re-written as:

$$M[mk\text{-}Program(nsl)] \triangleq \underline{tixe}\ [id \mapsto Ml[sel(id,nsl)] \mid id\in dlabs(nsl)]\ \underline{in}\ Ml[nsl]$$

$$Ml[<>] \triangleq I_{STORE}$$

$$Ml[nsl] \triangleq M[s\text{-}body(\underline{hd}nsl)];Ml[\underline{tl}nsl]$$

$$M[mk\text{-}Goto(id)] \triangleq \underline{exit}(id)$$

Thus, the semantics of the example given at the beginning of this section is:

```
tixe [lab1 ↦ Ml[<st2,st3,goto lab2,st4,goto lab1>],
      lab2 ↦ Ml[<goto lab1>]] in
Ml[<st1,st2,st3,goto lab2,st4,goto lab1>]
```

Little change is required to define a language in which 'goto' statements can leave (and thus close) static phrase structures. Thus with:

```
Block :: .... s-body:Namedstmt*
```

it is only necessary to ensure that the epilogue semantics are performed even on abnormal block termination. Thus for:

```
begin ...
   begin ... goto lab ... end;
   ... lab: ...; ...
end
```

```
M[mk-Block(vars,procm,nsl)](ρ) ≜
    def ρ' : ... ;
    (λσ,a.(epilogue(...)(σ),a))°
        (tixe [id ↦ Ml[sel(id,nsl)](ρ') | id∈dlabs(nsl)] in
         Ml[nsl](ρ'))
```

Again a combinator can be defined:

```
always t1 in t2 = (λσ,a.(t1(σ),a))°t2
```

The *always* combinator can then be used to give:

```
M[mk-Block(vars,procm,nsl)](ρ) ≜
    def ρ' : ... ;
    always epilogue(...)
                                                        in
    (tixe [id ↦ Ml[sel(id,nsl)](ρ') | id∈dlabs(nsl)]   in
     Ml[nsl](ρ'))
```

Some languages permit 'goto' statements to branch into a phrase structure. Thus, in ALGOL 60:

```
begin goto lab; ... ;if p then lab:st1 else st2; ... end
```

is valid. The CPL language [Barron 63a] even allowed 'goto' statements into loops and blocks. The definition of such a language can use the

same transformations but must define *cue* functions which determine the appropriate starting point. *Cue* functions occur in the next chapter and their use in ALGOL 60 is shown in Chapter 6.

Languages in which "goto" statements can terminate dynamic objects like procedures present a further complication. Consider:

```
begin procedure p; ... goto lab; ... ;
      ...
      lab:...; ...
      call p; ...
end
```

The dynamic invocation of 'p' can be closed by the 'goto' statement. There are two problems which are closely akin to those treated above with variables. Firstly there might be other instances of the same label occurring in the environment of the call of 'p' - ALGOL 60 requires the "goto" to ignore such labels in the dynamic environment and to locate that instance of 'lab' in the static (textual) environment of 'p'. This problem might be resolved by making static changes to identifiers to make them unique. This solution would, however, not work for a language in which procedures can be passed as parameters across recursive calls. In this case it is necessary to have a precise indication of the activation in question. In the case of variables, new locations are chosen with respect to the domain of the store. For labels, a new set of *Activation identifiers* is introduced and appended to label denotations. These *Label denotations* are stored in the environment. Thus the definition might be written:

$$Block :: \ldots\ s\text{-}procm:(Id \xrightarrow{m} Proc)\ \ s\text{-}body:Namedstmt^*$$

$$TR = STORE \xrightarrow{\sim} STORE \times [LABDEN]$$

$$LABDEN :: AID\ \ Id$$

$$AID \quad \text{An infinite set}$$

$$ENV = Id \xrightarrow{m} DEN$$

$$DEN = \ldots \mid PROC \mid LABDEN$$

$$PROCDEN = SCALARLOC^* \times AID\text{-}set \rightarrow TR$$

$$M: Block \rightarrow ENV \rightarrow AID\text{-}set \rightarrow TR$$

```
M[mk-Block(...,procm,nsl)](ρ)(cas) ≜
    let aid∈(AID-cas)                                          in
    def ρ' : ρ + (...∪
                  [id ↦ mk-LABDEN(aid,id) | id∈dlabs(nsl)] );
    always epilogue(...)                                       in
    (tixe ...                                                  in
     Ml[nsl](ρ')(cas ∪ {aid}))

M[mk-Goto(id)](ρ)(cas) ≜ exit(ρ(id))
```

The set *(cas)* of current activation identifiers must be passed along the dynamic calling sequence and a convenient way of doing this is shown below.

It is possible to prove useful results about such a definition. The most important such theorem shows that, for well-formed programs with 'goto' statements referring only to statically known labels, there is no way that a (dynamic) jump can lead to an inactive or closed piece of text. This is true of ALGOL 60 even with procedures and labels being allowed as parameters. Even the 'switch' declaration is so constrained as to preserve the property. This important property is, however, lost as soon as (unconstrained) label or procedure variables are allowed. ALGOL 68 [van Wijngaarden 75a] manages to keep the lid on this Pandora's box by syntactic rules. PL/I [ECMA 76a] is less inhibited. The need to define which such uses are in error, gives rise in [Bekić 74a] to a new entity which keeps track of the active activations.

As mentioned above, there are other aspects of systems which can be conveniently defined using *exit*s. The most prevalent class of such features is error handling. Three types of errors must be distinguished:

(i) for some system, it might be required that certain diagnostics are to be produced by any implementation, such error handling becomes part of the system definition like any other feature.

(ii) for many systems which are to be implemented more than once, there are some user errors for which it is thought unwise to constrain the implementation treatment (for example referencing uninitialized variables in a programming language).

(iii) any misuse of the meta-language which makes the definition meaning-

less (for example mismatch of arguments and parameters to semantic functions) must clearly be avoided.

It is the second category of error which remains to be discussed. The definition must show that certain use leads to an error but leaves open what action follows the error. Some implementations may check and produce a useful diagnostic; others may omit the necessary checking code and just run on after the error. The definition is essentially stating further constraints on the domain of valid inputs but, unlike context conditions, invalidity is in general only detectable dynamically. The error situations where further computation is undefined are indicated in the definition by writing:

error

The semantics being the same as:

exit(error)

with the additional rule that no *fixe* can trap such an exit value. Examples of the use of *error* occur below in the handling of input, reference to variables and evaluation of subscripts. It might be argued that the presence of features which result in undetected errors makes a system dangerous to use. Once again, no position is taken on this (pragmatic) issue here: the definition method provides a way of indicating and checking for such "features".

4.8 STORAGE MODEL

In the *STORE* above, only one type of variable has been considered; furthermore only scalar variables are allowed. Some modest extensions can now be considered. The syntax of a block can be extended to permit declarations of different types of variable:

Block :: *s-dclm:(Id* m *Scalartype)*...
Scalartype = INT | BOOL

Clearly the range of store must be extended to permit the storage of appropriate values. Thus:

$$
\begin{array}{ll}
STORE & = SCALARLOC \xrightarrow{m} [SCALARVALUE] \\
SCALARLOC & = \text{Infinite Set} \\
SCALARVALUE & = Bool \mid Int
\end{array}
$$

The environment is also suitably extended. If variables of either type are to be passed (by location) as parameters, type checking must be included. Different languages vary as to whether this checking can be done statically or dynamically. A dangerous lacuna can result from passing procedures as parameters: ALGOL 60 loses type control; Pascal retains it but introduces a restriction; ALGOL 68 erects a whole structure of types in order to give a complete solution.

The other extension of the storage model to be considered here is the treatment of arrays. The syntactic extension is:

$$
\begin{array}{lll}
Block & :: & s\text{-}dclm:(Id \xrightarrow{m} Type) \ \ldots \\
Type & :: & Scalartype \ \ [Expr^{+}]
\end{array}
$$

A non-*nil* subscript list defines the dimensions of an array. A *nil* subscript list being used to indicate scalar variables. A check must be made on block entry to establish that all expressions evaluate to positive integers. What is to be done with *STORE*? This depends on other language features. If arrays are always handled as a unit, the structure of the array value can be made part of store:

$$
\begin{array}{ll}
STORE & = LOC \xrightarrow{m} VALUE \\
VALUE & = SCALARVALUE \mid ARRAYVALUE \\
ARRAYVALUE & = Int^{+} \xrightarrow{m} [SCALARVALUE]
\end{array}
$$

Alternatively, if elements of arrays are to be passed (by location) as arguments to procedures, then locations themselves must become structured objects. Thus:

$$
\begin{array}{ll}
STORE & = SCALARLOC \xrightarrow{m} [SCALARVALUE] \\
ENV & = Id \xrightarrow{m} DEN \\
DEN & = ARRAYLOC \mid SCALARLOC \\
ARRAYLOC & = Int^{+} \xleftrightarrow{m} SCALARLOC
\end{array}
$$

(Notice that members of *ARRAYLOC* are one-one mappings.) This is the combination of features defined below. Dynamic checking of array bounds is also indicated.

The treatment of records is similar to arrays (cf. chapter 7). Some languages require more basic extensions. In particular, some of the PL/I features need an implicit characterization of locations (see [Bekić 71b, 74a]).

4.9 STATES

The transformations considered above have been concerned only with *STORE*. The main criteria for choosing to place entities in the domain and range of transformations is that they can be both read and written. It is for this reason that the environment is treated separately. Certain language features prompt the need for further state components and these are handled by using structured objects (*STATE*) in the domain and range of transformations.

One language area requiring more state components is input/output statements. Suppose the language includes the notion of one input (read only) and one output (write only) "file". If the only values which can be handled by input/output are integers the state becomes:

$$\begin{aligned} &\mathit{STATE} :: \ \mathrm{STR}{:}\mathit{STORE} \quad \mathrm{IN}{:}\mathit{Int}^* \quad \mathrm{OUT}{:}\mathit{Int}^* \\ &\mathit{TR} = \mathit{STATE} \xrightarrow{\sim} \mathit{STATE} \times [\mathit{LABDEN}] \end{aligned}$$

Creation of an initial state and disposal of the final state is illustrated below. The final combinators to be given here provide clearer reference to components of the state. Thus:

$$\begin{aligned} &\mathrm{IN} := e = \lambda\sigma.\mathit{mk\text{-}STATE}(\mathrm{STR}(\sigma), e, \mathrm{OUT}(\sigma)) \\ &\underline{c}\ \mathrm{IN} = \lambda\sigma.\mathrm{IN}(\sigma) \end{aligned}$$

then (using the syntax of the next section):

$$\begin{aligned} &\mathit{M}[\mathit{mk\text{-}In}(vr)](\rho) \triangleq \\ &\quad \underline{\mathit{if}}\ \underline{c}\ \mathrm{IN} = <> \ \underline{\mathit{then}}\ \underline{\mathit{error}} \\ &\quad \underline{\mathit{else}}\ (\underline{\mathit{def}}\ v: \ \underline{\mathit{hd}}\ \underline{c}\ \mathrm{IN}; \\ &\qquad\quad \underline{\mathit{def}}\ l: \ \mathit{Mloc}[vr](\rho); \\ &\qquad\quad \mathrm{STR} := \underline{c}\ \mathrm{STR} + [l \mapsto v]; \\ &\qquad\quad \mathrm{IN} := \underline{\mathit{tl}}\ \underline{c}\ \mathrm{IN}) \end{aligned}$$

4.10 A DEFINITION

The definition of a small language illustrating all of the points made in this chapter can now be given. The main static and dynamic semantic functions are split under the syntactic objects; auxiliary functions are collected at the end of the section.

Abbreviations

Names of sets are abbreviated as indicated in *italics:*

*Bool*ean
*Const*ant
*decl*aration
*den*otation
*env*ironment
*expr*ession
*id*entifier
*int*eger
*loc*ation
operator
*param*eter
*proc*edure
*ref*erence
right hand side
*stat*ement
*trans*formation
*var*iable

The following type clause abbreviations (~) are used:

$M: D \Rightarrow \quad \sim \quad M: D \to Tr$

$M: D \Rightarrow R \quad \sim \quad M: D \to STATE \xrightarrow{\sim} (STATE \times [LABDEN] \times R)$

Static Environment

The validity of an abstract program with respect to context conditions is defined by the *WF* function. This function creates and uses a static environment which contains attribute information. This same *Staticenv* is used by the *TP* function which determines the types of expressions etc. (*Attr* is defined in the section on procedures below.)

$Staticenv = Id \xrightarrow{m} (Attr \mid \underline{LABEL} \mid Procattr)$
$Procattr :: Attr^*$

Certain obvious steps have been taken to shorten the *WF* functions given below. For example, if:

$Q :: Q_1\ Q_2\ \ldots\ Q_n$

then a rule (or part of a rule) of the form:

$$WF[mk\text{-}Q(Q_1,Q_2,\ldots,Q_n)](senv) \triangleq WF[Q_1](senv) \wedge WF[Q_2](senv) \wedge \ldots \wedge WF[Q_n](senv)$$

is omitted.

Semantic Objects

The state for this simple language contains the values for the (scalar) locations, the set of activation identifiers in use and the input and output files:

$$\begin{array}{lll} STATE & :: & \text{STR: } STORE \\ & & \text{AIDS: } AID\text{-}set \\ & & \text{IN: } Int^* \\ & & \text{OUT: } Int^* \\ STORE & = & SCALARLOC \xrightarrow{m} [SCALARVALUE] \\ SCALARLOC & & \text{Infinite set} \\ SCALARVALUE & = & Bool \mid Int \\ AID & & \text{Infinite set} \end{array}$$

The denotation of (local) identifiers are contained in an environment which is a parameter to the meaning function (M).

$$\begin{array}{lll} ENV & = & Id \xrightarrow{m} DEN \\ DEN & = & LOC \mid LABDEN \mid PROCDEN \\ LOC & = & SCALARLOC \mid ARRAYLOC \\ ARRAYLOC & = & Nat^+ \xrightarrow{m} SCALARLOC \text{ where } al \in ARRAYLOC \supset (\exists nl)(\underline{dom}\, al = rect(nl)) \\ LABDEN & :: & s\text{-}aid: AID \quad s\text{-}lab: Id \\ PROCDEN & = & LOC^* \to TR \end{array}$$

Transformations reflect possible abnormal termination:

$$TR = STATE \xrightarrow{\sim} STATE \times [LABDEN]$$

Programs

$$Program :: Stmt$$

```
WF[mk-Program(s)] ≜ WF[s]([])
type: Program → Bool
```

comment the empty environment passed to the context condition for statements reflects the fact that there are no (global) variables.

```
M[mk-Program(s)](inl) ≜
    let state0 = mk-STATE([],{},inl,<>) in
    OUT(M[s]([])(state0))
type: Program → Int* → Int*
```

comment the only result of executing a program is its output list of values - the state transition is purely local.

Statements

```
Stmt = Block | If | While | Call | Goto |
       Assign | In | Out | NULL
```

Within this section the following types are to be assumed unless otherwise stated.

```
WF: Stmt → Staticenv → Bool
M:  Stmt → ENV → TR

Block :: s-dclm:Id m Dcl s-procm:Id m Proc s-body:Namedstmt*

WF[mk-Block(dclm,procm,nsl)](senv) ≜
   let labl = contndll(nsl) in
   is-uniquel(labl) ∧
   is-disjointl(<elemslabl,domprocm,domdclm>) ∧
   (let lenv = [id ↦ ATTR[dclm(id)] | id∈domdclm] ∪
               [id ↦ ATTR[procm(id)] | id∈domprocm] ∪
               [id ↦ LABEL | id∈elemslabl] in
    let renv = senv \ domlenv in
    let nenv = senv + lenv in
    (∀dcl∈rngdclm)(WF[dcl](renv)) ∧
    (∀proc∈rngprocm)(WF[proc](nenv)) ∧
    (∀ns∈elemsnsl)(WF[ns](nenv)))
```

comment *renv* is used for declarations because local variables should not be used in defining array bounds - use of *nenv* for procedures shows that (mutual) recursion is permitted in the language.

```
M[mk-Block(dclm,procm,nsl)](env) ≙
    def cas: c s-aids;
    let aid ∈ (AID - cas) in
    s-aids := c s-aids ∪ {aid};
    def nenv: env +
        ([id ↦ M[dclm(id)](env)    | id∈domdclm]          ∪
         [id ↦ M[procm(id)](nenv)  | id∈domprocm]         ∪
         [id ↦ mk-LABDEN(aid,id)   | id∈elemscontndll(nsl)]);
    always epilogue(domdclm,aid)(nenv)
    in (tixe [mk-LABDEN(aid,id) ↦ M[sel(id,nsl)](nenv)
                                | id∈elemscontndll(nsl)] in
        M[nsl](nenv))
```

note: declarations, procedures and *epilogue* are defined after the remaining statements.

```
Namedstmt :: s-nm:[Id] s-body:Stmt

M[nsl](env) ≙
   for i = 1 to lennsl do M[s-body(nsl[i])](env)
type: Namedstmt* → ENV =>

If :: s-test:Expr  s-th:Stmt  s-el:Stmt
```

comment the branches of the conditional cannot be labelled.

```
WF[mk-If(e,th,el)](senv) ≙ TP[e](senv) = mk-Scalarattr(BOOL)

M[mk-If(e,th,el)](env) ≙
   def b: M[e](env);
   if b then M[th](env) else M[el](env)

While :: s-test:Expr  s-body:Stmt

WF[mk-While(e,s)](senv) ≙ TP[e](senv)=mk-Scalarattr(BOOL)
```

```
M[mk-While(e,s)](env) ≜
    let wh=(def v:M[e](env); if v then M[s](env);wh else I_STATE)
    in wh
```

comment wh is defined recursively.

```
Call :: s-pn:Id  s-app:Varref*

WF[mk-Call(pid,apl)](senv) ≜
    pid∈domsenv ∧ senv(pid)∈Procattr ∧
    (let mk-Procattr(fpl) = senv(pid) in
     lenapl=lenfpl ∧ (∀i∈indsfpl)(TP[apl[i]](senv)=fpl(i)))
```

comment the actual parameters must match the formal parameter type

```
M[mk-Call(pid,apl)](env) ≜
    def locl: <M[apl(i)](env) | 1≤i≤lenapl>;
    let f = env(pid) in
    f(locl)
```

comment creation of procedure denotations is shown in the handling of *Procedure* below.

```
Goto :: s-lab:Id

WF[mk-Goto(lab)](senv) ≜ lab∈domsenv ∧ senv(lab)=LABEL

M[mk-Goto(lab)](env)  ≜ exit (env(lab))

Assign :: s-lhs:Varref  s-rhs:Expr

WF[mk-Assign(lhs,rhs)](senv) ≜ TP[rhs](senv)=TP[lhs](senv)

M[mk-Assign(lhs,rhs)](env)  ≜  def loc :  M[lhs](env);
                               def v   :  M[rhs](env);
                               STR     :=  c STR + [loc ↦ v]
```

```
In :: s-var:Varref

WF[mk-In(vr)](senv) Δ TP[vr](senv)=mk-Scalarattr(INT)

M[mk-In(vr)](env)   Δ  def inl: c IN;
                       if inl = <> then error
                       else (def loc :  M[vr](env);
                             IN       := tlinl;
                             STR      := c STR + [loc ↦ hdinl])

Out :: s-val:Expr

WF[mk-Out(e)](senv) Δ TP[e](senv)=mk-Scalarattr(INT)

M[mk-Out(e)](env)   Δ  def v :  M[e](env);
                       OUT   := c OUT^<v>

M[NULL](env)        Δ I_STATE
```

Declarations

```
Dcl        =  Scalardcl | Arraydcl
Scalardcl  :: Scalartype
Arraydcl   :: s-sctp:Scalartype  s-bdl:Expr+
Scalartype =  INT | BOOL

ATTR: Dcl → Attr

ATTR[mk-Scalardcl(sctp)]    Δ mk-Scalarattr(sctp)

ATTR[mk-Arraydcl(sctp,bdl)] Δ mk-Arrayattr(sctp,lenbdl)

WF[mk-Arraydcl(sctp,bdl)](senv) Δ
    (∀bd∈elemsbdl)(TP[bd](senv)=mk-Scalarattr(INT))
type: Arraydcl → Staticenv → Bool

M[mk-Scalardcl(sctp)](env) Δ  def ulocs : dom c STR;
                              let l∈(SCALARLOC - ulocs) in
                              STR := c STR ∪ [l ↦ nil];
                              return(l)
```

```
epilogue(ids,aid)(env) ≜
    let scelocs = {env(id) | id∈ids ∧ env(id)∈SCALARLOC} ∪
        union {rng(env(id)) | id∈ids ∧ env(id)∈ARRAYLOC} in
    STR  := c STR \ scelocs;
    AIDS := c AIDS - {aid}

type: Id-set × AID → ENV → TR

M[mk-Arraydcl(sctp,bdl)](env) ≜
    def bdvl : <M[bdl(i)](env) | 1≤i≤lenbdl>;
    if (∃i∈indsbdvl)(bdvl(i)<1) then error
    else (def ulocs : dom c STR;
          let al∈ARRAYLOC be s.t. is-disjointl(<ulocs,rngal>) ∧
                                 domal = rect(bdvl)          in
          STR := c STR = [scl ↦ nil | scl∈rngal];
          return(al))
type: Dcl → ENV => LOC
```

Procedures

```
Proc       :: s-fpl:Parm*  s-body:Stmt
Parm       :: s-nm:Id  s-attr:Attr
Attr       =  Scalarattr | Arrayattr
Scalarattr :: Scalartype
Arrayattr  :: s-sctp:Scalartype  s-bdinf:Nat

ATTR[mk-Proc(fpl,s)] ≜
   mk-Procattr(<s-attr(fpl(i)) | 1≤i≤lenfpl>)
type: Proc → Procattr

WF[mk-Proc(fpl,s)](senv) ≜
   is-uniquel(<s-nm(fpl[i]) | 1≤i≤lenfpl>) ∧
   (let nenv=senv + [s-nm(fpl[i]) ↦ s-attr(fpl[i]) | i∈indsfpl] in
    WF(s)(nenv))

M[mk-Proc(pl,s)](env) ≜
   let f(al)=(let nenv = env + [s-nm(pl[i]) ↦ al[i] | i∈indspl] in
              M[s](nenv))
   in f
type: Proc → ENV → PROCDEN
```

comment note that it is the environment of the declaring block (static) which is used as the basis for *nenv*

Expressions

Expr = Infixexpr | Rhsref | Const

In this section functions, unless otherwise stated, are of types:

WF: Expr → Staticenv → Bool
TP: Expr → Staticenv → Attr
M: Expr → ENV => SCALARVALUE

Infixexpr :: Expr' Op Expr

Op = Intop | Boolop | Comparisonop

WF[mk-Infixexpr(e1,op,e2)](senv) $\triangleq$
*op∈Intop ∧ TP[e1](senv)=TP[e2](senv)=mk-Scalarattr(*INT*)* ∨
*op∈Boolop ∧ TP[e1](senv)=TP[e2](senv)=mk-Scalarattr(*BOOL*)* ∨
*op∈Comparisonop ∧ TP[e1](senv)=TP(e2)(senv)=mk-Scalarattr(*INT*)*

TP[mk-Infixexpr(e1,op,e2)](senv) $\triangleq$
*if op∈Intop then mk-Scalarattr(*INT*) else mk-Scalarattr(*BOOL*)*

M[mk-Infixexpr(e1,op,e2)](env) $\triangleq$ *def v1 : M[e1](env);*
def v2 : M[e2](env);
return M[op](v1,v2)

M for the various operators yields their meaning:

M: Op → (SCALARVALUE × SCALARVALUE → SCALARVALUE)

Rhsref :: Varref
Varref :: s-nm:Id s-bdp:[Expr⁺]

WF[mk-Rhsref(vr)](senv) $\triangleq$ *TP[vr](senv)∈Scalarattr*

TP[mk-Rhsref(vr)](senv) $\triangleq$ *TP[vr](senv)*

```
WF[mk-Varref(id,bdp)](senv) Δ
    id∈domsenv ∧ senv(id)∈Attr ∧
    (bdp=nil ∧ senv(id)∈Scalarattr) ∨
    (senv(id)∈Arrayattr ∧
          (let mk-Arrayattr(sctp,dim) = senv(id) in
           lenbdp = dim ∧
           (∀bd∈elemsbdp)(TP[bd](senv)=mk-Scalarattr(INT))))
type: Varref → Staticenv → Bool

TP[mk-Varref(id,bdp)](senv) Δ
    if senv(id)∈Scalarattr then senv(id)
    else if bdp≠nil then (let mk-Arrayattr(sctp,bdi) = senv(id) in
                          mk-Scalarattr(sctp))
         else senv(id)
type: Varref → Staticenv → Attr

M[mk-Rhsref(vr)](env) Δ def loc: M[vr](env);
                        def v: (c STR)(loc);
                        if v=nil then error else return(v)
```

comment note how the location is evaluated by access to store because the right hand side contexts require values.

```
M[mk-Varref(id,bdp)](env) Δ
    if bdp=nil then return(env(id))
    else (let aloc = env(id) in
          def esscl: <M[bdp(i)](env) | 1≤i≤lenbdp>;
          if ¬(esscl∈domaloc) then error else return(aloc(esscl)))
type: Varref → ENV => LOC

Const = Intconst | Boolconst

TP: Intconst → Staticenv → {mk-Scalarattr(INT)}

TP: Boolconst → Staticenv → {mk-Scalarattr(BOOL)}
```

M function is an identity for constant.

Auxiliary Functions

contndll: Namedstmt → Id**

yields the list of those identifiers used as *s-nm* part of the elements of the given statement list

is-uniquel: X → Bool*

indicates if a list contains unique elements (i.e. no duplicates)

is-disjointl: (X-set) → Bool*

indicates whether the sets in the list are pairwise disjoint

rect: Nat → (Nat*)-set*

generates the set of valid indices within the given bounds

sel: Id × Namedstmt → Namedstmt**

pre-sel(id,nsl) $\triangleq$ *id∈elemscontndll(nsl)*

returns the sublist whose first statement has *id* as a label providing that the arguments satisfy the pre-condition.

4.11 NON-DETERMINISM

It is observed above that this language definition fixes the order of evaluation of sub-expressions. This is in keeping with the decision to minimize the discussion of non-determinism in the current book. On the other hand, non-deterministic selection has been deliberately built into the selection of new locations and activation identifiers. The definition is showing that the particular choice made is irrelevant in the sense that it would not affect the overall outcome of the program. If, however, some particular choice were defined, it could make it more difficult to prove some implementations correct. Although admitting this non-determinism is useful for implementations, it would be unnecessarily confusing to employ relational or power domain denotations (cf. [Jones 81a], [Plotkin 76a]) to cope with this problem.