Chapter 10

Formal Syntax and Semantics

1 ◆ SYNTAX

A programming language is defined by specifying its **syntax** (structure) and its **semantics** (meaning). Syntax normally means context-free syntax because of the almost universal use of context-free grammars as a syntax-specification mechanism. Syntax defines what sequences of symbols are valid; syntactic validity is independent of any notion of what the symbols mean. For example, a context-free syntax might say that `A := B + C` is syntactically valid, while `A := B +;` is not.

Context-free grammars are described by **productions** in **BNF** (Backus-Naur Form, or Backus Normal Form, named after John Backus and Peter Naur, major designers of Algol-60). For example, part of the syntax of Pascal is shown in Figure 10.1.

Figure 10.1

```
Program ::= program IDENTIFIER ( FileList ) ;                    1
     Declarations begin Statements end .                         2
FileList ::= IDENTIFIER | IDENTIFIER , FileList                  3
Declarations ::= ConstantDecs TypeDecs VarDecs ProcDecs          4
ConstantDecs ::= const ConstantDecList | ε                       5
ConstantDecList ::= IDENTIFIER = Value; ConstantDecList | ε      6
```

The identifiers on the left-hand sides of the rules are called **nonterminals**. Each rule shows how such a nonterminal can be expanded into a collection of nonterminals (which require further expansion) and **terminals**, which are lexical tokens of the programming language. In our example, **program** and IDENTIFIER are terminals, and Program and Statements are nonterminals. I use | to indicate alternatives and ε to indicate an empty string.

Sometimes, a clearer notation may be used for BNF; the purpose of notation, after all, is to specify ideas precisely and clearly. The `FileList` and `ConstantDecList` productions use recursion to represent arbitrarily long lists. I can rewrite those productions as shown in Figure 10.2, introducing iteration for the recursion.

Figure 10.2

```
FileList ::= [ IDENTIFIER +,]                                    1
ConstantDecList ::= [ IDENTIFIER = Value ; *]                    2
```

Here I use brackets `[` and `]` to surround repeated groups. I end the group either with `*`, which means 0 or more times (line 2), with `+`, which means 1 or more times (line 1), or with neither, which means 0 or 1 time. Optionally, following the `*` or `+` is a string that is to be inserted between repetitions. So line 1 means that there may be one or more identifiers, and if there are more than one, they are separated by `,` characters. Line 2 means there may zero or more constant declarations; each is terminated by the `;` character. This notation obscures whether the repeated items are to be associated to the left or to the right. If this information is important, it can be specified in some other way, or the productions can be written in the usual recursive fashion.

The BNF specification is helpful for each of the three software-tool aspects of a programming language.

1. It helps the programming language designer specify exactly what the language looks like.
2. It can be used by automatic compiler-generator tools to build the parser for a compiler.
3. It guides the programmer in building syntactically correct programs.

BNF is inadequate to describe the syntax for some languages. For example, Metafont dynamically modifies the meanings of input tokens, so that it is not so easy to apply standard BNF.

BNF also fails to cover all of program structure. Type compatibility and scoping rules (for example, that `A := B + C` is invalid if `B` or `C` is Boolean) cannot be specified by context-free grammars. (Although context-sensitive grammars suffice, they are never used in practice because they are hard to parse.) Instead of calling this part of program structure static semantics, as has become customary, let me call it **advanced syntax**. Advanced syntax augments context-free specifications and completes the definition of what valid programs look like. Advanced syntax can be specified in two ways:

1. Informally via a programming language report, as is done for most programming languages. An informal specification can be compact and easy to read but is usually imprecise.
2. Formally (for example, via two-level van Wijngaarten grammars or attribute grammars).

Attribute grammars are one popular method of formal specification of advanced syntax. They formalize the semantic checks often found in compilers. As an example of attribute grammars, the production `E ::= E + T` might be augmented with a type attribute for `E` and `T` and a predicate requiring type compatibility, as shown in Figure 10.3.

Figure 10.3 $(E_2.type = numeric) \wedge (T.type = numeric)$

where E_2 denotes the second occurrence of E in the production. Attribute grammars are a reasonable blend of formality and readability, and they are relatively easy to translate into compilers by standard techniques, but they can still be rather verbose.

2 ◆ AXIOMATIC SEMANTICS

Semantics are used to specify what a program does (that is, what it computes). These semantics are often specified very informally in a language manual or report. Alternatively, a more formal **operational semantics** interpreter model can be used. In such a model, a program state is defined, and program execution is described in terms of changes to that state. For example, the semantics of the statement A := 1 is that the state component corresponding to A is changed to 1. The LISP interpreter presented in Chapter 4 is operational in form. It defines the execution of a LISP program in terms of the steps needed to convert it to a final reduced form, which is deemed the result of the computation. The Vienna Definition Language (VDL) embodies an operational model in which abstract trees are traversed and decorated to model program execution [Wegner 72]. VDL has been used to define the semantics of PL/I, although the resulting definition is quite large and verbose.

Axiomatic semantics model execution at a more abstract level than operational models [Gries 81]. The definitions are based on formally specified predicates that relate program variables. Statements are defined by how they modify these relations.

As an example of axiomatic definitions, the axiom defining var := exp usually states that a predicate involving var is true after statement execution if and only if the predicate obtained by replacing all occurrences of var by exp is true beforehand. For example, for y > 3 to be true after execution of the statement y := x + 1, the predicate x + 1 > 3 would have to be true before the statement is executed.

Similarly, y = 21 is true after execution of x := 1 if y = 21 is true before its execution, which is a roundabout way of saying that changing x doesn't affect y. However, if x is an alias for y (for example, if x is a formal reference-mode parameter bound to an actual parameter y), the axiom is invalid. In fact, aliasing makes axiomatic definitions much more complex. This is one reason why attempts to limit or ban aliasing are now common in modern language designs (for example, Euclid and Ada).

The axiomatic approach is good for deriving proofs of program correctness, because it avoids implementation details and concentrates on how relations among variables are changed by statement execution. In the assignment axiom, there is no concept of a location in memory being updated; rather, relations among variables are transformed by the assignment. Although axioms can formalize important properties of the semantics of a programming language, it is difficult to use them to define a language completely. For example, they cannot easily model stack overflow or garbage collection.

Denotational semantics is more mathematical in form than operational semantics, yet it still presents the notions of memory access and update that are central to von Neumann languages. Because they rely upon notation and

terminology drawn from mathematics, denotational definitions are often fairly compact, especially in comparison with operational definitions. Denotational techniques have become quite popular, and a definition for all of Ada (excluding concurrency) has been written. Indeed, this definition was the basis for some early Ada compilers, which operated by implementing the denotational representation of a given program.[1] A significant amount of effort in compiler research is directed toward finding automatic ways to convert denotational representations to equivalent representations that map directly to ordinary machine code [Wand 82; Appel 85]. If this effort is successful, a denotational definition (along with lexical and syntactic definitions) may be sufficient to automatically produce a working compiler.

The field of axiomatic semantics was pioneered by C. A. R. Hoare [Hoare 69]. The notation

```
{P} S {R}
```

is a mathematical proposition about the semantics of a program fragment S. It means, "If predicate P is true before program S starts, and program S successfully terminates, then predicate R will be true after S terminates."

The predicates (P and R) typically involve the values of program variables. P is called the **precondition** and R the **postcondition** of the proposition above. The precondition indicates the assumptions that the program may make, and the postcondition represents the result of correct computation. If P and R are chosen properly, such a proposition can mean that S is a **conditionally correct** program, which means it is correct if it terminates.

Relatively strong conditions hold for very few program states; relatively weak conditions hold for very many. The strongest possible condition is `false` (it holds for no program state); the weakest possible condition is `true` (it holds for every program state). A strong proposition is one with a weak precondition or a strong postcondition (or both); thus

```
{true} S {false}
```

is exceptionally strong. In fact, it is so strong that it is true only of nonterminating programs. It says that no matter what holds before S is executed, nothing at all holds afterward. Conversely,

```
{false} S {true}
```

is an exceptionally weak proposition, true of all programs. It says that given unbelievable initial conditions, after S finishes, one can say nothing interesting about the state of variables.

---

[1] The first Ada implementation to take this approach was the NYU Ada/Ed system, infamous for its slowness. Its authors claim this slowness is due primarily to inefficient implementation of certain denotational functions.

2.1 Axioms

The programming language designer can specify the meaning of control structures by stating axioms, such as the axiom of assignment in Figure 10.4.

Figure 10.4

```
Axiom of assignment                                                    1
{P[x → y]} x := y {P}                                                  2
where                                                                  3
      x  is an identifier                                              4
      y  is an expression without side effects, possibly containing x  5
```

This notation says that to prove P after the assignment, one must first prove a related predicate. $P[x \rightarrow y]$ means the predicate P with all references to x replaced by references to y. For instance,

$$\{y < 3 \wedge z < y\}\ x := y\ \{x < 3 \wedge z < y\}$$

is a consequence of this axiom.

In addition to axioms, axiomatic semantics contain **rules of inference**, which specify how to combine axioms to create provable propositions. They have the form:

if X **and** Y **then** Z

That is, if one already knows X and Y, then proposition Z is proven as well. Figure 10.5 shows some obvious rules of inference.

Figure 10.5

```
Rules of consequence                                                   1
if {P} S {R} and R ⇒ Q then {P} S {Q}                                  2
if {P} S {R} and Q ⇒ P then {Q} S {R}                                  3
```

Since $R \Rightarrow S$ means "R is stronger than S," the rules of consequence say that one may always weaken a postcondition or strengthen a precondition. In other words, one may weaken a proposition that is already proven.

The easiest control structure to say anything about is the composition of two statements, as in Figure 10.6.

Figure 10.6

```
Axiom of composition                                                   1
if {P} S1 {Q} and {Q} S2 {R} then {P} S1; S2 {R}                       2
```

Iteration with a **while** loop is also easy to describe, as shown in Figure 10.7.

Figure 10.7

```
Axiom of iteration                                                     1
if {P ∧ B} S {P} then                                                  2
      {P} while B do S end {¬ B ∧ P}                                   3
```

That is, to prove that after the loop B will be false and that P still holds, it suffices to show that each iteration through the loop preserves P, given that B

holds at the outset of the loop. P is called an **invariant** of the loop, because the loop does not cause it to become false.

Figure 10.8 presents an axiom for conditional statements.

Figure 10.8

```
Axiom of condition                                                    1
if {P ∧ B} S {Q} and {P ∧ ¬B} T {Q} then                              2
     {P} if B then S else T end {Q}                                   3
```

2.2 A Simple Proof

I will now use these axioms to prove a simple program correct. The program of Figure 10.9 is intended to find the quotient and remainder obtained by dividing a dividend by a divisor. It is not very efficient.

Figure 10.9

```
remainder := dividend;                                                1
quotient := 0;                                                        2
while divisor ≤ remainder do                                          3
     remainder := remainder - divisor;                                4
     quotient := quotient + 1                                         5
end;                                                                  6
```

I would like the predicate shown in Figure 10.10 to be true at the end of this program.

Figure 10.10

```
{FINAL: remainder < divisor ∧                                         1
     dividend = remainder + (divisor * quotient)}                     2
```

The proposition I must prove is {true} Divide {FINAL}. Figure 10.11 presents a proof.

Figure 10.11

```
true ⇒ dividend = dividend + divisor * 0 [algebra]                    1

{dividend = dividend + divisor*0} remainder := dividend
     {dividend = remainder + divisor*0} [assignment]                  2

{dividend = remainder + divisor*0} quotient := 0
     {dividend = remainder + divisor*quotient} [assignment]           3

{true} remainder := dividend {dividend = remainder+divisor*0}
     [consequence, 1, 2]                                              4

{true} remainder := dividend; quotient := 0
     {dividend = remainder+divisor*quotient}
     [composition, 3, 4]                                              5

dividend = remainder+divisor*quotient ∧ divisor ≤ remainder ⇒
     dividend=(remainder-divisor)+divisor*(1+quotient)
     [algebra]                                                        6
```

```
{dividend=(remainder-divisor)+divisor*(1+quotient)}
      remainder := remainder-divisor
      {dividend=remainder+divisor*(1+quotient)} [assignment]      7

{dividend=remainder+divisor*(1+quotient)}
      quotient := quotient+1
      {dividend=remainder+divisor*quotient} [assignment]          8

{dividend=(remainder-divisor)+divisor*(1+quotient)}
      remainder := remainder-divisor; quotient := quotient+1
      {dividend=remainder+divisor*quotient}
      [composition, 7, 8]                                         9

{dividend = remainder+divisor*quotient ∧ divisor ≤ remainder}
      remainder := remainder-divisor; quotient := quotient+1
      {dividend=remainder+divisor*quotient}
      [consequence, 6, 9]                                         10

{dividend = remainder+divisor*quotient}
      while divisor≤remainder do
            remainder := remainder-divisor;
            quotient := quotient+1
      end
      {remainder < divisor ∧
            dividend=remainder+divisor*quotient}
      [iteration, 10]                                             11

{true} Divide {FINAL} [composition, 5, 11]                        12
```

This style of proof is not very enlightening. It is more instructive to decorate the program with predicates in such a way that an interested reader (or an automated theorem prover) can verify that each statement produces the stated postcondition given the stated precondition. Each loop needs to be decorated with an invariant. Figure 10.12 shows the same program with decorations.

Figure 10.12

```
{true}                                                            1
{dividend = dividend + divisor*0}                                 2
remainder := dividend;                                            3
{dividend = remainder + divisor*0}                                4
quotient := 0;                                                    5
{invariant: dividend = remainder + divisor*quotient}              6
while divisor ≤ remainder do                                      7
      {dividend = (remainder − divisor) +                         8
            divisor * (quotient+1)}                               9
      remainder := remainder - divisor;                           10
      {dividend = remainder + divisor * (quotient + 1)}           11
      quotient := quotient + 1                                    12
      {dividend = remainder + divisor * quotient}                 13
end;                                                              14
{remainder<divisor ∧                                              15
      dividend = remainder + (divisor*quotient)}                  16
```

Unfortunately, the program is erroneous, even though I have managed to prove it correct! What happens if `dividend` = 4 and `divisor` = –2? The **while** loop never terminates. The program is only conditionally, not totally, correct.

The idea of axiomatic semantics has proved fruitful. It has been applied not only to the constructs you have seen, but also to more complex ones such as procedure and function call, **break** from a loop, and even **goto**. Figure 10.13 shows two examples of concurrent programming constructs to which it has been applied [Owicki 76].

Figure 10.13

Parallel execution axiom

if $\forall\, 0 \le i \le n$, $\{P_i\}\ S_i\ \{Q_i\}$,
and no variable free in P_i or Q_i is changed in $S_{j \ne i}$
and all variables in I(r) belong to resource r,
then
$\{P_1 \wedge \ldots \wedge P_n \wedge I(r)\}$
resource r: **cobegin** S_1 // ... // S_n **coend**
$\{Q_1 \wedge \ldots \wedge Q_n\}$

Critical section axiom

if I(r) is the invariant from the **cobegin** statement
and $\{I(r) \wedge P \wedge B\}\ S\ \{I(r) \wedge Q\}$
and no variable free in P or Q is changed in
another thread
then {P} **region** r **await** B **do** S **end** {Q}

The **cobegin** and **region** statements are described in Chapter 7. If the formal axiomatic specification of a construct would suffice to make it intelligible, this example should require no further clarification. However, it may help to point out several facts.

- A **resource** is a set of shared variables.
- The **region** statement may only appear in a **cobegin**.
- **Region** statements for the same resource may not be nested.

The axiomatic method has given rise to an attitude summarized in the following tenets:

1. Programmers should be aware of the propositions that are meant to hold at different stages of the program.
2. The precondition and the postcondition of each whole program should be stated explicitly.
3. Students learning to program should write out the loop invariant explicitly for each loop.
4. Language constructs that do not have simple axioms (such as **goto** and multiple assignment) should not be used.
5. Programmers should prove their programs correct.
6. Proof checkers should be built to assist programmers in proving their programs correct. Such checkers should understand the axioms and enough algebra so that only occasional decorations (such as loop invariants) should be needed.
7. Programmers should develop their programs by starting with the postcondition and working slowly backward, attempting to render it true.

8. Programming languages should allow the programmer to explicitly show loop invariants, preconditions and postconditions to procedure calls, and other decorations, and the compiler should include a proof checker.

This attitude has led to extensive research in programming language design (Alphard and Eiffel were designed with the last point in mind) and automatic theorem provers. However, these tenets are not universally accepted. The strong argument can be made, for example, that program proofs are only as good as the precondition/postcondition specification, and that it is just as easy to introduce a bug in the specifications as it is in the program. For example, a sorting routine might have a postcondition that specifies the result be sorted but might accidentally omit the requirement that the elements be a permutation of the values in the input. Furthermore, it is hard to put much faith in an automated proof that is so complex that no human is willing to follow it.

2.3 Weakest Preconditions

The suggestion that a program can itself be developed by attention to the axiomatic meaning of language constructs and that programmers should develop their programs backward was elucidated by Edsger W. Dijkstra [Dijkstra 75]. Instead of seeing the axioms as static relations between preconditions and postconditions, Dijkstra introduced the concept of weakest precondition. I will say that P = wp(S, Q) if the following statements hold:

- {P} S {Q}. That is, P is a precondition to S.
- S is guaranteed to terminate, given P. That is, S shows **total correctness**, not just conditional correctness.
- If {R} S {Q}, then R ⇒ P. That is, P is the weakest precondition, so {P} S {Q} is the strongest proposition that can be made given S and Q.

Weakest preconditions satisfy several properties:

1. For any statement S, wp(S, false) = false (law of the excluded miracle).
2. If P ⇒ Q, then wp(S, P) ⇒ wp(S, Q) (related to the rules of consequence).
3. wp(S, P) ∧ wp(S, Q) = wp(S, P ∧ Q) (again, related to rules of consequence).

The axioms shown earlier can be restated in terms of wp, as shown in Figure 10.14.

Figure 10.14

Empty statement	1
wp(**skip**, R) = R	2
Assignment statement	3
wp(x := y, R) = R[x → y]	4
Composition	5
wp(S_1, S_2) = wp(S_1, wp(S_2))	6

```
Condition                                                              7
wp(if B then S else T end, R) =                                        8
     B ⇒ wp(S, R) ∧ ¬ B ⇒ wp(T, R)                                     9

Iteration                                                              10
wp(while B do S end, R) =                                              11
     ∃ i ≥ 0 such that H_i(R)                                          12
     where                                                             13
          H_0(R) = R ∧ ¬B                                              14
          H_k(R) = wp(if B then S else skip end,H_{k-1}(R))            15
```

Given these axioms, it is possible to start at the end of the program with the final postcondition and to work backward attempting to prove the initial precondition. With enough ingenuity, it is even possible to design the program in the same order. Let us take a very simple example. I have two integer variables, x and y. I would like to sort them into the two variables a and b. A proposition R that describes the desired result is shown in Figure 10.15.

Figure 10.15

```
R = a ≤ b ∧ ((a = x ∧ b = y) ∨ (a = y ∧ b = x))
```

My task is to find a program P such that wp(P,R) = true; that is, the initial precondition should be trivial. In order to achieve equalities like a = x, I will need to introduce some assignments. But I need two alternative sets of assignments, because I can't force a to be the same as both x and y at once. I will control those assignments by a conditional statement. The entire program P will look like Figure 10.16.

Figure 10.16

```
P = if B then S else T end
```

I will determine B, S, and T shortly. The condition axiom gives me wp(P,R), as in Figure 10.17.

Figure 10.17

```
wp(P,R) = B ⇒ wp(S,R) ∧ ¬ B ⇒ wp(T,R)
```

I will now make a leap of faith and assume that S should contain assignment statements in order to force part of R to be true, as in Figure 10.18.

Figure 10.18

```
S = a := x; b := y;
```

The assignment statement axiom gives me wp(S,R), as shown in Figure 10.19.

Figure 10.19

```
wp(S,R) = x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x))          1
     = x ≤ y                                                   2
```

This equation tells me that statement S alone would almost serve as my program, except that it would have a remaining precondition. A similar set of assignments can force the other part of R to be true, as in Figure 10.20.

Figure 10.20

```
T = a := y; b := x;                                                    1
wp(T,R) = y ≤ x ∧ ((y = x ∧ x = y) ∨ (y = y ∧ x = x))                  2
        = y ≤ x                                                        3
```

I can now combine statements S and T into the conditional statement P, giving me Figure 10.21.

Figure 10.21

```
wp(P,R) = B ⇒ x ≤ y ∧ ¬ B ⇒ y ≤ x
```

I can now choose B to be x < y (it would also work if I chose x ≤ y). This choice allows me to demonstrate that wp(P,R) is true. The entire program P is shown in Figure 10.22.

Figure 10.22

```
if x < y then                                                          1
     a := x;                                                           2
     b := y;                                                           3
else                                                                   4
     a := y;                                                           5
     b := x;                                                           6
end                                                                    7
```

Examples involving loops are even less intuitive. Although the concept of weakest precondition is mathematically elegant, it has not caught on as a tool for programmers.

3 ◆ DENOTATIONAL SEMANTICS

The study of denotational semantics was pioneered by Dana Scott and Christopher Strachey of Oxford University, although many individuals have contributed to its development. A denotational definition is composed of three components: a syntactic domain, a semantic domain, and a number of semantic functions. **Semantic functions** map elementary syntactic objects (for example, numerals or identifiers) directly to their semantic values (integers, files, memory configurations, and so forth). Syntactic structures are defined in terms of the composition of the meanings of their syntactic constituents. This method represents a structured definitional mechanism in which the meaning of a composite structure is a function of the meaning of progressively simpler constituents. As you might guess, unstructured language features (most notably **goto**s) are less easily modeled in a denotational framework than structured features.

The **syntactic domain** contains the elementary tokens of a language as well as an abstract syntax. The syntax specified by a conventional context-free grammar is termed a **concrete syntax** because it specifies the exact syntactic structure of programs as well as their phrase structure. That is, a concrete syntax resolves issues of grouping, operator associativity, and so forth. An **abstract syntax** is used to categorize the kinds of syntactic structures that exist. It need not worry about exact details of program representation or how substructures interact; these issues are handled by the concrete syntax. Thus, in an abstract syntax, an **if** statement might be represented by

```
Stmt → if Expr then Stmt else Stmt
```

without worrying that not all expressions or statements are valid in an **if** statement or that **if** statements are closed by **end** to deal with the dangling-**else** problem.

Semantic domains define the abstract objects a program manipulates. These include integers (the mathematical variety, without size limits), Booleans, and memories (modeled as functions mapping addresses to primitive values). Semantic functions map abstract syntactic structures to corresponding semantic objects. The meaning of a program is the semantic object produced by the appropriate semantic function. For simple expressions, this object might be an integer or real; for more complex programs it is a function mapping input values to output values, or a function mapping memory before execution to memory after execution.

Concrete examples will make this abstract discussion much clearer. I will build a denotational description of a programming language by starting with very simple ideas and enhancing them little by little. As a start, I present in Figure 10.23 the semantics of binary literals—sequences of 0s and 1s. Because syntactic and semantic objects often have a similar representation (for example, 0 can be a binary digit or the integer zero), I will follow the rule that syntactic objects are always enclosed by [and] . The syntactic domain will be named BinLit and defined by abstract syntax rules. The semantic domain will be N, the natural numbers. The semantic function will be named E (for "Expression") and will map binary literals into natural numbers. The symbol | separates alternative right-hand sides in productions.

Figure 10.23

```
Abstract syntax                                          1

    BN ∈ BinLit                                          2

    BN → Seq                                             3
    Seq → 0 | 1 | Seq 0 | Seq 1                          4

Semantic domain                                          5

    N = {0,1,2, ...}                                     6

Semantic function                                        7

    E: BinLit  → N                                       8

    E[0] = 0                                             9
    E[1] = 1                                             10
    E[Seq 0] = 2 × E[Seq]                                11
    E[Seq 1] = 2 × E[Seq] + 1                            12
```

The operators used in the semantic function (× , + , =) are standard integer operators.

I have made a small start at defining the semantics of a programming language. At the heart of each denotational-semantics definition is a set of semantic functions. The meaning of a program is, in general, a function

(usually built up out of other functions) that maps program inputs to program outputs. In the simple example so far, the programming language has no input or output, so the semantic function just takes literals and produces numbers. I will refine this definition until I can describe a significant amount of a programming language.

A summary of all the syntactic and semantic domains and the semantic functions I introduce is at the end of this chapter for quick reference. First, however, I need to introduce some background concepts and notation.

3.1 Domain Definitions

Denotational semantics is careful to specify the exact domains on which semantic functions are defined. This specification is essential to guarantee that only valid programs are ascribed a meaning. I will use the term **domain** to mean a set of values constructed (or defined) in one of the ways discussed below. This careful approach allows me to talk about actual sets and functions as the denotations of syntactic objects while avoiding the paradoxes of set theory.

I will always begin with a set of **basic domains**. For a simple programming language, basic syntactic domains might include: `Op`, the finite domain of operators; `Id`, the identifiers; and `Numeral`, the numerals. Basic semantic domains include `N`, the natural numbers, and `Bool`, the domain of truth values. I can also define finite basic domains by enumeration (that is, by simply listing the elements). For example the finite domain `{true, false}` defines the basic semantic domain of Boolean values. I assume the basic domains are familiar objects whose properties are well understood.

New domains can be defined by applying **domain constructors** to existing domains. I will show three domain constructors corresponding to Cartesian product, disjoint union, and functions. For each constructor, I will show an ML equivalent. All the denotational semantic specifications I will show can be coded (and tested) in ML (discussed in Chapter 3).

3.2 Product Domains

Given domains D_1 and D_2, their **product domain**, $D = D_1 \otimes D_2$, consists of ordered pairs of elements of the component domains. That is,

$$x \in D_1 \otimes D_2 \equiv x = < x_1, x_2 >$$

where $x_1 \in D_1$ and $x_2 \in D_2$.

Product domain D provides two selector functions, Hd_D (the head of a tuple), and Tl_D (the tail). These behave in a fairly natural way, as shown in Figure 10.24.

Figure 10.24

$$Hd_D(< x_1, x_2 >) = x_1 \quad 1$$
$$Tl_D(< x_1, x_2 >) = x_2 \quad 2$$

Again, $x_1 \in D_1$, and $x_2 \in D_2$. I will rarely need to mention these functions explicitly.

The ML equivalent of a product domain is a tuple with two elements. That is, if D_1 and D_2 are ML types, then the product type $D = D_1 \otimes D_2$ is just $D_1 * D_2$. Instead of selectors, I will use patterns to extract components. The tuple constructor will serve as a domain constructor.

3.3 Disjoint-Union Domains

Let D_1 and D_2 be domains. Their **disjoint union**, $D_1 \oplus D_2$, consists of elements of either D_1 or D_2, where each value carries with it an indication of which domain it came from. Formally, the elements of $D = D_1 \oplus D_2$ are

$$\{ < 1, x_1 > \mid x_1 \in D_1 \} \cup \{ < 2, x_2 > \mid x_2 \in D_2 \} \ .$$

Disjoint-union domain D provides two injection functions, $\mathtt{In}D_1$ and $\mathtt{In}D_2$, as in Figure 10.25.

Figure 10.25

$$\mathtt{In}D_1(x_1) = < 1, x_1 > \qquad 1$$
$$\mathtt{In}D_2(x_2) = < 2, x_2 > \qquad 2$$

As usual, $x_1 \in D_1$, and $x_2 \in D_2$.

This form of disjoint union may seem unnecessarily complicated, but it has the advantage that the meaning of $D_1 \oplus D_2$ is independent of whether D_1 and D_2 are disjoint. For example, such obvious properties as

$$\forall x_1 \in D_1 \ \forall x_2 \in D_2 \ . \ \mathtt{In}D_1(x_1) \neq \mathtt{In}D_2(x_2)$$

remain true even if $D_1 = D_2$.

The ML equivalent of a disjoint union is a **datatype**. That is, if D_1 and D_2 are ML types, then Figure 10.26 shows the disjoint-union type $D = D_1 \oplus D_2$.

Figure 10.26

```
datatype D                                  1
    = FirstComponent of D1                  2
    | SecondComponent of D2;                3
```

The ML notation allows me to introduce names for the two components, which will be helpful in testing from which underlying domain a member of the disjoint-union domain comes.

3.4 Function Domains

Given domains D_1 and D_2, their **function domain** $D_1 \to D_2$ is a set of functions mapping elements of D_1 to elements of D_2. For technical reasons, $D_1 \to D_2$ means not all functions from D_1 to D_2, but rather a subset of them, called the continuous ones. Every computable function (hence every function I will need) is continuous. If $f \in D_1 \to D_2$ and $x_1 \in D_1$, the application of f to x_1, written $f(x_1)$ or $f\ x_1$, is an element of D_2.

There are several ways to package multiple parameters to a function. Just as in ML, they can be packaged into a single tuple or curried. Function values can be parameters or returned results, just like values of any other

type.

I will need notation for a few simple functions. First, there is the constant function. For example,

$$f(x{:}D) = 17$$

denotes the function in $D \to N$ that always produces the value 17.

Second, I will need a function that differs from an existing function on only a single parameter value. Suppose $f \in D_1 \to D_2$, $x_1 \in D_1$, and $x_2 \in D_2$. Then

$$f[x_1 \leftarrow x_2]$$

denotes the function that differs from f only by producing result x_2 on parameter x_1; that is:

$$f[x_1 \leftarrow x_2]\, y = \textbf{if } y = x_1 \textbf{ then } x_2 \textbf{ else } f\ y$$

This simple device allows me to build up all almost-everywhere-constant functions—functions that return the same result on all but finitely many distinct parameter values. This mechanism is particularly useful in modeling declarations and memory updates.

3.5 Domain Equations

I will define a collection of domains $D_1, \ldots, D_k$ by a system of formal equations, as in Figure 10.27.

Figure 10.27

$$\begin{array}{ll} D_1 = rhs_1 & 1 \\ \quad \ldots & 2 \\ D_k = rhs_k & 3 \end{array}$$

Each right-hand side rhs_i is a domain expression, built from basic domains (and possibly from some of the D_i themselves) using the domain constructors given above.

For technical reasons, it is important that I not treat these formal equations as meaning strict equality. Instead, I use a somewhat more liberal interpretation. I say that domains $D_1, \ldots, D_k$ comprise a solution to the above system of domain equations if, for each i, D_i is isomorphic to the domain denoted by rhs_i; that is, there exists a one-to-one, onto function between them.

While I have not shown that this liberal interpretation of domain equations is technically necessary, you can certainly appreciate its convenience. Consider the single equation:

$$D = N \oplus Bool\ .$$

Intuitively, the set $N \cup Bool$ has all the properties required of a solution to this equation. The right-hand side of this equation denotes

$$N \oplus Bool \equiv \{ < 1, x > \mid x \in N\} \cup \{ < 2, y > \mid y \in Bool\}$$

which is clearly not equal to $N \cup Bool$. However, it is easy to see that the two sets are isomorphic, since N and $Bool$ are disjoint, so by the liberal interpretation of equations as isomorphisms, $N \cup Bool$ is a solution to the equation. Thus, as intuition suggests, if D_1 and D_2 are disjoint domains, no confusion results from taking $D_1 \oplus D_2$ to be $D_1 \cup D_2$ rather than using the full mechanism of the disjoint-union domain constructor.

3.6 Nonrecursive Definitions

I need to introduce just a bit more terminology. In a system of domain equations, each right-hand side is a domain expression, consisting of applications of domain constructors to basic domains and possibly to some of the domains D_i being defined by the system of equations. A right-hand side that uses no D_i, that is, one that consists entirely of applications of domain constructors to basic domains, is **closed**. A right-hand side rhs that is not closed has at least one use of a D_i; I will say that D_i **occurs free** in rhs. For example, in

$$D_{17} = D_{11} \oplus (D_{11} \otimes N) \oplus (D_{12} \otimes N)$$

rhs_{17} has two free occurrences of the name D_{11} and one free occurrence of the name D_{12}; no other names occur free in rhs_{17}.

A system S of domain equations is nonrecursive if it can be ordered as in Figure 10.28,

Figure 10.28

$$\begin{array}{ll} D_1 = rhs_1 & 1 \\ \quad \ldots & 2 \\ D_k = rhs_k & 3 \end{array}$$

where only the names $D_1, \ldots, D_{i-1}$ are allowed to appear free in rhs_i. In particular, this definition implies that rhs_1 is closed.

A solution to a nonrecursive system of domain equations S can be found easily by a process of repeated back substitution, as follows. Begin with the system S, in which rhs_1 is closed. Build a new system S^2 from S by substituting rhs_1 for every occurrence of the name D_1 in the right-hand sides of S. You should convince yourself of the following:

1. S^2 has no free occurrences of D_1.
2. S^2 is equivalent to S in the sense that every solution to S^2 is a solution to S, and conversely.
3. Both rhs_1^2 and rhs_2^2 are closed.

Now build system S^3 from S^2 by substituting rhs_2^2 for every occurrence of D_2 in the right-hand sides of S^2 . Just as above, the following hold:

1. S^3 has no free occurrences of D_1 or D_2.
2. S^3 is equivalent to S^2 (and hence to S).
3. All of rhs_1^3, rhs_2^3, and rhs_3^3 are closed.

The pattern should now be clear: Repeat the substitution step to produce S^k, in which all of $rhs_1, \ldots, rhs_k$ are closed. There is an obvious solution to S^k: Evaluate all the right-hand sides.

A simple example may help. Let S be the nonrecursive system shown in Figure 10.29.

Figure 10.29

$$D_1 = N \otimes N \quad 1$$
$$D_2 = N \oplus D_1 \quad 2$$
$$D_3 = D_1 \otimes D_2 \quad 3$$

Then S^2 is given in Figure 10.30.

Figure 10.30

$$D_1 = N \otimes N \quad 1$$
$$D_2 = N \oplus (N \otimes N) \quad 2$$
$$D_3 = (N \otimes N) \otimes D_2 \quad 3$$

Finally S^3 is given in Figure 10.31.

Figure 10.31

$$D_1 = N \otimes N \quad 1$$
$$D_2 = N \oplus (N \otimes N) \quad 2$$
$$D_3 = (N \otimes N) \otimes (N \oplus (N \otimes N)) \quad 3$$

Now all right-hand sides are closed.

3.7 Recursive Definitions

A system of domain equations is recursive if no matter how it is ordered there is at least one i such that rhs_i contains a free occurrence of D_j for some $j \geq i$. That is, a system is recursive if it cannot be reordered to eliminate forward references. Intuitively, such a system is an inherently circular definition.

BNF definitions for syntax can be recursive as well. The context-free grammar descriptions of typical programming languages routinely contain recursive production rules like:

```
Expr → Expr op Expr
```

Intuitively, this rule states that an expression can be built by applying an operator to two subexpressions. A recursive collection of grammar rules defines the set of all objects that can be constructed by finitely many applications of the rules. Such recursive rules are indispensable; they are the only way a finite set of context-free production rules can describe the infinite set of all valid programs. Similarly, if you try to define semantics with only nonrecursive domain equations, you will soon discover they are not powerful enough.

Unfortunately, interpreting a recursive system of domain equations can be subtle. In an ML representation of domain equations, I will just declare the equations with the **rec** modifier, so that they can depend on each other. I will ignore any problems that circularity might raise. But consider the innocuous-looking equation of Figure 10.32.

Figure 10.32

$$D = N \oplus (N \otimes D)$$

Interpreting this equation as if it were a production, you might conclude that the domain D consists of (or is isomorphic to) the set of all nonempty finite sequences of elements of N. However, the set D′ of all sequences (finite or infinite) over N is also a solution to Figure 10.32, since every (finite or infinite) sequence over N is either a singleton or an element of N followed by a (finite or infinite) sequence.

Where there are two solutions, it makes sense to look for a third. Consider the set of all (finite or infinite) sequences over N in which 17 does not occur infinitely often. This too is a solution. This observation opens the floodgates. Rather than 17, I can exclude the infinite repetition of any finite or infinite subset of N to get yet another solution to Figure 10.32—for example, the set of all sequences over N in which no prime occurs infinitely often.

By this simple argument, the number of distinct solutions to Figure 10.32 is at least as big as 2^N —the power set, or set of all subsets, of N. Which solution to Figure 10.32 is the right one? The one I want is the one that corresponds to a BNF-grammar interpretation—the set of finite sequences.

Any solution to Figure 10.32 need only satisfy the equation up to isomorphism; but I will find an exact solution. From Figure 10.32 I can determine the (infinite) set of all closed expressions denoting elements of D. A few of these are shown in Figure 10.33.

Figure 10.33

<1,0>	1
<1,1>	2
<1,2>	3
<1,3>	4
...	5
<2,(0 ⊗ <1,0>>	6
<2,(1 ⊗ <1,0>>	7
<2,(2 ⊗ <1,0>>	8
<2,(3 ⊗ <1,0>>	9
...	10
<2,(0 ⊗ <1,1>>	11
<2,(0 ⊗ <1,2>>	12
<2,(0 ⊗ <1,3>>	13
...	14

The (infinite) set of the values of these expressions yields an exact solution to Figure 10.32. It can also be shown that this is the smallest solution, in that it is isomorphic to a subset of any other solution. In general, the solution to prefer when there are many possible solutions to a recursive system of domain equations is the smallest one.

Equations of the form of Figure 10.32 arise so frequently that their solutions have a notation: If D is already defined, then the solution to

$$D' = D \oplus (D \otimes D')$$

is called D^*.

Function domains cause problems in recursive systems of domain equations. Even a simple recursive equation like

$$D = \cdots \oplus (D \to D) \oplus \cdots$$

is suspect. Any solution to this equation would have the property that some subset of itself was isomorphic to its own function space. Unfortunately, if a set has more than one element, then the cardinality of its function space is strictly greater than the cardinality of the set itself, so no such isomorphism is possible!

Am I stuck? Not really. As mentioned above, I interpret $D \to D$ to mean not all functions from D to D, but just a distinguished set of functions called the continuous ones. There are sufficiently few continuous functions that the above cardinality argument does not apply, but sufficiently many of them that all functions computable by programming languages are continuous.

3.8 Expressions

Now that I have discussed domains, I can begin to create richer and more realistic semantic functions. I first extend my definition of binary literals to include infix operators; see Figure 10.34.

Figure 10.34

Abstract syntax 1

T ∈ Exp 2

T → T + T 3
T → T - T 4
T → T * T 5
T → Seq 6
Seq → 0 | 1 | Seq 0 | Seq 1 7

Semantic domain 8

N = {0,1,2, ..., -1, -2, ...} 9

Semantic function 10

E: Exp → N 11

E[0] = 0 12
E[1] = 1 13
E[Seq 0] = 2 × E[Seq] 14
E[Seq 1] = 2 × E[Seq] + 1 15

$E[T_1 + T_2] = E[T_1] + E[T_2]$ 16
$E[T_1 - T_2] = E[T_1] - E[T_2]$ 17
$E[T_1 * T_2] = E[T_1] \times E[T_2]$ 18

This example can be specified in ML as shown in Figure 10.35.

Figure 10.35

```
-- abstract syntax                                                    1

     datatype Operator = plus | minus | times;                        2
     datatype Exp                                                     3
          = BinLit of int list -- [0,1] means 10 = 2                  4
          | Term of Exp*Operator*Exp;                                 5

-- semantic functions                                                 6

     val rec E =                                                      7
          fn BinLit([0]) => 0                                         8
           | BinLit([1]) => 1                                         9
           | BinLit(0 :: tail) => 2*E(BinLit(tail))                   10
           | BinLit(1 :: tail) => 1+2*E(BinLit(tail))                 11
           | Term(x, plus, y) => E(x) + E(y)                          12
           | Term(x, minus, y) => E(x) - E(y)                         13
           | Term(x, times, y) => E(x) * E(y);                        14
```

Because it is easier to access the front of a list than the rear, I chose to let BinLits (line 4) store least-significant bits at the front of the list. A benefit of the ML description is that it can be given to an ML interpreter to check. For instance, I have checked the code shown in Figure 10.36.

Figure 10.36

```
in:  E(Term(BinLit([1,1]), plus,                                      1
                BinLit([0,1]))); -- 3 + 2                             2
out: 5 : int                                                          3
```

To include division, I must define what division by zero means. To do so, I augment the semantic domain with an error element, $\perp$. That is, I now have a domain of $R = N \oplus \{\perp\}$, where R represents "results." Because this is a disjoint-union domain, I can test which subdomain a given semantic element belongs to. I use the notation v?D to test if value v is in domain D. I also will use the following concise conditional-expression notation:

$b \Rightarrow x,y$ means **if** b **then** x **else** y

Errors must propagate through arithmetic operations, so I need to upgrade the semantic functions. Figure 10.37 presents the denotation of expressions with division.

Figure 10.37

Semantic domain 1

$R = N \oplus \{\perp\}$ 2

```
Semantic function                                                          3

     E: Exp → R                                                            4

     E[0] = 0                                                              5
     E[1] = 1                                                              6
     E[Seq 0] = 2 × E[Seq]                                                 7
     E[Seq 1] = 2 × E[Seq] + 1                                             8

     E[T1 + T2] = E[T1]?N ∧ E[T2]?N ⇒ E[T1] + E[T2], ⊥                     9
     E[T1 − T2] = E[T1]?N ∧ E[T2]?N ⇒ E[T1] − E[T2], ⊥                     10
     E[T1 * T2] = E[T1]?N ∧ E[T2]?N ⇒ E[T1] × E[T2], ⊥                     11
     E[T1 / T2] = E[T1]?N ∧ E[T2]?N ⇒ (E[T2] = 0 ⇒ ⊥, E[T1] / E[T2]), ⊥    12
```

This definition is unrealistic in that it ignores the finite range of computer arithmetic. Since I have an error value, I can use it to represent range errors. I will introduce a function range such that:

```
range: N → {minInt..maxInt} ⊕ {⊥}.                                         1
range(n) = minInt ≤ n ≤ maxInt ⇒ n, ⊥                                      2
```

Figure 10.38 shows how to insert Range into the definition of E.

Figure 10.38

```
Semantic function                                                          1

     E: Exp → R                                                            2

     E[0] = 0                                                              3
     E[1] = 1                                                              4
     E[Seq 0] = E[Seq]?N ⇒ range(2 × E[Seq]), ⊥                            5
     E[Seq 1] = E[Seq]?N ⇒ range(2 × E[Seq] + 1), ⊥                        6

     E[T1 + T2] = E[T1]?N ∧ E[T2]?N ⇒ range(E[T1] + E[T2]), ⊥              7
     E[T1 − T2] = E[T1]?N ∧ E[T2]? ⇒ range(E[T1] − E[T2]), ⊥               8
     E[T1 * T2] = E[T1]?N ∧ E[T2]?N ⇒ range(E[T1] × E[T2]), ⊥              9
     E[T1/T2] = E[T1]?N ∧ E[T2]?N ⇒                                        10
          (E[T2] = 0 ⇒ ⊥, range(E[T1] / E[T2])), ⊥                         11
```

It is time to show the ML equivalent, given in Figure 10.39.

Figure 10.39

```
-- tools                                                                   1

     val SayError = fn (str, result) => -- report error                    2
          (    output(std_out, str);                                       3
               result -- returned                                          4
          );                                                               5

-- limits                                                                  6
     val MaxInt = 1000; -- or whatever                                     7
     val MinInt = -1000; -- or whatever                                    8
```

```
-- abstract syntax                                                    9

     datatype Operator = plus | minus | times | divide;              10
     datatype Exp                                                    11
          = BinLit of int list -- [0,1] means 10 = 2                 12
          | Term of Exp*Operator*Exp;                                13

-- semantic domains                                                  14

     datatype R                                                      15
          = NaturalR of int                                          16
          | ErrorR;                                                  17

-- semantic functions                                                18

     val Range =                                                     19
          fn NaturalR(a) =>                                          20
               if MinInt ≤ a and a ≤ MaxInt then                     21
                    NaturalR(a)                                      22
               else                                                  23
                    SayError("overflow", ErrorR)                     24
          | _ => ErrorR;                                             25

     val Add =                                                       26
          fn (NaturalR(a),NaturalR(b)) => NaturalR(a+b);             27
     val Sub =                                                       28
          fn (NaturalR(a),NaturalR(b)) => NaturalR(a-b);             29
     val Mul =                                                       30
          fn (NaturalR(a),NaturalR(b)) => NaturalR(a*b);             31
     val Div =                                                       32
          fn (NaturalR(a),NaturalR(0)) =>                            33
               SayError("Divide by zero", ErrorR)                    34
           | (NaturalR(a),NaturalR(b)) =>                            35
               NaturalR(floor(real(a)/real(b)));                     36

     val rec E =                                                     37
          fn BinLit([0]) => NaturalR(0)                              38
           | BinLit([1]) => NaturalR(1)                              39
           | BinLit(0 :: tail) =>                                    40
               let val NaturalR(num) = E(BinLit(tail))               41
               in  NaturalR(2*num)                                   42
               end                                                   43
           | BinLit(1 :: tail) =>                                    44
               let val NaturalR(num) = E(BinLit(tail))               45
               in  NaturalR(2*num + 1)                               46
               end                                                   47
           | Term(x, plus, y) => Range(Add(E(x), E(y)))              48
           | Term(x, minus, y) => Range(Sub(E(x), E(y)))             49
           | Term(x, times, y) => Range(Mul(E(x), E(y)))             50
           | Term(x, divide, y) => Range(Div(E(x), E(y)));           51
```

I have introduced an error routine SayError (lines 2–5) so that a user can see exactly what sort of error has occurred instead of just getting a result of ⊥. The Range function (lines 19–25) not only checks ranges, but also makes sure

that its parameter is a natural number. I have split out Add, Sub, Mul, and Div (lines 26–36), so that they can check the types of their parameters. I could have given them alternatives that return ⊥ if the types are not right. The semantic function E (lines 37–51) needs to convert parameters of type BinLit to results of type R.

Any realistic programming language will have more than one type, which I illustrate by adding the semantic domain Bool corresponding to Booleans. I also add the comparison operator = that can compare two integers or two Booleans. The additions I need to upgrade Figure 10.38 are given in Figure 10.40.

Figure 10.40

```
Abstract syntax                                                          1

    T → T = T                                                            2

Semantic domain                                                          3

    R = N ⊕ Bool ⊕ {⊥}                                                   4

Semantic function                                                        5

    E[T1 = T2] = (E[T1]?N ∧ E[T2]?N) ∨ (E[T1]?Bool ∧ E[T2]?Bool) ⇒        6
        (E[T1] = E[T2]), ⊥                                               7
```

3.9 Identifiers

I can now introduce predeclared identifiers, including true and false, maxint, minint, and so forth. Let Id be the syntactic domain of identifiers, and let L be a semantic lookup function such that L: Id → V, where V = N ⊕ Bool ⊕ {udef}. That is, L returns an integer or Boolean value, or udef if the identifier is undefined. The additions needed for Figure 10.40 are given in Figure 10.41.

Figure 10.41

```
Abstract syntax                                                          1

    I ∈ Id                                                               2
    T → I                                                                3

Semantic domains                                                         4

    V = N ⊕ Bool ⊕ {udef}                                                5

Semantic functions                                                       6

    L: Id → V                                                            7

    E[I] = L[I]?{udef} ⇒ ⊥, L[I]                                         8
```

3.10 Environments

The next step is to introduce programmer-defined named constants. This step requires the concept of an environment that is updated when declarations are made. An **environment** is a function that maps identifiers (drawn from the syntactic domain) into results. I will denote the domain of environments as U, where $U = Id \to V$ and $V = N \oplus Bool \oplus \{udef\} \oplus \{\perp\}$, as in Figure 10.42. If $u \in U$ and $I \in Id$, then u[I] is an integer, Boolean, udef, or $\perp$, depending on how and whether I has been declared. I can incorporate the definition of predeclared named constants by including them in u_0, a predefined environment. I no longer need the lookup function L.

Figure 10.42

Semantic domain 1

$V = N \oplus Bool \oplus \{udef\} \oplus \{\perp\}$ 2
$U = Id \to V$ 3

Semantic functions 4

$E[I] = u_0[I]?\{udef\} \Rightarrow \perp, u_0[I]$ 5

The environment approach is useful because environments can be computed as the results of semantic functions (those that define the meaning of a local constant declaration).

It is time to expand my abstract syntax for a program into a sequence of declarations followed by an expression that yields the result of a program. I can specify whatever I like for the meaning of a redefinition of an identifier. In Figure 10.43, redefinitions will have no effect.

I will introduce two new semantic functions: D, which defines the semantic effect of declarations, and M, which defines the meaning of a program. D is curried; it maps a declaration and an old environment into a new environment in two steps. There is a major change to E; it now maps an expression and an environment into a result. Pr is the syntactic domain of all programs; Decls is the syntactic domain of declarations.

Figure 10.43

Abstract syntax 1

```
P ∈ Pr -- a program                                    2
T ∈ Exp -- an expression                               3
I ∈ Id -- an identifier                                4
Def ∈ Decls -- a declaration                           5

P → Def T                                              6
Def → ε -- empty declaration                           7
Def → I = T ; -- constant declaration                  8
Def → Def Def -- declaration list                      9
T → I -- identifier expression                         10
```

Semantic domains 11

```
R = N ⊕ Bool ⊕ {⊥}  -- program results                          12
V = N ⊕ Bool ⊕ {udef} ⊕ {⊥}  -- lookup values                   13
U = Id → V  -- environments                                     14
```

Semantic functions 15

```
E: Exp → U → R                                                  16
D: Decls → U → U                                                17
M: Pr → R                                                       18

M[Def T] = E[T]u                                                19
      where u = D[Def]u₀.                                       20

D[ε]u = u                                                       21
D[I = T]u = u[I]?{udef} ⇒ u[I ← e], u                           22
      where e = E[T]u.                                          23
D[Def₁ Def₂] u = D[Def₂]v                                       24
      where v = D[Def₁]u.                                       25

E[I] = u[I] ?{udef} ⇒ ⊥, u[I]                                   26
E[0]u = 0                                                       27
E[1]u = 1                                                       28
E[Seq 0] u = E[Seq]u?N ⇒ range(2 × E[Seq]u), ⊥                   29
E[Seq 1] u = E[Seq]u?N ⇒ range(2 × E[Seq]u + 1), ⊥               30

E[T₁ + T₂] u = E[T₁]u?N ∧ E[T₂]u?N ⇒                             31
      range(E[T₁] u + E[T₂]u), ⊥                                32
E[T₁ − T₂] u = E[T₁]u?N ∧ E[T₂]u?N ⇒                             33
      range(E[T₁] u − E[T₂]u), ⊥                                34
E[T₁ * T₂] u = E[T₁]u?N ∧ E[T₂]u?N ⇒                             35
      range(E[T₁] u × E[T₂]u), ⊥                                36
E[T₁ / T₂] u = E[T₁]u?N ∧ E[T₂]u?N ⇒                             37
      (E[T₂] u = 0 ⇒ ⊥, range(E[T₁] u / E[T₂]u)), ⊥             38
E[T₁ = T₂]u = (E[T₁]u?N ∧ E[T₂]u?N) ∨ (E[T₁]u?Bool ∧ E[T₂]u?Bool) ⇒  39
      (E[T₁]u = E[T₂]u), ⊥                                      40
```

Lines 19–20 define the meaning of a program to be the value of the expression T in the environment u formed by modifying the initial environment u_0 by the declarations. Lines 22–23 show how declarations modify a given environment u by substituting the meaning of T for the identifier I in u. Multiple declarations build the final environment in stages (lines 24–25).

Line 22 explicitly ignores attempts to redefine an identifier, but I can make the language a bit more realistic. I will let a redefinition of an identifier return an environment in which the identifier is bound to a new kind of error value named redef. E of a redefined identifier will yield ⊥. I extend the domain V of possible environment values to include redef. Figure 10.44 shows the differences.

Figure 10.44

```
Semantic domains                                                    1

    V = N ⊕ Bool ⊕ {⊥} ⊕ {udef} ⊕{redef}                             2

Semantic functions                                                  3

    D[I = T]u = u[I]?{udef} ⇒ u[I ← e], u[I ← redef]                4
          where e = E[T]u.                                          5
    E[I]u = u[I]?({udef} ⊕ {redef}) ⇒ ⊥, u[I]                       6
```

At this point I could add block structure, but since programs only compute a single expression, scoping isn't needed yet. Instead, I will put in variables.

3.11 Variables

I can model variables in several ways. The most general model employs an environment that maps identifiers to locations and a store that maps locations to values. This is how most languages are implemented, and it would allow me to model aliasing, reuse of storage, and so forth.

For the present, I'll use a simpler interpreter model and continue to use the environment function to map identifiers directly to values. I will also store a flag that indicates if a value can be changed (that is, if it's an L-value, not an R-value). An interpreter does roughly the same thing, maintaining a runtime symbol table for all program variables. From the semantic point of view, the distinction between interpreters and compilers is irrelevant—what is important is what the answer is, not how it's produced. The interpreter approach will allow interesting variations. For example, an untyped language (like Smalltalk) is just as easy to model as a strongly typed language.

I begin by extending the environment domain U as in Figure 10.45 to include an indication of how an identifier can be used:

```
U = Id → {var, const, uninit} ⊗ V
```

`Uninit` models the fact that after a variable is declared, it may be assigned to, but not yet used. After a variable is assigned a value, its flag changes from `uninit` to `var`. It is time to introduce statements. (In denotational formalisms, statements are usually called commands.) A statement maps an environment into a new environment (or ⊥). That is,

```
S: Stm → U → (U ⊕ {⊥})
```

where S is the semantic function for statements, and `Stm` is the syntactic domain of statements.

I will first add only variable declarations and assignment statements to the programming language. Since there is no I/O, I will define the result of the program to be the final value of an identifier that is mentioned in the program header, as in Figure 10.45, which produces 1 as its result.

Figure 10.45

```
program(x)                                                    1
     x : integer;                                             2
     x := 1;                                                  3
end                                                           4
```

To simplify the definitions, I will use $\wedge$ and $\vee$ as short-circuit operators: Only those operands needed to determine the truth of an expression will be evaluated. Thus,

$$e?N \wedge e > 0$$

is well defined even if e is a Boolean, in which case $e > 0$ is undefined.

Further, if some $e \in D$, where $D = (D_1 \otimes D_2) \oplus D_3$, and D_3 isn't a product domain, then $Hd(e)?D_1$ will be considered well defined (with the value false) if $e \in D_3$. That is, if e isn't in a product domain, I will allow Hd(e) or Tl(e) to be used in a domain test. This sloppiness should cause no confusion, since if e isn't in a product domain, then Hd(e) or Tl(e) isn't in any domain. Use of Hd(e) or Tl(e) in other than a domain test is invalid if e isn't in a product domain.

Figure 10.46 presents a new language specification, building on the one in Figure 10.43 (page 330).

Figure 10.46

```
Abstract syntax                                                1

     P ∈ Pr -- a program                                       2
     T ∈ Exp -- an expression                                  3
     I ∈ Id -- an identifier                                   4
     Def ∈ Decls -- a declaration                              5
     St ∈ Stm -- a statement                                   6

     P → program (I) Def St end -- program                     7
     Def →ε -- empty declaration                               8
     Def → I = T; -- constant declaration                      9
     Def → I : integer; -- integer variable declaration        10
     Def → I : Boolean; -- Boolean variable declaration        11
     Def → Def Def -- declaration list                         12

     St →ε -- empty statement                                  13
     St → I := T -- assignment statement                       14
     St → St St -- statement list                              15

Semantic domains                                               16

     R = N ⊕ Bool ⊕ {⊥} -- program results                     17
     V = N ⊕ Bool ⊕ {⊥} ⊕ {udef} ⊕{redef} -- id value          18
     U = Id → {var, const, uninit} ⊗ V -- environments         19
```

Semantic functions 20

E: Exp → U → R 21
D: Decls → U → U 22
M: Pr → R 23
S: Stm → U → (U ⊕ {⊥}) 24

M[**program** (I) Def St **end**] = c?U ⇒ E[I]c, ⊥ 25
  where u = D[Def]u$_0$; c = S[St]u. 26
D[ε]u = u 27
D[I = T]u = u[I]?{udef} ⇒ u[I ← f], u[I ← redef] 28
  where e = E[T]u; f = e?⊥ ⇒ ⊥, < const, e > . 29
D[I: integer]u = u[I]?{udef} ⇒ u[I ← e], u[I ← redef] 30
  where e = <uninit, InN(0)> 31
D[I: Boolean]u = u[I]?{udef} ⇒ u[I ← e], u[I ← redef] 32
  where e = <uninit, InBool(true)>. 33
D[Def$_1$ Def$_2$]u = D[Def$_2$]v 34
  where v = D[Def$_1$]u. 35
E[I]u = v?({redef} ⊕ {udef} ⊕ {⊥}) ⇒ ⊥, 36
  (Hd(v) = uninit ⇒ ⊥, Tl(v)) 37
  where v = u[I]. 38
S[ε]u = u 39
S[I: = T]u = v?({redef}⊕{udef}⊕{⊥}) ∨ (Hd(v) = const) ∨ e?{⊥} ⇒ 40
  ⊥, (e?N ∧ Tl(v)?N) ∨ (e?Bool ∧ Tl(v)?Bool) ⇒ 41
    u[I ← < var, e >], ⊥ 42
  where e = E[T]u; v = u[I]. 43
S[St$_1$ St$_2$] u = g?U ⇒ S[St$_2$]g, ⊥ 44
  where g = S[St$_1$]u. 45

The easiest way to read the semantic functions is to first look at the **where** clauses to see the local shorthands. (These are like ML **let** blocks.) Then look at the definition itself, following the case where no errors are encountered. Much of each definition necessarily deals with checking for error situations, which tend to confuse the central issue. When I describe definitions, I will generally ignore all the error cases and concentrate on the usual case. Lastly, assure yourself that the functions are given parameters of the correct domains and produce results in the correct domains.

For example, lines 40–43 describe what an assignment does to the environment u. Start with line 43. The local variable e stands for the value of the right-hand side of the assignment in environment u, and v stands for the meaning of the identifier on the left-hand side. This meaning is evaluated in the same environment u, so if evaluating the right-hand side had a side effect (it can't yet, but it might later), that effect is ignored in determining the identifier's meaning. Then (line 40) given that v is properly declared and not a constant, given that e evaluates successfully, and given (line 41) that the expression and identifier are the same type, the statement creates a new environment (line 42) that is just like the old one with the identifier reassigned.

To check domain consistency, I will ignore all error cases and write out these few lines again in Figure 10.47.

Figure 10.47

```
S[I:= T]u = u[I ← < var, e >]                                    1
      where e = E[T]u.                                           2
```

Now I can painstakingly infer the type of S, as shown in Figure 10.48.

Figure 10.48

```
E: Exp → U → R                                                   1
E[T]: U → R                                                      2
e = E[T]u: R                                                     3
e: V, since V is a superset of R                                 4

u: U                                                             5
u: Id → {var, const, uninit} ⊗ V                                 6
u[I]: {var, const, uninit} ⊗ V                                   7

< var, e > : {var, const, uninit} ⊗ V                            8
u[I ← < var, e >]: U                                             9
u[I ← < var, e >]: U ⊕ {⊥}, which is a superset of U             10

S: Stm → U → (U ⊕ {⊥})                                           11
S[I:= T]: U → (U ⊕ {⊥})                                          12
S[I:= T]u: U ⊕ {⊥}                                               13
```

Line 10 shows the type of the right-hand side of the equation in line 1, and line 13 shows the type of the left-hand side. They match. It was necessary to raise several types; see lines 4 and 10. If this example were coded in ML, I would need to use explicit type converters.

Other notes on Figure 10.46: The value of 0 in line 31 is arbitrary since I don't allow access to variables with an `uninit` flag. In the definition of statement execution (lines 43–44), as soon as a statement yields ⊥, all further statement execution is abandoned.

As I suggested earlier, my definitions can easily be modified to handle untyped languages like Smalltalk. I would of course modify the variable-declaration syntax to omit the type specification. A variable would assume the type of the object assigned to it. The definitions of E and S would be written as in Figure 10.49.

Figure 10.49

```
E[I]u = v?({redef} ⊕ {udef} ⊕ {⊥}) ⇒ ⊥, Tl(v)                    1
      where v = u[I].                                            2

S[I:= T]u = v?({redef}⊕{⊥}) ∨ (Hd(v) = const) ∨ e?{⊥} ⇒           3
      ⊥, u[I ← < var, e >]                                       4
      where e = E[T]u; v = u[I].                                 5
```

3.12 Conditional and Iterative Statements

Conditional execution and iterative execution for a fixed number of iterations are readily modeled with the additions to the previous definition shown in Figure 10.50.

Figure 10.50

```
Abstract syntax                                                          1

    St → if T then St else St                                            2
    St → do T times St                                                   3

Semantic functions                                                       4

    S[if T then St1 else St2]u = e?Bool ⇒                                5
        (e ⇒ S[St1]u, S[St2]u), ⊥                                        6
        where e = E[T]u.                                                 7

    S[do T times St]u = e?N ⇒ vm , ⊥                                     8
        where e = E[T]u; m = max(0, e); v0 = u;                          9
        vi+1 = vi?U ⇒ S[St]vi, ⊥.                                        10
```

In lines 8–10, v_i is the environment after `i` iterations of the loop.

The semantic definition of a **while** loop requires special care. The problem is that some **while** loops will never terminate, and I would like a mathematically sound definition of all loops. I might try to build on the definition for the **do** loop, but for nonterminating loops that would create an infinite sequence of intermediate environments (v_i's).

I will follow standard mathematical practice for dealing with infinite sequences and try to determine if a limit exists. I will then be able to conclude that infinite loops have a value of $\perp$, though the semantic function for **while** loops will not always be computable (because of decidability issues). Following Tennent, I will define a sequence of approximations to the meaning of a **while** loop [Tennent 81].

Let $p_0 \equiv \perp$. This formula represents a **while** loop whose Boolean expression has been tested zero times. Since a loop can't terminate until its Boolean expression has evaluated to `false`, p_0 represents the base state in which the definition hasn't yet established termination. Now I define p_{i+1} recursively, as in Figure 10.51.

Figure 10.51

```
p_{i+1}(u) = e?Bool ⇒ (e ⇒ (v?{⊥}⇒⊥, p_i(v)), u), ⊥                       1
    where e = E[T]u; v = S[St]u.                                         2
```

If a **while** loop terminates without error after exactly one evaluation of the control expression (because the expression is initially `false`), $p_1(u)$ returns u (the environment after zero iterations through the loop). In all other cases, $p_1(u)$ returns $\perp$.

If a **while** loop terminates without error after at most two evaluations of the control expression, $p_2(u)$ returns v, the environment after loop termination. In all other cases, $p_2(u)$ returns $\perp$. In general, if a loop terminates after n iterations, $p_m(u)$ for $m \geq n$ will yield the environment after termination, given an initial environment u. For all terminating loops, the limit of $p_i(u)$ as $i \to \infty$ is the environment after loop termination. If the loop doesn't terminate or encounters a runtime error, then all p_i's return $\perp$, which is then trivially the limit as $i \to \infty$. The sequence of p_i's always converges, so the limit is always defined. This leads to the definition of a **while** loop given in Figure 10.52.

Figure 10.52

$$S[\textbf{while } T \textbf{ do } St]u = \lim_{i \to \infty} p_i(u) \qquad 1$$
$$\textbf{where } p_{i+1}(w) = e?Bool \Rightarrow (e \Rightarrow (v?\{\bot\} \Rightarrow \bot, p_i(v)), w), \bot; \qquad 2$$
$$e = E[T]w;\ v = S[St]w. \qquad 3$$

In general, the above limit is not computable (because the halting problem is undecidable), but the limit can be computed for some infinite loops (and all finite loops). For example, it doesn't take an oracle to decide that the loop in Figure 10.53 has some problems.

Figure 10.53

```
while true do                    1
    x := x + 1                   2
```

What does the denotational definition say about this loop? Assuming `true` hasn't been redefined, the semantic function is shown in Figure 10.54.

Figure 10.54

$$p_{i+1}(u) = (\texttt{true} \Rightarrow (v?\{\bot\} \Rightarrow \bot, p_i(v)), u) = v?\{\bot\} \Rightarrow \bot, p_i(v) \qquad 1$$
$$\textbf{where } v = S[St]u. \qquad 2$$

Now, $p_{i+1}(u)$ is either equal to $\bot$ or $p_i(v)$. Similarly, $p_i(v)$ is either equal to $\bot$ or $p_{i-1}(v')$. But $p_0(s) \equiv \bot$ for all s, so each p_i must reduce to $\bot$, so $\bot$ is the limit of the sequence. The loop fails either because `x` overflows or because the loop doesn't terminate. Since both failings are represented by $\bot$, the denotational definition has correctly handled this example.

3.13 Procedures

I now consider simple procedures of the abstract form shown in Figure 10.55.

Figure 10.55

```
procedure I;                     1
    St                           2
```

Procedures are invoked by a **`call`** statement (for example, **`call`** `I`). Since there are no scope rules yet, a procedure invocation is equivalent to macro substitution and immediate execution of the procedure's body. A procedure can call another procedure, but I will forbid recursion for now. Since a procedure name is a synonym for a list of statements, it represents a mapping from an environment to an updated environment or to $\bot$. The semantic domain for procedure declarations is given in Figure 10.56.

Figure 10.56

$$Proc = U \to (U \oplus \{\bot\})$$

I need to upgrade the environment domain to include procedures, as well as introduce a new flag `opencall`. I will set `opencall` when a procedure call is in progress, but not yet completed. To prevent recursion, I will disallow invoking a procedure that has `opencall` set. The environment domain U is now as shown in Figure 10.57.

Figure 10.57

```
V = N ⊕ Bool ⊕ Proc ⊕ {⊥} ⊕ {udef} ⊕{redef} -- id value                    1
U = Id → {var, const, uninit, opencall} ⊗ V -- environments                2
```

These domain equations are recursive: U references Proc, and Proc references U. Before, I used f[x ← y] to denote the function equal to f for all parameters except x, where y is to be returned. In the case that y is a member of a product domain, I will extend the notation;

```
f[Hd[x ← y]]
```

will denote the function equal to f for all parameters except x, where Hd(f(x)) = y, but Tl(f(x)) is unchanged; f[Tl[x ← y]] will have an analogous definition. Figure 10.58 gives the new part of the definition, building on Figure 10.46 (page 333).

Figure 10.58

```
Abstract syntax                                                            1

    Def → procedure I; St                                                  2
    St → call I                                                            3

Semantic domains                                                           4

    Proc = U → (U ⊕ {⊥}) -- procedure declaration                          5
    V = N ⊕ Bool ⊕ Proc ⊕ {⊥} ⊕ {udef} ⊕{redef} -- id value                6
    U = Id → {var, const, uninit, opencall} ⊗ V -- environments            7

Semantic functions                                                         8

    D[procedure I; St]u = u[I]?{udef} ⇒ u[I ← c], u[I ← redef]             9
        where c = < const, InProc(S[St]) > .                               10

    S[call I]u = Tl(v)?Proc ∧ Hd(v) = const ∧ w?U ⇒                         11
        w[Hd[I ← const]], ⊥                                                12
        where v = u[I]; w = Tl(v)(u[Hd[I ← opencall]]);                    13
```

A procedure declaration (lines 9–10) updates the current environment u by calculating the meaning of the body St and converting the result to domain Proc (line 10). This result is used to build a meaning for I in the environment (line 9). The definition of procedure invocation in lines 11–13 first modifies I in the environment u to indicate the call is open, then applies the body of procedure I (Tl(v) in line 13), storing the resulting environment in w. It then returns w, but first restores the definition of I (line 12).

3.14 Functions

Functions, like procedures, execute a list of statements. They also return a value by evaluating an expression immediately prior to return. For the present, I will constrain functions to be nonrecursive. The abstract syntax of integer functions will be as shown in Figure 10.59.

Figure 10.59

```
integer function I;                                    1
      St;                                              2
return(T)                                              3
```

Boolean functions have an analogous structure. Functions can be called to yield a value via the **eval** operator (for example, **eval** F).

Introducing function calls into the language raises the specter of side effects. Since I am building a definition, I can handle side effects pretty much as I wish. I might, for example, make them invalid and enforce this rule by comparing the environment after function invocation with that in place before invocation. Any changes would indicate side effects and yield an error result. Alternately, I could erase side effects by resuming execution after a function call with the same environment in place before the call. Although these alternatives are easy to denote, neither would be particularly easy for a compiler writer to implement, especially after the language definition includes I/O.

In the interests of realism, I will bite the bullet and allow side effects. The structure of the E function, which defines the meaning of expressions (which must now include function calls), will change. It will return not only the result value but also an updated environment. I add this facility by defining RR, the new domain of results:

$$RR = U \otimes (N \oplus Bool \oplus \{\perp\})$$

The semantic domain for function calls is:

$$Func = U \rightarrow RR$$

The semantic domain V is also extended to include Func.

The language allows constant declarations of the form I = T. Now that T includes function calls, the definition of constants is complicated by the fact that a call may induce side effects in the environment. This situation is undesirable (though it could be modeled, of course), so I will follow the lead of most languages and assume that a function call in this context is forbidden by the concrete syntax. Figure 10.60 shows what functions add to the definition of Figure 10.46 (page 333).

Figure 10.60

Abstract syntax 1

```
Def → Integer function I; St; return(T);               2
Def → Boolean function I; St; return(T);               3

T → eval I -- function invocation                      4
```

Semantic domains

```
RR = U ⊗ (N ⊕ Bool ⊕ {⊥}) -- expression result
Func = U → RR
V = N ⊕ Bool ⊕ Proc ⊕ Func {⊥} ⊕ {udef} ⊕{redef} -- id value
U = Id → {var, const, uninit, opencall} ⊗ V -- environments
```

Semantic functions

```
E: Exp → U → RR

M[program (I) Def St end] = c?U ⇒ Tl(E[I]c), ⊥
      where u = D[Def]u0; c = S[St]u.

D[I = T]u = u[I]?{udef} ⇒ u[I ← f], u[I ← redef]
      where e = E[T]u; f = e?{⊥} ⇒ ⊥, < const, Tl(e) > .
D[Integer function I; St; return(T)]u =
      u[I]?{udef} ⇒ u[I ← f], u[I ← redef]
      where f = < const, InFunc(v) > ; c = S[St]; e(w) = E[T](c(w));
      v(w) = c(w)?{⊥} ∨ e(w)?{⊥} ⇒ ⊥, (Tl(e(w))?N ⇒ e(w), ⊥).
D[Boolean function I; St; return(T)]u =
      u[I]?{udef} ⇒ u[I ← f], u[I ← redef]
      where f = < const, InFunc(v) > ; c = S[St]; e(w) = E[T](c(w));
      v(w) = c(w)?{⊥} ∨ e(w)?{⊥} ⇒ ⊥, (Tl(e(w))?Bool ⇒ e(w), ⊥).

E[0]u = < u, 0 >
E[1]u = < u, 1 >
E[Seq 0] u = e?N ∧ range(2 × e)?N ⇒ < u, 2 × e > , ⊥
      where e = Tl(E[Seq]u).
E[Seq 1] u = e?N ∧ range(2 × e + 1)?N ⇒ < u, 2 × e + 1 > , ⊥
      where e = Tl(E[Seq]u).

E[T1 + T2] u = Tl(e)?N ∧ Tl(f)?N ∧
      range(Tl(e) + Tl(f))?N ⇒ < Hd(f), Tl(e) + Tl(f) > , ⊥
      where e = E[T1]u; f = E[T2]Hd(e).
E[T1 – T2] u = Tl(e)?N ∧ Tl(f)?N ∧
      range(Tl(e) – Tl(f))?N ⇒ < Hd(f), Tl(e) – Tl(f) > , ⊥
      where e = E[T1]u; f = E[T2]Hd(e).
E[T1 * T2] u = Tl(e)?N ∧ Tl(f)?N ∧
      range(Tl(e) × Tl(f))?N ⇒ < Hd(f), Tl(e) × Tl(f) > , ⊥
      where e = E[T1]u; f = E[T2]Hd(e).
E[T1 / T2] u = Tl(e)?N ∧ Tl(f)?N ∧ Tl(f)≠0 ∧
      range(Tl(e) / Tl(f))?N ⇒ < Hd(f), Tl(e)/Tl(f) > , ⊥
      where e = E[T1]u; f = E[T2]Hd(e).
E[T1 = T2] u = (Tl(e)?N ∧ Tl(f)?N) ∨ (Tl(e)?Bool ∧ Tl(f)?Bool) ⇒
      < Hd(f), (Tl(e) = Tl(f)) > , ⊥
      where e = E[T1]u; f = E[T2]Hd(e).
E[eval I]u = Tl(v)?Func ∧ Hd(v) = const ∧ w≠⊥ ⇒
      w[Hd[I ← const]], ⊥
      where v = u[I]; w = Tl(v)(u[Hd[I ← opencall]]).
E[I]u = v?({redef} ⊕ {⊥} ⊕{udef}) ⇒ ⊥,
      (Hd(v) = uninit ⇒ ⊥, < u, Tl(v) >)
      where v = u[I].
```

```
S[I:=T]u = v?({redef} ⊕ {⊥} ⊕{udef}) ∨                                    51
      (Hd(v) = const) ∨ e?{⊥} ⇒ ⊥,                                       52
      (Tl(e)?N ∧ Tl(v)?N) ∨ (Tl(e)?Bool ∧ Tl(v)?Bool) ⇒                  53
      Hd(e)[I ←< var, Tl(e) >], ⊥                                        54
      where e = E[T]u; v = u[I].                                         55
S[if T then St1 else St2]u =                                             56
      Tl(e)?Bool ⇒ (Tl(e) ⇒ S[St1]Hd(e), S[St2]Hd(e)), ⊥                 57
      where e = E[T]u.                                                   58
S[do T times St]u = Tl(e)?N ⇒ v_m(Hd(e)), ⊥                              59
      where e = E[T]u; m = max(0, Tl(e));                                60
      v_0(w) = w; v_{i+1}(w) = v_i(w)?U ⇒ S[St]v_i(w), ⊥.                 61
S[while T do St]u = lim_{i→∞} p_i(u)                                     62
      where p_0(w) = ⊥;                                                  63
      p_{i+1}(w) = Tl(e)?Bool ⇒                                          64
            (Tl(e) ⇒ (v?{⊥} ⇒ ⊥, p_i(v)), Hd(e)), ⊥;                     65
      e = E[T]w; v = S[St]Hd(e).                                         66
```

In line 14, I assume that the concrete syntax forbids function calls in the definition of a constant.

3.15 Recursive Routines

The danger in allowing recursive routines is that the definitions may become circular. As it stands, I define the meaning of a call in terms of the meaning of its body. If recursion is allowed, the meaning of a routine's body may itself be defined in terms of any calls it contains. My current definition breaks this potential circularity by forbidding calls of a routine (directly or indirectly) from its own body.

I will generalize the definition of a subroutine call to allow calls of bounded depth. The meaning of a routine with a maximum call depth of n will be defined in terms of the meaning of the subroutine's body with subsequent calls limited to a depth of $n-1$. The meaning of a call with a maximum depth of zero is $\perp$.

If a call to a routine will ever return, then it can be modeled by a call limited to depth n as long as n is sufficiently large. As n approaches ∞, the bounded-call-depth model converges to the unbounded-call model if the latter ever returns. But if a routine call doesn't ever return, then the bounded-call-depth model will always produce an error result $\perp$, which is a correct definition of an infinite recursion. Thus the limit as n approaches ∞ of the bounded-call-depth model is $\perp$, which I will take as the definition of the meaning of a call of unbounded depth that never returns. This approach parallels how I handled unbounded iteration, which isn't surprising, given the similarity of looping and subroutine call.

I will redefine U to replace the opencall flag with an integer representing the maximum depth to which a given procedure or function can be called. If this value is zero, the call is invalid. What used to be opencall is now represented by 0; the previous model always had a maximum call depth of 1. Figure 10.61 shows the necessary additions.

Figure 10.61

$$U = \mathtt{Id} \to (\{\mathtt{var}, \mathtt{const}, \mathtt{uninit}\} \oplus \mathsf{N}) \otimes \mathsf{V} \quad 1$$

$$S[\textbf{call}\ \mathtt{I}]u = \mathtt{Tl}(v)?\mathtt{Proc} \Rightarrow \lim_{i \to \infty} p_i(u, v), \bot \quad 2$$

where $v = u[\mathtt{I}]$; 3

$p_0(u', v') = \bot$; 4

$p_{i+1}(u', v') = \mathtt{Hd}(v') = \mathtt{const} \Rightarrow (w?\mathsf{U} \Rightarrow w[\mathtt{Hd}[\mathtt{I} \leftarrow \mathtt{const}]], \bot),$ 5

$\mathtt{Hd}(v') > 0 \wedge y?\mathsf{U} \Rightarrow y[\mathtt{Hd}[\mathtt{I} \leftarrow \mathtt{Hd}(v')]], \bot;$ 6

$w = \mathtt{Tl}(v')(u'[\mathtt{Hd}[\mathtt{I} \leftarrow \mathtt{i}]]);$ 7

$y = \mathtt{Tl}(v')(u'[\mathtt{Hd}[\mathtt{I} \leftarrow \mathtt{Hd}(v') - 1]]).$ 8

$$E[\textbf{eval}\ \mathtt{I}]u = \mathtt{Tl}(v)?\mathtt{Func} \Rightarrow \lim_{i \to \infty} p_i(u, v), \bot \quad 9$$

where $v = u[\mathtt{I}]$; 10

$p_0(u', v') = \bot$; 11

$p_{i+1}(u', v') = \mathtt{Hd}(v') = \mathtt{const} \Rightarrow (w \neq \bot\{\bot\} \Rightarrow \{\bot\}$ 12

$w[\mathtt{Hd}[\mathtt{I} \leftarrow \mathtt{const}]], \bot),$ 13

$\mathtt{Hd}(v') > 0 \wedge y \neq \bot \Rightarrow y[\mathtt{Hd}[\mathtt{I} \leftarrow \mathtt{Hd}(v')]], \bot;$ 14

$w = \mathtt{Tl}(v')(u'[\mathtt{Hd}[\mathtt{I} \leftarrow \mathtt{i}]]);$ 15

$y = \mathtt{Tl}(v')(u'[\mathtt{Hd}[\mathtt{I} \leftarrow \mathtt{Hd}(v') - 1]]).$ 16

3.16 Modeling Memory and Files

I am now ready to model variables more accurately. I will use a finite semantic domain Loc to name addressable memory locations. A semantic domain Mem will model memories as a mapping from Loc to an integer or Boolean value or to error values uninitInt, uninitBool, unalloc:

$$\mathtt{Mem} = \mathtt{Loc} \to \mathsf{N} \oplus \mathtt{Bool} \oplus \mathtt{uninitInt} \oplus \mathtt{uninitBool} \oplus \mathtt{unalloc}$$

The uninitialized flag will now be in the memory mapping, not the environment mapping. Two different uninit flags are used to remember the type an uninitialized location is expected to hold. If a memory location is marked as unalloc, then it can be allocated for use (and possibly deallocated later). If $m \in \mathtt{Mem}$, then I define alloc as follows:

$$\mathtt{alloc}(m) = \text{any } l \in \mathtt{Loc} \text{ such that } m(l) = \mathtt{unalloc}$$
$$= \bot \text{ if no such } l \text{ exists}$$

Alloc specifies no particular memory allocation pattern; this definition allows implementations the widest latitude in memory management.

I will model files as finite sequences over integers, Booleans, and eof, the end-of-file flag. I define the semantic domain File as:

$$\mathtt{File} = (\mathsf{N} \oplus \mathtt{Bool} \oplus \mathtt{eof})^*$$

That is, a file is a potentially infinite string of typed values. My definitions will never consider values in files following the first eof. Programs will now take an input file and produce an output file (or $\bot$). To model this semantics, I will have a semantic domain State that consists of a memory and a pair of files:

$$State = Mem \otimes File \otimes File$$

At any point during execution, the current state is a combination of the current memory contents, what is left of the input file, and what has been written to the output file.

My definition of environments will now more nearly match the symbol tables found in conventional compilers. I will map identifiers to constant values, locations or routines:

$$V = N \oplus Bool \oplus Loc \oplus Proc \oplus Func \oplus \{\perp\} \oplus \{udef\} \oplus \{redef\}$$
$$U = Id \to V$$

Statements will take an environment and a state and will produce an updated state or an error value. Declarations will take an environment and state and will produce an updated environment and state (since memory allocation, performed by declarations, will update the original state). Figure 10.62 shows the additions and changes to the formal definition.

Figure 10.62

Abstract syntax 1

P → **program** Def St **end** -- program 2

St → **read** I -- read statement 3
St → **write** T -- write statement 4

Semantic domains 5

State = Mem ⊗ File ⊗ File 6
RR = State ⊗ (N ⊕ Bool ⊕ {⊥}) 7
Proc = (U → State → (State ⊕ {⊥})) ⊗ Loc 8
Func = (U → State → RR) ⊗ Loc 9
Mem = Loc → N ⊕ Bool ⊕ {uninitInt} ⊕ {uninitBool} ⊕ 10
{unalloc} 11
File = (N ⊕ Bool ⊕ {eof})* 12
V = N ⊕ Bool ⊕ Loc ⊕ Proc ⊕ Func ⊕ {⊥}⊕{udef}⊕{redef} 13
U = Id → V 14

Semantic functions 15

E: Exp → U → State → RR 16
D: Decls → (U ⊗ State) → (U ⊗ State) 17
M: Pr → File → File ⊕ {⊥} 18
S: Stm → U → State → (State ⊕ {⊥}) 19

E[0] u s = < s, 0 > 20
E[1] u s = < s, 1 > 20
E[Seq0] u s = e?N ∧ range(2 × e)?N ⇒ < s, 2 × e > , ⊥ 22
where e = Tl(E[Seq] u s). 23
E[Seq1] u s = e?N ∧ range(2 × e + 1)?N ⇒ < s, 2 × e + 1 > , ⊥ 24
where e = Tl(E[Seq] u s). 25

$$
\begin{array}{ll}
\mathsf{E}[\mathsf{I}]\ \mathsf{u}\ \mathsf{s} = \mathsf{v}?(\{\bot\} \oplus \{\mathtt{redef}\} \oplus \{\mathtt{udef}\}) \Rightarrow \bot, & 26\\
\quad \mathsf{v}?\mathsf{Loc} \Rightarrow (\mathsf{m}(\mathsf{v})?(\{\mathtt{uninitInt}\}\oplus\{\mathtt{uninitBool}\}) \Rightarrow \bot, & 27\\
\quad < \mathsf{s}, \mathsf{m}(\mathsf{v}) >), < \mathsf{s}, \mathsf{v} > & 28\\
\quad \textbf{where}\ \ \mathsf{v} = \mathsf{u}[\mathsf{I}];\ \mathsf{s} = < \mathsf{m}, \mathsf{i}, \mathsf{o} > . & 29\\
\mathsf{E}[\mathsf{T}_1 + \mathsf{T}_2]\ \mathsf{u}\ \mathsf{s} = \mathsf{Tl}(\mathsf{e})?\mathsf{N} \wedge \mathsf{Tl}(\mathsf{f})?\mathsf{N} \wedge & 30\\
\quad \mathsf{range}(\mathsf{Tl}(\mathsf{e}) + \mathsf{Tl}(\mathsf{f}))?\mathsf{N} \Rightarrow < \mathsf{Hd}(\mathsf{f}), \mathsf{Tl}(\mathsf{e}) + \mathsf{Tl}(\mathsf{f}) > , \bot & 31\\
\quad \textbf{where}\ \ \mathsf{e} = \mathsf{E}[\mathsf{T}_1]\ \mathsf{u}\ \mathsf{s};\ \mathsf{f} = \mathsf{E}[\mathsf{T}_2]\ \mathsf{u}\ \mathsf{Hd}(\mathsf{e}). & 32\\
\mathsf{E}[\mathsf{T}_1 - \mathsf{T}_2]\ \mathsf{u}\ \mathsf{s} = \mathsf{Tl}(\mathsf{e})?\mathsf{N} \wedge \mathsf{Tl}(\mathsf{f})?\mathsf{N} \wedge & 33\\
\quad \mathsf{range}(\mathsf{Tl}(\mathsf{e}) - \mathsf{Tl}(\mathsf{f}))?\mathsf{N} \Rightarrow < \mathsf{Hd}(\mathsf{f}), \mathsf{Tl}(\mathsf{e}) - \mathsf{Tl}(\mathsf{f}) > , \bot & 34\\
\quad \textbf{where}\ \ \mathsf{e} = \mathsf{E}[\mathsf{T}_1]\ \mathsf{u}\ \mathsf{s};\ \mathsf{f} = \mathsf{E}[\mathsf{T}_2]\ \mathsf{u}\ \mathsf{Hd}(\mathsf{e}). & 35\\
\mathsf{E}[\mathsf{T}_1 * \mathsf{T}_2]\ \mathsf{u}\ \mathsf{s} = \mathsf{Tl}(\mathsf{e})?\mathsf{N} \wedge \mathsf{Tl}(\mathsf{f})?\mathsf{N} \wedge & 36\\
\quad \mathsf{range}(\mathsf{Tl}(\mathsf{e}) \times \mathsf{Tl}(\mathsf{f}))?\mathsf{N} \Rightarrow < \mathsf{Hd}(\mathsf{f}), \mathsf{Tl}(\mathsf{e}) \times \mathsf{Tl}(\mathsf{f}) > , \bot & 37\\
\quad \textbf{where}\ \ \mathsf{e} = \mathsf{E}[\mathsf{T}_1]\ \mathsf{u}\ \mathsf{s};\ \mathsf{f} = \mathsf{E}[\mathsf{T}_2]\ \mathsf{u}\ \mathsf{Hd}(\mathsf{e}). & 38\\
\mathsf{E}[\mathsf{T}_1 / \mathsf{T}_2]\ \mathsf{u}\ \mathsf{s} = \mathsf{Tl}(\mathsf{e})?\mathsf{N} \wedge \mathsf{Tl}(\mathsf{f})?\mathsf{N} \wedge \mathsf{Tl}(\mathsf{f}) \neq 0 \wedge & 39\\
\quad \mathsf{range}(\mathsf{Tl}(\mathsf{e})/\mathsf{Tl}(\mathsf{f}))?\mathsf{N} \Rightarrow < \mathsf{Hd}(\mathsf{f}), \mathsf{Tl}(\mathsf{e})/\mathsf{Tl}(\mathsf{f}) > , \bot & 40\\
\quad \textbf{where}\ \ \mathsf{e} = \mathsf{E}[\mathsf{T}_1]\ \mathsf{u}\ \mathsf{s};\ \mathsf{f} = \mathsf{E}[\mathsf{T}_2]\ \mathsf{u}\ \mathsf{Hd}(\mathsf{e}). & 41\\
\mathsf{E}[\mathsf{T}_1 = \mathsf{T}_2]\ \mathsf{u}\ \mathsf{s} = (\mathsf{Tl}(\mathsf{e})?\mathsf{N} \wedge \mathsf{Tl}(\mathsf{f})?\mathsf{N}) \vee (\mathsf{Tl}(\mathsf{e})?\mathsf{Bool} \wedge \mathsf{Tl}(\mathsf{f})?\mathsf{Bool}) \Rightarrow & 42\\
\quad < \mathsf{Hd}(\mathsf{f}), (\mathsf{Tl}(\mathsf{e}) = \mathsf{Tl}(\mathsf{f})) > , \bot & 43\\
\quad \textbf{where}\ \ \mathsf{e} = \mathsf{E}[\mathsf{T}_1]\ \mathsf{u}\ \mathsf{s};\ \mathsf{f} = \mathsf{E}[\mathsf{T}_2]\ \mathsf{u}\ \mathsf{Hd}(\mathsf{e}). & 44\\
\mathsf{E}[\textbf{eval}\,\mathsf{I}]\ \mathsf{u}\ \mathsf{s} = \mathsf{v}?\mathsf{Func} \Rightarrow \lim\limits_{\mathsf{i} \to \infty} \mathsf{p}_\mathsf{i}(\mathsf{s}, \mathsf{v}), \bot & 45\\
\quad \textbf{where}\ \ \mathsf{v} = \mathsf{u}[\mathsf{I}];\ \mathsf{p}_0(\mathsf{s}', \mathsf{v}') = \bot; & 46\\
\quad \mathsf{p}_{\mathsf{i}+1}(\mathsf{s}', \mathsf{v}') = \mathsf{m}(l)?\{\mathtt{uninitInt}\} \Rightarrow & 47\\
\qquad (\mathsf{w}?\{\bot\} \Rightarrow \bot, \mathsf{w}[\mathsf{Hd}[l \leftarrow \mathtt{uninitInt}]]); & 48\\
\quad \mathsf{m}(l) > 0 \wedge \mathsf{y}?\{\bot\} \Rightarrow \bot, \mathsf{y}[\mathsf{Hd}[l \leftarrow \mathsf{m}(l)]]; & 49\\
\quad \mathsf{s}' = < \mathsf{m}, \mathsf{I}, \mathsf{O} > ;\ \mathsf{v}' = < \mathsf{f}, l > ;\ \mathsf{w} = \mathsf{fu} < \mathsf{m}[l \leftarrow \mathsf{i}], \mathsf{I}, \mathsf{O} > ; & 50\\
\quad \mathsf{y} = \mathsf{fu} < \mathsf{m}[l \leftarrow \mathsf{m}(l) - 1], \mathsf{I}, \mathsf{O} > . & 51
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{D}[\varepsilon] < \mathsf{u}, \mathsf{s} > = < \mathsf{u}, \mathsf{s} > & 52\\
\mathsf{D}[\mathsf{I}\text{: }\mathtt{integer}] < \mathsf{u}, \mathsf{s} > = \mathsf{u}[\mathsf{I}]?\{\mathtt{udef}\} \Rightarrow & 53\\
\quad (l\,?\{\bot\} \Rightarrow < \mathsf{u}[\mathsf{I} \leftarrow \bot], \mathsf{s} > , < \mathsf{u}[\mathsf{I} \leftarrow l], & 54\\
\quad \mathsf{i} < \mathsf{m}[l \leftarrow \mathtt{uninitInt}], \mathsf{i}, \mathsf{o} >>), < \mathsf{u}[\mathsf{I} \leftarrow \mathtt{redef}], \mathsf{s} > & 55\\
\quad \textbf{where}\ \ \mathsf{s} = < \mathsf{m}, \mathsf{i}, \mathsf{o} > ;\ l = \mathtt{alloc}(\mathsf{m}). & 56\\
\mathsf{D}[\mathsf{I}\text{: }\mathtt{Boolean}] < \mathsf{u}, \mathsf{s} > = \mathsf{u}[\mathsf{I}]?\{\mathtt{udef}\} \Rightarrow & 57\\
\quad (l\,?\{\bot\} \Rightarrow < \mathsf{u}[\mathsf{I} \leftarrow \bot], \mathsf{s} > , & 58\\
\quad < \mathsf{u}[\mathsf{I} \leftarrow l], < \mathsf{m}[l \leftarrow \mathtt{uninitBool}], \mathsf{i}, \mathsf{o} >>), & 59\\
\quad < \mathsf{u}[\mathsf{I} \leftarrow \mathtt{redef}], \mathsf{s} > & 60\\
\quad \textbf{where}\ \ \mathsf{s} = < \mathsf{m}, \mathsf{i}, \mathsf{o} > ;\ l = \mathtt{alloc}(\mathsf{m}). & 61\\
\mathsf{D}[\mathsf{Def}_1\ \mathsf{Def}_2] < \mathsf{u}, \mathsf{s} > = \mathsf{D}[\mathsf{Def}_2] < \mathsf{v}, \mathsf{t} > & 62\\
\quad \textbf{where}\ \ < \mathsf{v}, \mathsf{t} > = \mathsf{D}[\mathsf{Def}_1] < \mathsf{u}, \mathsf{s} > . & 63\\
\mathsf{D}[\mathsf{I} = \mathsf{T}] < \mathsf{u}, \mathsf{s} > = \mathsf{u}[\mathsf{I}]?\{\mathtt{udef}\} \Rightarrow < \mathsf{u}[\mathsf{I} \leftarrow \mathsf{f}], \mathsf{s} > , & 64\\
\quad < \mathsf{u}[\mathsf{I} \leftarrow \mathtt{redef}], \mathsf{s} > & 65\\
\quad \textbf{where}\ \ \mathsf{e} = \mathsf{E}[\mathsf{T}]\ \mathsf{u}\ \mathsf{s};\ \mathsf{f} = \mathsf{e}?\{\bot\} \Rightarrow \bot, \mathsf{Tl}(\mathsf{e}). & 66\\
\mathsf{D}[\textbf{procedure}\ \mathsf{I};\ \mathsf{St}] < \mathsf{u}, \mathsf{s} > = \mathsf{u}[\mathsf{I}]?\{\mathtt{udef}\} \Rightarrow & 67\\
\quad (l\,?\{\bot\} \Rightarrow < \mathsf{u}[\mathsf{I} \leftarrow \bot], \mathsf{s} > , < \mathsf{u}[\mathsf{I} \leftarrow < \mathsf{c}, l >], & 68\\
\quad < \mathsf{m}[l \leftarrow \mathtt{uninitInt}], \mathsf{i}, \mathsf{o} >>), < \mathsf{u}[\mathsf{I} \leftarrow \mathtt{redef}], \mathsf{s} > & 69\\
\quad \textbf{where}\ \ \mathsf{c} = \mathsf{S}[\mathsf{St}];\ \mathsf{s} = < \mathsf{m}, \mathsf{i}, \mathsf{o} > ;\ l = \mathtt{alloc}(\mathsf{m}). & 70\\
\mathsf{D}[\mathtt{Integer}\ \textbf{function}\ \mathsf{I};\ \mathsf{St};\ \textbf{return}(\mathsf{T})] < \mathsf{u}, \mathsf{s} > = & 71\\
\quad \mathsf{u}[\mathsf{I}]?\{\mathtt{udef}\} \Rightarrow (l\,?\{\bot\} \Rightarrow < \mathsf{u}[\mathsf{I} \leftarrow \bot], \mathsf{s} > , & 72\\
\quad < \mathsf{u}[\mathsf{I} \leftarrow < \mathsf{v}, l >], < \mathsf{m}[l \leftarrow \mathtt{uninitInt}], \mathsf{i}, \mathsf{o} >>), & 73\\
\quad < \mathsf{u}[\mathsf{I} \leftarrow \mathtt{redef}], \mathsf{s} > & 74\\
\quad \textbf{where}\ \ \mathsf{s} = < \mathsf{m}, \mathsf{i}, \mathsf{o} > ;\ l = \mathtt{alloc}(\mathsf{m});\ \mathsf{c} = \mathsf{S}[\mathsf{St}]; & 75\\
\quad \mathsf{e}(\mathsf{w}, \mathsf{t}) = \mathsf{E}[\mathsf{T}]\ \mathsf{w}\ \mathsf{c}(\mathsf{w}, \mathsf{t}); & 76\\
\quad \mathsf{v}(\mathsf{w}, \mathsf{t}) = \mathsf{c}(\mathsf{w}, \mathsf{t})?\{\bot\} \vee \mathsf{e}(\mathsf{w}, \mathsf{t})?\{\bot\} \Rightarrow \bot, & 77\\
\qquad (\mathsf{Tl}(\mathsf{e}(\mathsf{w}, \mathsf{t}))?\mathsf{N} \Rightarrow \mathsf{e}(\mathsf{w}, \mathsf{t}), \bot). & 78
\end{array}
$$

```
D[Boolean function I; St; return(T)] < u, s > =                                  79
      u[I]?{udef} ⇒ (l ?{⊥} ⇒ < u[I ← ⊥], s > ,                                  80
      < u[I ←< v, l >], < m[l ← uninitInt], i, o >>),                            81
      < u[I ← redef], s >                                                        82
      where  s = < m, i, o > ; l = alloc(m); c = S[St];                          83
      e(w, t) = E[T] w c(w, t);                                                  84
      v(w, t) = c(w, t)?{⊥} ∨ e(w, t)?{⊥} ⇒ ⊥,                                   85
            (Tl(e(w, t))?Bool ⇒ e(w, t), ⊥).                                     86

M[program Def St end]i = c?{⊥} ⇒ ⊥, Tl(Tl(c))                                    87
      where  < u, s > = D[Def] < u0, < m0, i, eof >> ;                           88
      c = S[St] u s                                                              89

S[ε] u s = s                                                                     90
S[St1 St2] u s = g?{⊥} ⇒ ⊥, S[St2] u g                                           91
      where  g = S[St1] u s.                                                     92
S[I:= T] u s = v?Loc ∧                                                           93
      ((Tl(e)?N ∧ m(v)?N ⊕{uninitInt}) ∨                                         94
      (Tl(e)?Bool ∧ m(v)?Bool ⊕{uninitBool})) ⇒                                  95
      < m[v ← Tl(e) >], i, o > , ⊥                                               96
      where  e = E[T] u s; Hd(e) = < m, i, o > ; v = u[I].                       97
S[read I] u s = v?Loc ∧ i≠eof ∧                                                  98
      ((Hd(i)?N ∧ m(v)?N ⊕{uninitInt}) ∨                                         99
      (Hd(i)?Bool ∧ m(v)?Bool ⊕{uninitBool})) ⇒                                  100
      < m[v ← Hd(i) >], Tl(i), o > , ⊥                                           101
      where  s = < m, i, o > ; v = u[I].                                         102
S[write T] u s = e?{⊥} ⇒ ⊥, < m, i, append(o, < Tl(e), eof >) >                  103
      where  e = E[T] u s; Hd(e) = < m, i, o > .                                 104
S[if T then St1 else St2] u s =                                                  105
      Tl(e)?Bool ⇒ (Tl(e) ⇒ S[St1] u Hd(e), S[St2] u Hd(e)), ⊥                   106
      where  e = E[T] u s.                                                       107
S[do T times St] u s = Tl(e)?N ⇒ vm(Hd(e)), ⊥                                    108
      where  e = E[T] u s; m = max(0, Tl(e)); v0(w) = w;                         109
      vi+1(w) = vi(w)?{⊥} ⇒ ⊥, S[St] u vi(w).                                    110
S[while T do St] u s = lim_{i→∞} pi(s)                                           111
      where  p0(w) = ⊥;                                                          112
      pi+1(w) = Tl(e)?Bool ⇒ (Tl(e) ⇒                                            113
            (v?{⊥} ⇒ ⊥, pi(v)), Hd(e)), ⊥;                                       114
      e = E[T] u w; v = S[St] u Hd(e).                                           115
S[call I] u s = v?Proc ⇒ lim_{i→∞} pi(s, v), ⊥                                   116
      where  v = u[I];                                                           117
      p0(s′, v′) = ⊥;                                                            118
      pi+1(s′, v′) = m(l)?{uninitInt} ⇒ (w?{⊥} ⇒ ⊥,                              119
            w[Hd[l ← uninitInt]]),                                               120
            m(l) > 0 ∧ y?{⊥} ⇒ ⊥, y[Hd[l ← m(l)]];                               121
      s′ = < m, I, O > ; v′ = < f, l > ; w = f u < m[l ← i], I, O > ;            122
      y = f u < m[l ← m(l) − 1], I, O > .                                        123
```

The syntax for programs (line 2) no longer needs an identifier in the header. I assume integers and Booleans each require one location in memory. I still forbid function calls in the definition of a constant. Append (line 102) concatenates two sequences, each terminated by eof. The initial memory configuration, in which all locations map to unalloc, is m_0 (line 87).

The location associated with procedures and functions (lines 8 and 9) is used to hold the depth count, which appears in the definition of procedure (lines 115–122) and function (lines 44–50) calls. This count is no longer kept in the environment, because expressions and statements now update states, not environments. If no calls of a routine are in progress, its associated memory location will contain `uninitInt`.

3.17 Blocks and Scoping

I will now model block structure and name scoping by adding a **begin-end** block to the syntax, as in Figure 10.63.

Figure 10.63

```
St → begin Def St end
```

As in most block-structured languages, declarations within a block are local to it, and local redefinition of a nonlocal identifier is allowed. Rather than a single environment, I will employ a sequence of environments, with the first environment representing local declarations, and the last environment representing the outermost (predeclared) declarations. The new semantic domain UU = U* will represent this sequence of environments. All definitions will be made in the head of the environment sequence, while lookup will proceed through the sequence of environments, using the functions Top and Find, shown in Figure 10.64.

Figure 10.64

```
Top: UU → U                                                          1
Top(u) = u?U ⇒ u, Hd(u)                                              2

Find: UU → Id → V                                                    3
Find(u)[I] = Top(u)[I]?{udef} ⇒                                      4
      (u?U ⇒ ⊥, Find(Tl(u))[I]), Top(u)[I]                           5
```

Block structure introduces a memory-management issue. Most languages specify that memory for local variables is created (or allocated) upon block entry and released upon block exit. To model allocation, I create a function Free (Figure 10.65) that records the set of free memory locations.

Figure 10.65

```
Free: Mem → 2^Loc                                                    1
Free(m) = {l | m(l)=unalloc}                                         2
```

I will record free locations at block entry and reset them at block exit. Most implementations do this by pushing and later popping locations from a runtime stack. My definition, of course, does not require any particular implementation.

Figure 10.66 presents the definition of block structure, updating all definitions that explicitly use environments so that they now use sequences of environments. I also modify slightly the definition of the main program to put predeclared identifiers in a scope outside that of the main program.

Figure 10.66

Abstract syntax

St → **begin** Def St **end**

Semantic domains

UU = U* -- sequence of environments
Proc = (UU → State → (State ⊕ {⊥})) ⊗Loc
Func = (UU → State → RR) ⊗ Loc

Semantic functions

E: Exp → UU → State → RR
D: Decls → (UU ⊗ State) → (UU ⊗ State)
S: Stm → UU → State → (State ⊕ {⊥})

S[**begin** Def St **end**] u s = c?{⊥} ⇒ ⊥,
  < m[Free(Hd(s)) ← unalloc], i, o >
  where < v, t > = D[Def] << u_e, u > , s > ;
  c = S[St] v t = < m, i, o > .

M[**program** Def St **end**]i = c?{⊥} ⇒ ⊥, Tl(Tl(c))
  where < u, s > = D[Def] << u_e, u_0 > , < m_0, i, eof >> ;
  c = S[St] u s.

E[I] u s = v?({⊥} ⊕ {redef} ⊕ {udef}) ⇒ ⊥,
  v?Loc ⇒ (m(v)?({uninitInt}⊕{uninitBool}) ⇒ ⊥,
  < s, m(v) >), < s, v >
  where v = Find(u)[I]; s = < m, i, o > .
E[**eval** I] u s = v?Func ⇒ $\lim_{i \to \infty} p_i(s, v)$, ⊥
  where v = Find(u)[I];
  $p_0(s', v') = \perp$;
  $p_{i+1}(s', v') = m(l)?\{\text{uninitInt}\} \Rightarrow (w?\{\perp\} \Rightarrow$
    ⊥, w[Hd[l ← uninitInt]]),
    $m(l) > 0 \wedge y?\{\perp\} \Rightarrow \perp$, y[Hd[$l \leftarrow m(l)$]];
  s′ = < m, I, O > ; v′ = < f, l > ;
  w = f u < m[l ← i], I, O > ;
  y = f u < m[$l \leftarrow m(l) - 1$], I, O > .

D[I: integer] < u, s > = Hd(u)[I]?{udef} ⇒
  (l ?{⊥} ⇒ < u[Hd[I ← ⊥]], s > , < u[Hd[I ← l]],
  < m[l ← uninitInt], i, o >>), < u[Hd[I ← redef]], s >
  where s = < m, i, o > ; l = alloc(m).
D[I: Boolean] < u, s > = Hd(u)[I]?{udef} ⇒
  (l ?{⊥} ⇒ < u[Hd[[I ← ⊥]], s > , < u[Hd[[I ← l]],
  < m[l ← uninitBool], i, o >>), < u[Hd[[I ← redef]], s >
  where s = < m, i, o > ; l = alloc(m).
D[I = T] < u, s > = Hd(u)[I]?{udef} ⇒ < u[Hd[I ← f]], s > ,
  < u[Hd[I ← redef]], s >
  where e = E[T] u s; f = e?{⊥} ⇒ ⊥, Tl(e).

```
D[procedure I; St] < u, s > = Hd(u)[I]?{udef} ⇒                                  42
      (l ?{⊥} ⇒ < u[Hd[I ← ⊥]], s > , < u[Hd[I ←< c, l >]],                      43
      < m[l ← uninitInt], i, o >>), < u[Hd[I ← redef]], s >                      44
      where c = S[St]; s = < m, i, o > ; l = alloc(m).                           45
D[Integer function I; St; return(T)] < u, s > =                                  46
      Hd(u)[I]?{udef} ⇒ (l ?{⊥} ⇒ < u[Hd[I ← ⊥]], s > ,                          47
      < [Hd[I ←< v, l >]], < m[l ← uninitInt], i, o >>),                         48
      < u[Hd[I ← redef]], s >                                                    49
      where s = < m, i, o > ; l = alloc(m); c = S[St];                           50
      e(w, t) = E[T] w c(w, t);                                                  51
      v(w, t) = c(w, t)?{⊥} ∨ e(w, t)?{⊥} ⇒                                      52
            ⊥, (Tl(e(w, t))?N ⇒ e(w, t), ⊥).                                     53
D[Boolean function I; St; return(T)] < u, s > =                                  54
      Hd(u)[I]?{udef} ⇒ (l ?{⊥} ⇒ < u[Hd[I ← ⊥]], s > ,                          55
      < u[Hd[I ←< v, l >]], < m[l ← uninitInt], i, o >>),                        56
      < u[Hd[I ← redef]], s >                                                    57
      where s = < m, i, o > ; l = alloc(m); c = S[St];                           58
      e(w, t) = E[T] w c(w, t);                                                  59
      v(w, t) = c(w, t)?{⊥} ∨ e(w, t)?{⊥} ⇒ ⊥,                                   60
            (Tl(e(w, t))?Bool ⇒ e(w, t), ⊥).                                     61

S[I:= T] u s =                                                                   62
      v?Loc ∧ ((Tl(e)?N ∧ m(v)?N ⊕{uninitInt}) ∨                                 63
      (Tl(e)?Bool ∧ m(v)?Bool ⊕{uninitBool})) ⇒                                  64
      < m[v ← Tl(e) >], i, o > , ⊥                                               65
      where e = E[T] u s; Hd(e) = < m, i, o > ; v = Find(u)[I].                   66
S[read I] u s = v?Loc ∧ i ≠ eof ∧                                                67
      ((Hd(i)?N ∧ m(v)?N ⊕{uninitInt}) ∨ (Hd(i)?Bool ∧                           68
      m(v)?Bool ⊕ {uninitBool})) ⇒ < m[v ← Hd(i) >], Tl(i), o > , ⊥              69
      where s = < m, i, o > ; v = Find(u)[I].                                    70
S[call I] u s = v?Proc ⇒ lim_{i→∞} p_i(s, v), ⊥                                  71
      where v = Find(u)[I];                                                      72
      p_0(s′, v′) = ⊥;                                                           73
      p_{i+1}(s′, v′) = m(l)?{uninitInt} ⇒ (w?{⊥} ⇒ ⊥,                           74
            w[Hd[l ← uninitInt]]),                                               75
            m(l) > 0 ∧ y?{⊥} ⇒ ⊥, y[Hd[l ← m(l)]];                               76
      s′ = < m, I, 0 > ; v′ = < f, l > ;                                         77
      w = f u < m[l ← i], I, 0 > ;                                               78
      y = f u < m[l ← m(l) − 1], I, 0 > .                                        79
```

In lines 13 and 16, u_e is the empty environment in which all identifiers map to udef.

3.18 Parameters

Now that I have scoping, I will turn my attention to procedures and functions. As defined above, procedures and functions execute in the environment of the call, not the environment of definition. No environment is stored with a procedure or function definition; rather, they use an environment provided at the point of call. In other words, I have provided dynamic scoping and shallow binding, which is common in interpreted, but not in compiled, languages. I will now refine the model to use the more common static model of scoping.

I will also include reference-mode parameters to illustrate how parameter definition and binding are handled. The approach will be similar to that used with blocks. However, when a procedure or function is called, I will provide an initial local environment in which parameter names have been bound to the locations associated with corresponding actual parameters. This approach allows the possibility of aliasing. I will be careful therefore not to release storage associated with formal parameters, since this storage will belong to the actual parameters (which persist after the call). However, other local definitions will be treated like locals declared in blocks and released after the call.

Figure 10.67 extends the syntax of routine definitions and calls to include parameters:

Figure 10.67 **Abstract syntax**

```
Actuals ∈ Aparms
Formals ∈ Fparms

Def → procedure I ( Formals ); begin Def St end
Def → Integer function I ( Formals );
      Def St return(T);
Def → Boolean function I ( Formals );
      Def St return(T);
St → call I (Actuals)
T → eval I (Actuals)
Formals → I : integer;
Formals → I : Boolean;
Formals → ε
Formals → Formals Formals
Actuals → ε
Actuals → I
Actuals → Actuals Actuals
```

In the concrete syntax, a routine with no parameters may well omit parentheses, and actuals will be separated by commas. I don't have to worry about such details at the level of abstract syntax.

I will also create two new semantic functions, FP and AP, to define the meaning of formal and actual parameters. Figure 10.68 shows the changes to the definition.

Figure 10.68 **Semantic domains**

```
Parms = ((N ⊕ Bool) ⊗ Id ⊕ eol)*
Proc = (UU → State → (State ⊕ {⊥})) ⊗ Loc ⊗ Parms
Func = (UU → State → RR) ⊗ Loc ⊗ Parms
```

Semantic functions

```
FP: Fparms → Parms → Parms -- Formals
AP: Aparms → (UU ⊗Parms) → State → ((UU ⊗Parms) ⊕ {⊥})
      -- Actuals
```

```
FP[I: integer]p = append(p, << 0, I > , eol >)                                  9
FP[I: Boolean]p = append(p, << false, I > , eol >)                              10
FP[ε]p = p                                                                      11
FP[Formals1 Formals2] p = FP[Formals2]q                                         12
      where q = FP[Formals1]p.                                                  13

AP[I] < u, p > s = v?Loc ∧ p≠eol ∧                                              14
      ((Hd(pp)?N ∧ m(v)?N ⊕ {uninitInt}) ∨ (Hd(pp)?Bool ∧                        15
      m(v)?Bool ⊕ {uninitBool})) ⇒                                              16
      < u[Hd[Tl(pp) ← v]], Tl(p) > , ⊥                                          17
      where v = Find(Tl(u))[I]; pp = Hd(p); s = < m, i, o > .                   18
AP[ε] < u, p > s = < u, p >                                                     19
AP[Actuals1 Actuals2] < u, p > s = q?{⊥} ⇒ ⊥,                                   20
      AP[Actuals2] q s                                                          21
      where q = AP[Actuals1] < u, p > s.                                        22

D[procedure I (Formals); Def St] < u, s > =                                     23
      Hd(u)[I]?{udef} ⇒ (l ?{⊥} ⇒ < u[Hd[I ← ⊥]], s > ,                         24
      < uu, < m[l ← uninitInt], i, o >>), < u[Hd[I ← redef]], s >               25
      where f(v, t) = S[St]v′t′; < v′, t′ > = D[Def] << v, uu > , t > ;         26
      s = < m, i, o > ; l = alloc(m);                                           27
      uu = u[Hd[I ←< f, l, p >]]; p = FP[Formals]eol.                           28
D[Integer function(Formals) I; Def St return(T)]                                29
      < u, s > = Hd(u)[I]?{udef} ⇒                                              30
      (l ?{⊥} ⇒ < u[Hd[I ← ⊥]], s > ,                                           31
      < uu, < m[l ← uninitInt], i, o >>), < u[Hd[I ← redef]], s >               32
      where s = < m, i, o > ; l = alloc(m); e(w, r) = E[T](w c(w, r));          33
      c(v, t) = S[St]v′t′; < v′, t′ > = D[Def] << v, uu > , t > ;               34
      f(vv, tt) = c(vv, tt)?{⊥} ∨ e(vv, tt)?{⊥} ⇒                               35
            ⊥, (Tl(e(vv, tt))?N ⇒ e(vv, tt), ⊥);                                36
      uu = u[Hd[I ←< f, l, p >]]; p = FP[Formals]eol.                           37
D[Boolean function(Formals) I; Def St return(T)]                                38
      < u, s > = Hd(u)[I]?{udef} ⇒                                              39
      (l ?{⊥} ⇒ < u[Hd[I ← ⊥]], s > ,                                           40
      < uu, < m[l ← uninitInt], i, o >>), < u[Hd[I ← redef]], s >               41
      where s = < m, i, o > ; l = alloc(m); e(w, r) = E[T](w c(w, r));          42
      c(v, t) = S[St]v′t′; < v′, t′ > = D[Def] << v, uu > , t > ;               43
      f(vv, tt) = c(vv, tt)?{⊥} ∨ e(vv, tt)?{⊥} ⇒                               44
            ⊥, (Tl(e(vv, tt))?Bool ⇒ e(vv, tt), ⊥);                             45
      uu = u[Hd[I ←< f, l, p >]]; p = FP[Formals]eol.                           46

S[call I(Actuals)] u s = v?Proc ∧                                               47
      q≠⊥ ∧ Tl(q) = eol ⇒ lim_{i→∞} p_i(s, Hd(Hd(q))), ⊥                        48
      where v = Find(u)[I] = < f, l, r > ;                                      49
      p0(s′, u′) = ⊥;                                                           50
      p_{i+1}(s′, u′) = m(l)?{uninitInt} ⇒ (w?{⊥} ⇒ ⊥, ww),                      51
            m(l) > 0 ∧ y?{⊥} ⇒ ⊥, yy;                                           52
      q = AP[Actuals] << ue, u > , r > s;                                       53
      s′ = < m, I, 0 > ;                                                        54
      w = f u′ < m[l ← i], I, 0 > ;                                             55
      ww = w[Hd[l ← uninitInt]][Hd[Free(m) ← unalloc]];                         56
      y = f u′ < m[l ← m(l) − 1], I, 0 > ;                                      57
      yy = y[Hd[l ← m(l)]][Hd[Free(m) ← unalloc]].                              58
```

```
E[eval I(Actuals)] u s = v?Func ∧ q ≠ ⊥ ∧ Tl(q) = eol ⇒                    59
    lim_{i→∞} p_i(s, Hd(Hd(q))), ⊥                                          60
    where v = Find(u)[I] = < f, l, r > ;                                    61
    p_0(s', u') = ⊥;                                                        62
    p_{i+1}(s', u') = m(l)?{uninitInt} ⇒ (w?{⊥} ⇒ ⊥, ww),                    63
            m(l) > 0 ∧ y?{⊥} ⇒ ⊥, yy;                                       64
    q = AP[Actuals] << u_e, u > , r > s;                                    65
    s' = < m, I, O > ;                                                      66
    w = f u' < m[l ← i], I, O > ;                                           67
    ww = w[Hd[l ← uninitInt]][Hd[Free(m) ← unalloc]];                       68
    y = f u' < m[l ← m(l) − 1], I, O > ;                                    69
    yy = y[Hd[l ← m(l)]][Hd[Hd[Free(m) ← unalloc]]].                        70
```

The eol in line 2 represents "end of list."

3.19 Continuations

The denotational approach is very structured; the meaning of a construct is defined in terms of a composition of the meanings of the construct's constituents. The meaning of a program can be viewed as a top-down traversal of an abstract syntax tree from the root (the program nonterminal) to the leaves (identifiers, constants, and so forth). The meanings associated with the leaves are then percolated back up to the root, where the meaning of the whole program is determined.

This structured approach has problems with statements such as **break** or **goto** that don't readily fit the composition model. Further, it forces values to percolate throughout the whole tree, even if this action is unnecessary. Consider, for example, a **stop** statement. When **stop** is executed, I would like to discontinue statement evaluation and immediately return to the main program production, where the final result (the output file) is produced. But I can't; the meaning of **stop** must be composed with that of the remaining statements (even though **stop** means one must ignore the remaining statements!). As it stands, my definition of the meaning of a statement sequence (see lines 90–91 in Figure 10.62, page 345) checks for error on the first statement before evaluating the second. I could add another sort of value, like ⊥, that indicates that execution should stop, even though there is no error. This device would work but would be rather clumsy, as I would model **stop** not by stopping but by continuing to traverse program statements while ignoring them.

Continuations were invented to remedy these problems. A **continuation** is a function passed as a parameter to every semantic function. The semantic function determines its value as usual and then calls (directly or indirectly) the continuation with its value as a parameter. This approach is quite clever but is much less intuitive than the structured approach I have presented so far.

I will first consider expression continuations, which have a semantic domain EC, defined as:

$$EC = (N \oplus Bool) \to State \to R$$

The expression continuation takes a value and a state (since side effects in

evaluating the expression can change the state) and produces a result. The E semantic function will now include an expression continuation as a parameter:

$$E: Exp \to UU \to State \to EC \to R$$

E now produces a result rather than a state-result pair because state changes are included in the continuation component. Figure 10.69 now redefines the meaning of simple integer-valued bit strings. The expression continuation, k, uses the value and state computed by the semantic function E.

Figure 10.69 **Semantic functions**

```
E[0] u s k = k(0, s)                                                    2
E[1] u s k = k(1, s)                                                    3
E[Seq 0] u s k = E[Seq] u s k1                                          4
      where k1(r, t) = range(2 × r)?{⊥} ⇒ ⊥, k(2 × r, t).                5
E[Seq 1] u s k = E[Seq] u s k1                                          6
      where k1(r, t) = range(2 × r + 1)?{⊥} ⇒ ⊥, k(2 × r + 1, t).        7
```

It is no longer necessary to test if a construct produces ⊥; if it does, the construct returns ⊥ immediately. Otherwise, it calls its continuation parameter with values it knows to be valid. To see how evaluation proceeds, consider the following example. Evaluate $E[111]u_e\ s_0\ K$, where u_e and s_0 are the empty environment and initial state, and $K(r,s)=r$ returns the final result.

1. $E[111]u_e\ s_0\ K = E[11]u_e\ s_0\ k_1$, where
   $k_1(r_1, s_1) = range(2 \times r_1 + 1)?\{\bot\} \Rightarrow \bot, K(2 \times r_1 + 1, s_1)$.
2. $E[11]u_e\ s_0\ k_1 = E[1]u_e\ s_0\ k_2$, where
   $k_2(r_2, s_2) = range(2 \times r_2 + 1)?\{\bot\} \Rightarrow \bot, k_1(2 \times r_2 + 1, s_2)$.
3. $E[1]u_e\ s_0\ k_2 = k_2(1, s_0) = range(2 \times 1 + 1)?\{\bot\} \Rightarrow \bot, k_1(2 \times 1 + 1, s_0) =$
   $k_1(3, s_0) = range(2 \times 3 + 1)?\{\bot\} \Rightarrow \bot, K(2 \times 3 + 1, s_0) = K(7, s_0) = 7$.

Figure 10.70 shows how the binary operators are handled.

Figure 10.70 **Semantic functions**

```
E[T1 + T2] u s k = E[T1] u s k1                                         2
      where k1(r1, s1) = r1?N ⇒ E[T2] u s1 k2, ⊥;                       3
      k2(r2, s2) = r2?N ∧ range(r1 + r2)?N ⇒ k(r1 + r2, s2), ⊥.         4
```

Consider this example: Compute $E[22 + 33]u_e\ s_0\ K$, where again $K(r, s) = r$.

1. $E[22 + 33]u_e\ s_0\ K = E[22]u_e\ s_0\ k_1$, where
   $k_1(r_1, s_1) = r_1?N \Rightarrow E[33]\ u\ s_1\ k_2$, $k_2(r_2, s_2) = r_2?N$ and
   $range(r_1 + r_2)?N \Rightarrow k(r_1 + r_2, s_2), \bot$.
2. $E[22]u_e\ s_0\ k_1 = k_1(22, s_0) = 22?N \Rightarrow E[33]u_e\ s_0\ k_2, \bot =$
   $E[33]u_e\ s_0\ k_2 = k_2(33, s_0) = 33?N$ and
   $range(22 + 33)?N \Rightarrow K(22 + 33, s_0), \bot = K(55, s_0) = 55$.

The rest of the binary operators are similar in form, as shown in Figure 10.71.

Figure 10.71 **Semantic functions**

$$
\begin{array}{l}
\mathsf{E}[\![\mathsf{T}_1 - \mathsf{T}_2]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k} = \mathsf{E}[\![\mathsf{T}_1]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k}_1 \\
\quad \textbf{where}\ \ \mathsf{k}_1(\mathsf{r}_1, \mathsf{s}_1) = \mathsf{r}_1?\mathsf{N} \Rightarrow \mathsf{E}[\![\mathsf{T}_2]\!]\ \mathsf{u}\ \mathsf{s}_1\ \mathsf{k}_2, \bot; \\
\quad\quad \mathsf{k}_2(\mathsf{r}_2, \mathsf{s}_2) = \mathsf{r}_2?\mathsf{N} \wedge \mathsf{range}(\mathsf{r}_1 - \mathsf{r}_2)?\mathsf{N} \Rightarrow \mathsf{k}(\mathsf{r}_1 - \mathsf{r}_2, \mathsf{s}_2), \bot. \\
\mathsf{E}[\![\mathsf{T}_1 * \mathsf{T}_2]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k} = \mathsf{E}[\![\mathsf{T}_1]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k}_1 \\
\quad \textbf{where}\ \ \mathsf{k}_1(\mathsf{r}_1, \mathsf{s}_1) = \mathsf{r}_1?\mathsf{N} \Rightarrow \mathsf{E}[\![\mathsf{T}_2]\!]\ \mathsf{u}\ \mathsf{s}_1\ \mathsf{k}_2, \bot; \\
\quad\quad \mathsf{k}_2(\mathsf{r}_2, \mathsf{s}_2) = \mathsf{r}_2?\mathsf{N} \wedge \mathsf{range}(\mathsf{r}_1 \times \mathsf{r}_2)?\mathsf{N} \Rightarrow \mathsf{k}(\mathsf{r}_1 \times \mathsf{r}_2, \mathsf{s}_2), \bot. \\
\mathsf{E}[\![\mathsf{T}_1 / \mathsf{T}_2]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k} = \mathsf{E}[\![\mathsf{T}_1]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k}_1 \\
\quad \textbf{where}\ \ \mathsf{k}_1(\mathsf{r}_1, \mathsf{s}_1) = \mathsf{r}_1?\mathsf{N} \Rightarrow \mathsf{E}[\![\mathsf{T}_2]\!]\ \mathsf{u}\ \mathsf{s}_1\ \mathsf{k}_2, \bot; \\
\quad\quad \mathsf{k}_2(\mathsf{r}_2, \mathsf{s}_2) = \mathsf{r}_2?\mathsf{N} \wedge \mathsf{r}_2 \neq 0 \wedge \mathsf{range}(\mathsf{r}_1 / \mathsf{r}_2)?\mathsf{N} \Rightarrow \mathsf{k}(\mathsf{r}_1 / \mathsf{r}_2, \mathsf{s}_2), \bot. \\
\mathsf{E}[\![\mathsf{T}_1 = \mathsf{T}_2]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k} = \mathsf{E}[\![\mathsf{T}_1]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k}_1 \\
\quad \textbf{where}\ \ \mathsf{k}_1(\mathsf{r}_1, \mathsf{s}_1) = \mathsf{E}[\![\mathsf{T}_2]\!]\ \mathsf{u}\ \mathsf{s}_1\ \mathsf{k}_2; \\
\quad\quad \mathsf{k}_2(\mathsf{r}_2, \mathsf{s}_2) = (\mathsf{r}_1?\mathsf{N} \wedge \mathsf{r}_2?\mathsf{N}) \vee (\mathsf{r}_1?\mathsf{Bool} \wedge \mathsf{r}_2?\mathsf{Bool}) \Rightarrow \\
\quad\quad\quad \mathsf{k}(\mathsf{r}_1 = \mathsf{r}_2, \mathsf{s}_2), \bot.
\end{array}
$$

Identifier lookup is straightforward; see Figure 10.72.

Figure 10.72

$$
\begin{array}{l}
\mathsf{E}[\![\mathsf{I}]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k} = \mathsf{v}?(\{\bot\} \oplus \{\mathsf{redef}\} \oplus \{\mathsf{udef}\}) \Rightarrow \bot, \\
\quad \mathsf{v}?\mathsf{Loc} \Rightarrow (\mathsf{m}(\mathsf{v})?(\{\mathsf{uninitInt}\} \oplus \{\mathsf{uninitBool}\}) \Rightarrow \\
\quad \bot, \mathsf{k}(\mathsf{m}(\mathsf{v}), \mathsf{s})), \mathsf{k}(\mathsf{v}, \mathsf{s}) \\
\quad \textbf{where}\ \ \mathsf{v} = \mathsf{Find}(\mathsf{u})[\![\mathsf{I}]\!];\ \mathsf{s} = <\mathsf{m}, \mathsf{i}, \mathsf{o}>.
\end{array}
$$

To see how side effects are handled, I will introduce an assignment expression similar to that found in C: $\mathsf{I} \leftarrow \mathsf{T}$ is an expression that evaluates to T and (as a side effect) sets I to T, as in Figure 10.73.

Figure 10.73

$$
\begin{array}{l}
\mathsf{E}[\![\mathsf{I} \leftarrow \mathsf{T}]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k} = \mathsf{E}[\![\mathsf{T}]\!]\ \mathsf{u}\ \mathsf{s}\ \mathsf{k}_1 \\
\quad \textbf{where}\ \ \mathsf{k}_1(\mathsf{r}, \mathsf{t}) = \mathsf{v}?\mathsf{Loc} \wedge \\
\quad\quad ((\mathsf{r}?\mathsf{N} \wedge \mathsf{m}(\mathsf{v})?\mathsf{N} \oplus \{\mathsf{uninitInt}\}) \vee \\
\quad\quad (\mathsf{r}?\mathsf{Bool} \wedge \mathsf{m}(\mathsf{v})?\mathsf{Bool} \oplus \{\mathsf{uninitBool}\})) \Rightarrow \\
\quad\quad \mathsf{k}(\mathsf{r}, <\mathsf{m}[\mathsf{v} \leftarrow \mathsf{r}], \mathsf{i}, \mathsf{o}>), \bot. \\
\quad \mathsf{t} = <\mathsf{m}, \mathsf{i}, \mathsf{o}>;\ \mathsf{v} = \mathsf{Find}(\mathsf{u})[\![\mathsf{I}]\!].
\end{array}
$$

Consider this example: Compute $\mathsf{E}[\![\mathsf{I} + \mathsf{I} \leftarrow 0]\!]\mathsf{u}_0\ \mathsf{s}_0\ \mathsf{K}$, where u_0 and s_0 contain a variable I with value 10 and $\mathsf{K}(\mathsf{r}, \mathsf{s}) = \mathsf{r} + \mathsf{Hd}(\mathsf{s})(\mathsf{u}_0[\![\mathsf{I}]\!])$ adds the final value of I to the value of the expression.

1. $\mathsf{E}[\![\mathsf{I} + \mathsf{I} \leftarrow 0]\!]\mathsf{u}_0\ \mathsf{s}_0\ \mathsf{K} = \mathsf{E}[\![\mathsf{I}]\!]\mathsf{u}_0\ \mathsf{s}_0\ \mathsf{k}_1$, where
   $\mathsf{k}_1(\mathsf{r}_1, \mathsf{s}_1) = \mathsf{r}_1?\mathsf{N} \Rightarrow \mathsf{E}[\![\mathsf{I} \leftarrow 0]\!]\ \mathsf{u}\ \mathsf{s}_1\ \mathsf{k}_2, \bot.\ \mathsf{k}_2(\mathsf{r}_2, \mathsf{s}_2) = \mathsf{r}_2?\mathsf{N}.$
   $\mathsf{range}(\mathsf{r}_1 + \mathsf{r}_2)?\mathsf{N} \Rightarrow \mathsf{K}(\mathsf{r}_1 + \mathsf{r}_2, \mathsf{s}_2), \bot.$
2. $\mathsf{E}[\![\mathsf{I}]\!]\mathsf{u}_0\ \mathsf{s}_0\ \mathsf{k}_1 = \mathsf{v}?(\{\bot\} \oplus \{\mathsf{redef}\} \oplus \{\mathsf{udef}\}) \Rightarrow \bot.$
   $\mathsf{v}?\mathsf{Loc} \Rightarrow (\mathsf{m}(\mathsf{v})?(\{\mathsf{uninitInt}\} \oplus \{\mathsf{uninitBool}\}) \Rightarrow \bot, \mathsf{k}_1(\mathsf{m}(\mathsf{v}), \mathsf{s}_0)), \mathsf{k}_1(\mathsf{v}, \mathsf{s}_0),$
   where $\mathsf{v} = \mathsf{Find}(\mathsf{u}_0)[\![\mathsf{I}]\!],\ \mathsf{s}_0 = <\mathsf{m}, \mathsf{i}, \mathsf{o}>.$
3. $\mathsf{E}[\![\mathsf{I}]\!]\mathsf{u}_0\ \mathsf{s}_0\ \mathsf{k}_1 = \mathsf{k}_1(\mathsf{m}(\mathsf{v}), \mathsf{s}_0) = \mathsf{k}_1(10, \mathsf{s}_0) = 10?\mathsf{N} \Rightarrow$
   $\mathsf{E}[\![\mathsf{I} \leftarrow 0]\!]\ \mathsf{u}\ \mathsf{s}_0\ \mathsf{k}_2, \bot = \mathsf{E}[\![\mathsf{I} \leftarrow 0]\!]\ \mathsf{u}\ \mathsf{s}_0\ \mathsf{k}_2.$
4. $\mathsf{E}[\![\mathsf{I} \leftarrow 0]\!]\mathsf{u}_0\ \mathsf{s}_0\ \mathsf{k}_2 = \mathsf{E}[\![0]\!]\mathsf{u}_0\ \mathsf{s}_0\ \mathsf{k}_3$, where
   $\mathsf{k}_3(\mathsf{r}, \mathsf{t}) = \mathsf{v}?\mathsf{Loc} \wedge ((\mathsf{r}?\mathsf{N} \wedge \mathsf{m}(\mathsf{v})?\mathsf{N} \oplus \{\mathsf{uninitInt}\}) \vee$
   $(\mathsf{r}?\mathsf{Bool} \wedge \mathsf{m}(\mathsf{v})?\mathsf{Bool} \oplus \{\mathsf{uninitBool}\})) \Rightarrow \mathsf{k}_2(\mathsf{r}, <\mathsf{m}[\mathsf{v} \leftarrow \mathsf{r}], \mathsf{i}, \mathsf{o}>), \bot.$
   $\mathsf{t} = <\mathsf{m}, \mathsf{i}, \mathsf{o}>, \mathsf{v} = \mathsf{Find}(\mathsf{u})[\![\mathsf{I}]\!].$

5. $E[0]\ u\ s_0\ k_3 = k_3(0, s_0) = v?Loc \wedge ((0?N \wedge m(v)?N \oplus \{uninitInt\}) \vee (0?Bool \wedge m(v)?Bool \oplus \{uninitBool\})) \Rightarrow k_2(0, < m[v \leftarrow 0], i, o >), \perp.$
   $s_0 = < m, i, o > . v = Find(u)[I].$
6. $E[0]\ u\ s_0\ k_3 = k_2(0, < m[v \leftarrow 0], i, o >) = k_2(0, ss_0)$, where
   $ss_0 = < m[v \leftarrow 0], i, o > . \ k_2(0, ss_0) = 0?N.$
   $range(10 + 0)?N \Rightarrow K(0 + 10, ss_0), \perp = K(0 + 10, ss_0).$
7. $K(10, ss_0) = 10 + Hd(ss_0)(u_0[I]) = 10 + 0 = 10.$

Continuations execute in the state that is current when they are evaluated, not in the state that is current when they are defined (that's why they take state as a parameter).

3.20 Statement Continuations

I am now ready to consider statement continuations, which are particularly useful because they allow me to handle nonstructured control flows. I will first define SC, the semantic domain of statement continuations (see Figure 10.74). I will also slightly alter EC, the semantic domain of expression continuations. In both cases, the continuations will return Ans, the domain of program answers, reflecting the fact that expressions and statements are not executed in isolation, but rather in contexts in which they contribute to the final answer to be computed by the whole program.

Figure 10.74 **Semantic domains** 1

$Ans = File \oplus \{\perp\}$ 2
$EC = (N \oplus Bool) \rightarrow State \rightarrow Ans$ 3
$SC = State \rightarrow Ans$ 4

Statement continuations take only one parameter because the only program component updated by a statement is the state. Figure 10.75 extends the S semantic function to include a statement continuation parameter. All semantic functions now return Ans because they all execute by evaluating (directly or indirectly) some continuation function. The values that change during the computation of a semantic function (a result, environment, or state) are now parameters to a continuation function.

Figure 10.75 **Semantic functions** 1

$E: Exp \rightarrow UU \rightarrow State \rightarrow EC \rightarrow Ans$ 2
$S: Stm \rightarrow UU \rightarrow State \rightarrow SC \rightarrow Ans$ 3

To see the utility of statement continuations, consider the definition of statement composition in Figure 10.76.

Figure 10.76 $S[St_1\ St_2]\ u\ s\ c = S[St_1]\ u\ s\ c'$ 1
where $c'(s') = S[St_2]\ u\ s'c.$ 2

The statement continuation has a fairly intuitive interpretation: what to execute after the current statement. The advantage of the continuation ap-

proach is now evident. A statement need not execute its continuation if an abnormal transfer of control is indicated. A **stop** statement executes by returning an answer (the current value of the output file). Similarly, **goto** executes by looking up (and executing) a statement continuation stored in the environment as the value of the label!

I can now consider other statements, as shown in Figure 10.77.

Figure 10.77

```
S[ε] u s c = c(s)                                                              1
S[I:= T] u s c = E[T] u s k                                                    2
      where k(r, t) = v?Loc ∧                                                  3
            ((r?N ∧ m(v)?N ⊕ {uninitInt}) ∨                                    4
            (r?Bool ∧ m(v)?Bool ⊕ uninitBool)) ⇒                               5
            c(< m[v ← r], i, o >), ⊥;                                          6
            t = < m, i, o > ; v = Find(u)[I].                                  7
S[read I] u s c = v?Loc ∧ i ≠ eof ∧                                            8
      ((Hd(i)?N ∧ m(v)?N ⊕ uninitInt) ∨                                        9
      (Hd(i)?Bool ∧ m(v)?Bool ⊕ uninitBool)) ⇒                                 10
      c(< m[v ← Hd(i)], Tl(i), o >), ⊥                                         11
      where s = < m, i, o > ; v = Find(u)[I].                                  12
S[write T] u s c = E[T] u s k                                                  13
      where k(r, t) = c(< m, i, append(o, < r, eof >) >); t = < m, i, o > .    14
S[if T then St1 else St2] u s c =                                              15
      E[T] u s k                                                               16
      where k(r, t) = r?Bool ⇒ (r ⇒ S[St1] u t c, S[St2] u t c), ⊥.            17
S[do T times St] u s c = E[T] u s k                                             18
      where k(r, t) = r?N ⇒ v_m(t), ⊥;                                         19
      m = max(0, r); v_0(s′) = c(s′); v_{i+1}(s′) = S[St] u s′v_i .            20
S[while T do St] u s c = lim_{i→∞} p_i(s)                                      21
      where p_0(s′) = ⊥; p_{i+1}(s′) = E[T] u s′ k_{i+1};                      22
      k_{i+1}(r, t) = r?Bool ⇒ (r ⇒ S[St] u t p_i, c(t)), ⊥                    23
```

3.21 Declaration Continuations

A declaration continuation will map an environment and state into an answer. The D function will now take a declaration continuation from the domain DC, as in Figure 10.78.

Figure 10.78

```
DC = UU → State → Ans                                                          1
D: Decls → UU → State → DC → Ans                                               2
```

```
D[I: integer] u s d = Hd(u)[I]?{udef} ⇒                                    3
      (l ?{⊥} ⇒ d(u[Hd[I ← ⊥]], s),                                        4
      d(u[Hd[I ← l]], < m[l ← uninitInt], i, o >)),                        5
      d(u[Hd[I ← redef]], s)                                               6
      where s = < m, i, o > ; l = alloc(m).                                7
D[I: Boolean] u s d = Hd(u)[I]?{udef} ⇒                                    8
      (l ?{⊥} ⇒ d(u[Hd[I ← ⊥]], s),                                        9
      d(u[Hd[I ← l]], < m[l ← uninitBool], i, o >)),                       10
      d(u[Hd[I ← redef]], s)                                               11
      where s = < m, i, o > ; l = alloc(m).                                12
D[I = T] u s d = E[T] u s k                                                13
      where k(r, t) = Hd(u)[I]?{udef} ⇒                                    14
            d(u[Hd[I ← r]], t), d(u[Hd[I ← redef]], t).                     15
```

The expression T (line 13) can be allowed to contain function calls. If evaluation of T faults, E will simply return ⊥; otherwise, it executes the declaration continuation (d) with the value that T returns and a possibly updated state. Other definitions are given in Figure 10.79.

Figure 10.79

```
D[ε] u s d = d(u, s)                                                       1
D[Def1 Def2] u s d = D[Def1] u s d′                                        2
      where d′(v, t) = D[Def2] v t d.                                      3

S[begin Def St end] u s c = D[Def] < ue, u > s d                           4
      where d(v, t) = S[St] v t c′;                                        5
      c′(t′) = c(t′[Hd[Free(Hd(s)) ← unalloc]]).                           6
```

3.22 Procedures, Functions, and Parameters

I now define routines and parameters in the new continuation notation. First, declarations need to be handled, using D, DC, FP, and FC (formal parameter continuation), as shown in Figure 10.80.

Figure 10.80

```
FC = Parms → Ans                                                           1

FP: Fparms → Parms → FC → Ans                                              2

FP[I : integer] p f = f(append(p, << 0, I > , eol >))                      3
FP[I : Boolean] p f = f(append(p, << false, I > , eol >))                  4
FP[ε] p f = f(p)                                                           5
FP[Formals1 Formals2] p f = FP[Formals1] p f′                              6
      where f′(p′) = FP[Formals2]p′f.                                      7
```

Procedures are generalizations of statements, and, like all statements, take a statement continuation as a parameter. This continuation is essentially the return point of the procedure; see Figure 10.81.

Figure 10.81

```
Proc = (U → State → SC → Ans) ⊗ Loc ⊗ Parms                                  1

D[procedure I (Formals); Def St] u s d =                                     2
      FP[Formals]eol f,                                                      3
      where  f(p) = Hd(u)[I]?{udef} ⇒                                        4
            (l?{⊥} ⇒ d(u[Hd[I ← ⊥]], s),                                     5
            d(uu, < m[l ← uninitInt], i, o >)),                              6
            d(u[Hd[I ← redef]], s);                                          7
      s = < m, i, o > ; l = alloc(m); uu = u[Hd[I ←< r, l, p >]];            8
      r(v, t, cc) = D[Def] < v, uu > t d′;                                   9
      d′(v′, t′) = S[St]v′t′c′;                                              10
      c′(tt) = cc(tt[Hd[Free(t) ← unalloc]]).                                11
```

Since functions are a generalization of expressions, they will now include an expression continuation that represents the mechanism through which the function's value is returned, as in Figure 10.82.

Figure 10.82

```
Func = (U → State → EC → Ans) ⊗ Loc ⊗ Parms                                  1

D[Integer function(Formals) I; St; return(T)] u s d =                        2
      FP[Formals]eol f,                                                      3
      where  f(p) = Hd(u)[I]?{udef} ⇒                                        4
            (l?{⊥} ⇒ d(u[Hd[I ← ⊥]], s),                                     5
            d(uu, < m[l ← uninitInt], i, o >)),                              6
            d(u[Hd[I ← redef]], s);                                          7
      s = < m, i, o > ; l = alloc(m); uu = u[Hd[I ←< r, l, p >]];            8
      r(u′, s′, ec) = D[Def] < u′, uu > s′d′;                                9
      d′(v, t) = S[St] v t c;                                                10
      c(v′, t′) = E[T]v′t′k;                                                 11
      k(r, tt) = r?N ⇒ ec(r, tt[Hd[Free(s′) ← unalloc]]), ⊥.                 12
D[Boolean function(Formals) I; St; return(T)] u s d =                        13
      FP[Formals]eol f,                                                      14
      where  f(p) = Hd(u)[I]?{udef} ⇒                                        15
            (l?{⊥} ⇒ d(u[Hd[I ← ⊥]], s),                                     16
            d(uu, < m[l ← uninitInt], i, o >)),                              17
            d(u[Hd[I ← redef]], s);                                          18
      s = < m, i, o > ; l = alloc(m); uu = u[Hd[I ←< r, l, p >]];            19
      r(u′, s′, ec) = D[Def] < u′, uu > s′ d′;                               20
      d′(v, t) = S[St] v t c;                                                21
      c(v′, t′) = E[T]v′ t′ k;                                               22
      k(r, tt) = r?Bool ⇒ ec(r, tt[Hd[Free(s′) ← unalloc]]), ⊥.              23
```

It is time to consider actual-parameter evaluation and procedure and function calls; see Figure 10.83. AC is the semantic domain of actual-parameter continuations.

Figure 10.83

```
AC = UU → Parms → Ans                                                    1

AP: Aparms → UU → Parms → State → AC → Ans                               2

AP[I] u p s a = v?Loc ∧ p ≠ eol ∧                                        3
      ((Hd(pp)?N ∧ m(v)?N ⊕ uninitInt) ∨                                 4
      (Hd(pp)?Bool ∧ m(v)?Bool ⊕ uninitBool)) ⇒                          5
      a(u[Hd[Tl(pp) ← v]], Tl(p)), ⊥                                     6
      where v = Find(Tl(u))[I]; pp = Hd(p); s = < m, i, o > .            7
AP[ε] u p s a = a(u, p)                                                  8
AP[Actuals1 Actuals2] u p s a =                                          9
      AP[Actuals1] u p s a'                                              10
      where a'(u', p') = AP[Actuals2]u' p' s a.                          11

S[call I(Actuals)] u s c = v?Proc ⇒                                      12
      AP[Actuals] < ue, u > r sa, ⊥                                      13
      where a(u', q) = (q = eol) ⇒                                       14
            lim_{i→∞} pi(s, Hd(u')), ⊥;                                  15
      v = Find(u)[I] = < f, l, r > ;                                     16
      p0(s', w) = ⊥;                                                     17
      pi+1(s', w) = m(l) = uninitInt ⇒                                   18
            f w < m[l ← i], I, O > c1,                                   19
            m(l) > 0 ⇒ f w < m[l ← m(l) − 1], I, O > c2, ⊥;             20
      s' =< m, I, O > ;                                                  21
      c1(t1) = c(t1[Hd[l <- uninitInt]]);                                22
      c2(t2) = c(t2[Hd[l <- m(l)]]).                                     23

E[eval I (Actuals)] u s k = v?Func ⇒                                     24
      AP[Actuals] < ue, u > r sa, ⊥                                      25
      where a(u', q) = (q = eol) ⇒                                       26
            lim_{i→∞} pi(s, Hd(u')), ⊥;                                  27
      v = Find(u)[I] = < f, l, r > ;                                     28
      p0(s', w) = ⊥;                                                     29
      pi+1(s', w) = m(l) = uninitInt ⇒                                   30
            f w < m[l ← i], I, O > k1,                                   31
            m(l) > 0 ⇒ f w < m[l ← m(l) − 1], I, O > k2, ⊥;             32
      s' = < m, I, O > ;                                                 33
      k1(r1, t1) = k(r1, t1[Hd[l ← uninitInt]]);                         34
      k2(r2, t2) = k(r1, t2[Hd[l ← m(l)]]).                              35
```

Finally, I redefine the M function using continuations, as shown in Figure 10.84.

Figure 10.84

```
M: Pr → File → Ans                                                    1

M[program Def St end] i =                                             2
      D[Def] < u_e, u_0 > < m_0, i, eof > d                           3
      where d(v, t) = S[St] v tc; c(t') = Tl(Tl(t')).                 4
```

3.23 Flow of Control

Now that I have the machinery of continuations in place, I can illustrate how to implement statements that alter the flow of control. I begin with the **stop** statement, which forces immediate termination of execution. Figure 10.85 shows the semantic function.

Figure 10.85

```
S[stop] u s c = Tl(Tl(s))
```

Stop returns the output file component of the current state. It avoids the normal flow of control by ignoring its continuation parameter.

A more interesting illustration is **break**, which I will use to exit any of the structured statements in the language (**if**, **do**, **while**, **begin-end**). I will let any of these statements be optionally labeled with an identifier, which will follow normal scoping rules. I define **break** I to cause execution to immediately break out of the structure labeled with I and then to continue execution with the normal successor to the labeled statement. If I isn't declared as a label in the scope of the break, the statement produces an error value.

I extend V, the domain of environment contents, to include statement continuations:

```
U = Id → V
V = N ⊕ Bool ⊕ Loc ⊕ Proc ⊕ Func ⊕ SC ⊕ {⊥}⊕{udef}⊕{redef}
```

The meaning of a label on a structured statement will be the continuation associated with that statement. Figure 10.86 adds definitions for structured statements with labels (the definitions for unlabeled statements are, of course, retained).

Figure 10.86

```
S[I: if T thenSt_1 else St_2] u s c = E[T] uu s k                     1
      where k(r, t) = r?Bool ⇒ (r ⇒ S[St_1] uu t c, S[St_2] uu t c), ⊥;  2
      uu = Hd(u)[I]?{udef} ⇒ u[Hd[I ← c']], u[Hd[I ← redef]];          3
      c'(t') = c(t'[Hd[Free(Hd(s)) ← unalloc]]).                       4
S[I: do T times St] u s c = E[T] uu s k                               5
      where k(r, t) = r?N ⇒ v_m(t), ⊥;                                6
      m = max(0, t); v_0(s') = c(s');                                 7
      v_{i+1}(s') = S[St] uu s' v_i;                                  8
      uu = Hd(u)[I]?{udef} ⇒ u[Hd[I ← c']], u[Hd[I ← redef]];          9
      c'(t') = c(t'[Hd[Free(Hd(s)) ← unalloc]]).                      10
S[I: while T do St] u s c = lim_{i→∞} p_i(s)                          11
      where p_0(s') = ⊥; p_{i+1}(s') = E[T] uu s' k_{i+1};            12
      k_{i+1}(r, t) = r?Bool ⇒ (r ⇒ S[St] uu t p_i, c(t)), ⊥;         13
      uu = Hd(u)[I]?{udef} ⇒ u[Hd[I ← c']], u[Hd[I ← redef]];         14
      c'(t') = c(t'[Hd[Free(Hd(s)) ← unalloc]]);                      15
```

```
S[I: begin Def St end] u s c =                                          16
      D[Def] < u_e, uu > s d,                                           17
      where d(v, t) = S[St] v t c';                                     18
      c'(t') = c(t'[Hd[Free(Hd(s)) ← unalloc]]);                        19
      uu = Hd(u)[I]?{udef} ⇒ u[Hd[I ← c']], u[Hd[I ← redef]].           20
```

Break looks up its identifier, and if it is bound to a statement continuation, it executes that continuation in the current state; see Figure 10.87.

Figure 10.87

```
S[break I] u s c = v?SC ⇒ v(s), ⊥                                       1
      where v = Find(u)[I].                                             2
```

3.24 Summary of Syntactic and Semantic Domains and Semantic Functions

The domains used in the denotational definitions in this chapter have been upgraded during the progression of examples. Figure 10.88 lists the most recent meanings.

Figure 10.88

```
Syntactic domains                                                       1

      BinLit: binary literals; nonterminals BN, Seq                     2
      Exp: expressions; nonterminal T                                   3
      Id: identifiers; nonterminal I                                    4
      Pr: programs; nonterminal P                                       5
      Decls: declarations; nonterminal Def                              6
      Stm: statements; nonterminal St                                   7

Semantic domains                                                        8

      Basic:                                                            9
            N = {0, 1, 2, ...} (natural numbers)                        10
            Bool = {false, true} (Boolean values)                       11

      Complex:                                                          12
            Loc = {0, 1, ···} -- finite domain of memory locations      13
            Mem = Loc → N ⊕ Bool ⊕ {uninitInt} ⊕ {uninitBool} ⊕         14
                  {unalloc} -- memory location                          15
            File = (N ⊕ Bool ⊕ {eof})* -- contents of a file            16
            R = N ⊕ Bool ⊕ {⊥} -- value of an expression                17
            RR = State ⊗ (N ⊕ Bool ⊕ {⊥})                               18
                  -- result of function                                19
            State = Mem ⊗ File ⊗ File -- program state                  20
            Ans = File ⊕ {⊥} -- program result                         21
            V = N ⊕ Bool ⊕ Loc ⊕ Proc ⊕ Func ⊕ SC ⊕ {⊥} ⊕ {udef} ⊕      22
                  {redef} -- value of an identifier                     23
            U = Id → V -- environment                                   24
            UU = U* -- sequence of environments                         25
            Proc = (U → State → SC → Ans) ⊗ Loc ⊗ Parms                 26
                  -- procedure                                          27
            Func = (U → State → EC → Ans) ⊗ Loc ⊗ Parms                 28
                  -- function                                           29
```

```
Parms = ((N ⊕ Bool) ⊗ Id ⊕ eol)* -- parameters                30
SC = State → Ans -- statement continuation                    31
EC = (N ⊕ Bool) → State → Ans -- expression contin.           32
DC = UU → State → Ans -- declaration continuation             33
AC = UU → Parms → Ans -- actual parameter contin.             34
FC = Parms → Ans -- formal parameter continuation             35
```

Semantic functions 36

```
L: Id → V -- lookup                                           37
E: Exp → UU → State → EC → Ans -- expression                  38
S: Stm → UU → State → SC → Ans -- statement                   39
D: Decls → UU → State → DC → Ans -- declaration               40
M: Pr → File → Ans -- program                                 41
FP: Fparms → Parms → Parms -- formal parameters               42
AP: Aparms → UU → Parms → State → AC → Ans                    43
      -- actual parameters                                    44
```

4 ◆ FINAL COMMENTS

This long (and somewhat tedious) exercise shows that it is possible to specify exactly what a programming language designer allows in the syntax and means by the constructs of the language. Such a specification can guide the designer (to make sure that all cases are properly covered), the implementer (to make sure that the compiler and runtime support live up to the specifications), and the programmer (to make sure that language constructs are used as intended).

Formal specification can also be used to evaluate the clarity of a language. If the axiomatic semantics of a construct are hard to build and hard to understand, then perhaps the construct itself is hard to understand. For example, a multiple assignment statement has this structure:

```
x, y, z := 13, 16, x + 3;
```

Three assignments are made simultaneously. However, `x + 3` on the right-hand side depends on `x`, which is on the left-hand side. The order of evaluation makes a difference. It is not easy in axiomatic semantics to specify the rule for multiple assignment for this reason. Perhaps that complexity is a symptom that multiple assignment is itself an unclear concept.

As my brief forays into ML have shown, the specification can even be written in a programming language so that it can be checked for syntax and meaning. (Have you really read all the specifications? Did you find any mistakes?) Such a specification can even be used to interpret programs (written in abstract syntax, of course), more as a way of debugging the specification than understanding the meaning of the programs.

However, the fact that the specification is in a language, albeit a programming language, seems to reduce the question of formally specifying one language (the target) to specifying another (ML, for example). It requires that someone who wants to understand the target language specification needs to learn and understand some fairly complex notions, such as domain equations.

There is no guarantee that every error case has been dealt with, and the notation is complex enough that such an omission would probably pass unnoticed.

The fact that I have succeeded in denoting standard features of a programming language gives me no particular confidence that I could handle such constructs as CLU coroutines (Chapter 2), ML higher-level functions (Chapter 3), Prolog resolution (Chapter 8), Post guardians (Chapter 6), SNOBOL patterns (Chapter 9), or Modula monitors (Chapter 7). An enormous amount of cleverness is required to build denotational semantic definitions. As you have seen, introducing a single concept into a language is likely to modify the definitions for everything else. A typical modification involves making functions like E even higher-order. The result is anything but straightforward.

Several excellent books deal with programming language semantics. I can especially recommend Tennent [Tennent 81] and Pagan [Pagan 81].

EXERCISES

Review Exercises

10.1 Describe the language (that is, the set of strings) generated by this BNF grammar:

```
S ::= ( S ) S | ε
```

10.2 Show a different BNF grammar that generates exactly the same language as the grammar in Exercise 10.1.

10.3 Write BNF productions for **if** statements.

10.4 An **ambiguous grammar** is one that generates strings that have more than one parse tree. Is the grammar of Figure 10.89 ambiguous? Does it have any other problems?

Figure 10.89

```
Expression ::=                                    1
     Expression + Expression |                    2
     Expression * Expression |                    3
     INTEGER                                      4
```

10.5 Prove the program in Figure 10.90 correct.

Figure 10.90

```
{a < 3}                                           1
if a < 4 then x := 2 else x := 10 end;            2
{x = 2}                                           3
```

10.6 Compute:

```
wp(if a < 4 then x := 2 else x := 10 end, x = 2)
```

Challenge Exercises

10.7 In Figure 10.43 (page 330), I specify that redefinition of an identifier has no effect. Show how to modify the example so that redefinition hides the previous definition.

10.8 In line 25 of Figure 10.46 (page 333), why check that `c` is a member of `U`? What else could it be?

10.9 On page 331, I introduce `redef`. Why not just use `udef` for this purpose?

10.10 How would you code the semantics of a **`while`** loop (see Figure 10.52, page 337) in ML?