# Chapter 9
# DENOTATIONAL SEMANTICS

With formal semantics we give programs meaning by mapping them into some abstract but precise domain of objects. Using denotational semantics, we provide meaning in terms of mathematical objects, such as integers, truth values, tuples of values, and functions. For this reason, denotational semantics was originally called mathematical semantics.

Christopher Strachey and his Programming Research Group at Oxford developed denotational semantics in the mid 1960s; Dana Scott supplied the mathematical foundations in 1969. Although originally intended as a mechanism for the analysis of programming languages, denotational semantics has become a powerful tool for language design and implementation.

In this chapter we take a careful look at denotational semantics. We illustrate the methodology by specifying the language of a simple calculator and three programming languages: (1) Wren, (2) a language with procedures called Pelican, and (3) a language with goto's called Gull. The presentation differs from some of the literature on denotational semantics in that we enhance the readability of the specifications by avoiding the Greek alphabet and single character identifiers found in the traditional presentations.

## 9.1 CONCEPTS AND EXAMPLES

Denotational semantics is based on the recognition that programs and the objects they manipulate are symbolic realizations of abstract mathematical objects, for example,

strings of digits realize numbers, and

function subprograms realize (approximate) mathematical functions.

The exact meaning of "approximate" as used here will be made clear in Chapter 10. The idea of denotational semantics is to associate an appropriate mathematical object, such as a number, a tuple, or a function, with each phrase of the language. The phrase is said to **denote** the mathematical object, and the object is called the **denotation** of the phrase.

Syntactically, a phrase in a programming language is defined in terms of its constituent parts by its BNF specification. The decomposition of language phrases into their subphrases is reflected in the abstract syntax of the programming language as well. A fundamental principle of denotational semantics is that the definition be **compositional**. That means the denotation of a language construct is defined in terms of the denotations of its subphrases. Later we discuss reasons for having compositional definitions.

Traditionally, denotational definitions use special brackets, the emphatic brackets ⟦ ⟧, to separate the syntactic world from the semantic world. If p is a syntactic phrase in a programming language, then a denotational specification of the language will define a mapping *meaning*, so that *meaning* ⟦p⟧ is the denotation of p—namely, an abstract mathematical entity that models the semantics of p.

For example, the expressions "**2*4**", "**(5+3)**", "**008**", and "**8**" are syntactic phrases that all denote the same abstract object, namely the integer 8. Therefore with a denotational definition of expressions we should be able to show that

*meaning* ⟦**2*4**⟧ = *meaning* ⟦**(5+3)**⟧ = *meaning* ⟦**008**⟧ = *meaning* ⟦**8**⟧ = 8.

Functions play a prominent role in denotational semantics, modeling the bindings in stores and environments as well as control abstractions in programming languages. For example, the "program"

fact(n) = if n=0 then 1 else n*fact(n–1)

denotes the factorial function, a mathematical object that can be viewed as the set of ordered pairs,

{ <0,1>, <1,1>, <2,2>, <3,6>, <4,24>, <5,120>, <6,720>, ... },

and a denotational semantics should confirm this relationship.

A denotational specification of a programming language consists of five components, two specifying the syntactic world, one describing the semantic domains, and two defining the functions that map the syntactic objects to the semantic objects.

## The Syntactic World

**Syntactic categories** or **syntactic domains** name collections of syntactic objects that may occur in phrases in the definition of the syntax of the language—for example,

Numeral, Command, and Expression.

Commonly, each syntactic domain has a special metavariable associated with it to stand for elements in the domain—for example,

C : Command

E : Expression

N : Numeral

I : Identifier.

With this traditional notation, the colon means "element of". Subscripts will be used to provide additional instances of the metavariables.

**Abstract production rules** describe the ways that objects from the syntactic categories may be combined in accordance with the BNF definition of the language. They provide the possible patterns that the abstract syntax trees of language phrases may take. These abstract production rules can be defined using the syntactic categories or using the metavariables for elements of the categories as an abbreviation mechanism.

Command ::= **while** Expression **do** Command$^{+}$

E ::= N | I | E O E | – E

These rules are the abstract productions that were discussed in Chapter 1. They do not fully specify the details of syntax with respect to parsing items in the language but simply portray the possible forms of syntactic constructs that have been verified as correct by some other means.

## The Semantic World

**Semantic domains** are "sets" of mathematical objects of a particular form. The sets serving as domains have a lattice-like structure that will be described in Chapter 10. For now we view these semantic domains as normal mathematical sets and structures—for example,

Boolean = { true, false } is the set of truth values,

Integer = { ... , -2, -1, 0, 1, 2, 3, 4, ... } is the set of integers, and

Store = (Variable → Integer) consists of sets of bindings (functions mapping variable names to values).

We use the notation A → B to denote the set of functions with domain A and codomain B.

## The Connection between Syntax and Semantics

**Semantic functions** map objects of the syntactic world into objects in the semantic world. Constructs of the subject language—namely elements of the syntactic domains—are mapped into the semantic domains. These functions are specified by giving their syntax (domain and codomain), called their **signatures**—for example,

*meaning* : Program → Store

*evaluate* : Expression → (Store → Value)

and by using **semantic equations** to specify how the functions act on each pattern in the syntactic definition of the language phrases. For example,

*evaluate* ⟦$E_1$ + $E_2$⟧ sto = *plus*(*evaluate* ⟦$E_1$⟧ sto, *evaluate* ⟦$E_2$⟧ sto)

states that the value of an expression "$E_1$ + $E_2$" is the mathematical sum of the values of its component subexpressions. Note that the value of an expression will depend on the current bindings in the store, here represented by the variable "sto". The function *evaluate* maps syntactic expressions to semantic values—namely, integers—using mathematical operations such as *plus*. We refer to these operations as **auxiliary functions** in the denotational definition.

Figure 9.1 contains a complete denotational specification of a simple language of nonnegative integer numerals. This definition requires two auxiliary functions defined in the semantic world, where Number x Number denotes the Cartesian product.

*plus* : Number x Number → Number

*times* : Number x Number → Number.

---

**Syntactic Domains**

N : Numeral -- nonnegative numerals

D : Digit -- decimal digits

**Abstract Production Rules**

Numeral ::= Digit | Numeral Digit

Digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

**Semantic Domain**

Number = { 0, 1, 2, 3, 4, ... } -- natural numbers

**Semantic Functions**

*value* : Numeral → Number

*digit* : Digit → Number

**Semantic Equations**

*value* ⟦N D⟧ = *plus* (*times*(10, *value* ⟦N⟧), *digit* ⟦D⟧)

*value* ⟦D⟧ = *digit* ⟦D⟧

*digit* ⟦**0**⟧ = 0 *digit* ⟦**3**⟧ = 3 *digit* ⟦**6**⟧ = 6 *digit* ⟦**8**⟧ = 8

*digit* ⟦**1**⟧ = 1 *digit* ⟦**4**⟧ = 4 *digit* ⟦**7**⟧ = 7 *digit* ⟦**9**⟧ = 9

*digit* ⟦**2**⟧ = 2 *digit* ⟦**5**⟧ = 5

---

*Figure 9.1*: A Language of Numerals

We need two syntactic domains for the language of numerals. Phrases in this language are mapped into the mathematical domain of natural numbers. Generally we have one semantic function for each syntactic domain and one semantic equation for each production in the abstract syntax. To distinguish numerals (syntax) from numbers (semantics), different typefaces are employed. Note the compositionality of the definition in that the value of a phrase "N D" is defined in terms of the value of N and the value of D.

As an example of evaluating a numeral according to this denotational definition, we find the value of the numeral **65**:

$$\begin{aligned} \mathit{value}\ [\![\mathbf{65}]\!] &= \mathit{plus}(\mathit{times}(10, \mathit{value}\ [\![\mathbf{6}]\!]), \mathit{digit}\ [\![\mathbf{5}]\!]) \\ &= \mathit{plus}(\mathit{times}(10, \mathit{digit}\ [\![\mathbf{6}]\!]), 5) \\ &= \mathit{plus}(\mathit{times}(10, 6), 5) \\ &= \mathit{plus}(60, 5) = 65 \end{aligned}$$

Solely using the specification of the semantics of numerals, we can easily prove that $\mathit{value}\ [\![\mathbf{008}]\!] = \mathit{value}\ [\![\mathbf{8}]\!]$:

$$\begin{aligned} \mathit{value}\ [\![\mathbf{008}]\!] &= \mathit{plus}(\mathit{times}(10, \mathit{value}\ [\![\mathbf{00}]\!]), \mathit{digit}\ [\![\mathbf{8}]\!]) \\ &= \mathit{plus}(\mathit{times}(10, \mathit{plus}(\mathit{times}(10, \mathit{value}\ [\![\mathbf{0}]\!]), \mathit{digit}\ [\![\mathbf{0}]\!])), 8) \\ &= \mathit{plus}(\mathit{times}(10, \mathit{plus}(\mathit{times}(10, \mathit{digit}\ [\![\mathbf{0}]\!]), 0)), 8) \\ &= \mathit{plus}(\mathit{times}(10, \mathit{plus}(\mathit{times}(10, 0), 0)), 8) \\ &= 8 = \mathit{digit}\ [\![\mathbf{8}]\!] = \mathit{value}\ [\![\mathbf{8}]\!] \end{aligned}$$

Although the syntactic expression "**008**" inside the emphatic brackets is written in linear form, it actually represents the abstract syntax tree shown in Figure 9.2 that reflects its derivation

<numeral> ⇒ <numeral> <digit> ⇒ <numeral> <digit> <digit> ⇒
<digit> <digit> <digit> ⇒ **0** <digit> <digit> ⇒ **0 0** <digit> ⇒ **0 0 8**.

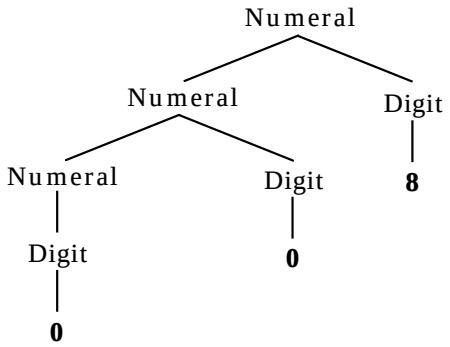


*Figure 9.2*: An Abstract Syntax Tree

The elements of the syntactic world inside of the emphatic brackets are always abstract syntax trees. We write them in a linear form only for convenience. The abstract production rules will be used to describe the abstract syntax trees and the concrete syntax to disambiguate them.

## Compositionality

The principle of compositionality has a long history in mathematics and the specification of languages (see the further readings at the end of this chapter). In his book [Tennent91] on the semantics of programming languages, Tennent suggests three reasons for using compositional definitions:

1. In a denotational definition, each phrase of a language is given a meaning that describes its contribution to the meaning of a complete program that contains it. Furthermore, the meaning of each phrase is formulated as a function of the denotations of its immediate subphrases. As a result, whenever two phrases have the same denotation, one can be replaced by the other without changing the meaning of the program. Therefore a denotational semantics supports the substitution of semantically equivalent phrases.

2. Since a denotational definition parallels the syntactic structure of its BNF specification, properties of constructs in the language can be verified by **structural induction**, the version of mathematical induction introduced in Chapter 8 that follows the syntactic structure of phrases in the language.

3. Compositionality lends a certain elegance to denotational definitions, since the semantic equations are structured by the syntax of the language. Moreover, this structure allows the individual language constructs to be analyzed and evaluated in relative isolation from other features in the language.

As a consequence of compositionality, the semantic function *value* is a homomorphism, which means that the function respects operations. As an illustration, consider a function $H : A \rightarrow B$ where A has a binary operation $f : A \times A \rightarrow A$ and B has a binary operation $g : B \times B \rightarrow B$. The function H is a **homomorphism** if $H(f(x,y)) = g(H(x),H(y))$ for all $x,y \in A$. For the example in Figure 9.1, the operation f is concatenation and $g(m,n) = plus(times(10, m), n)$. Therefore $value(f(x,y)) = g(value(x),value(y))$, which thus demonstrates that *value* is a homomorphism.

## Exercises

1. Using the language of numerals in Figure 9.1, draw abstract syntax trees for the numerals "**5**" and "**6789**".

2. Use the denotational semantics for numerals to derive the value of "**3087**".

3. Define a denotational semantics for the language of numerals in which the meaning of a string of digits is the number of digits in the string.

4. Define a denotational semantics for the language of octal (base 8) numerals. Use the definition to find the value of "**752**".

5. This is a BNF specification (and abstract syntax) of the language of Roman numerals less than five hundred.

   Roman ::= Hundreds Tens Units

   Hundreds ::= ε | **C** | **CC** | **CCC** | **CD**

   Tens ::= LowTens | **XL** | **L** LowTens | **XC**

   LowTens ::= ε | LowTens **X**

   Units ::= LowUnits | **IV** | **V** LowUnits | **IX**

   LowUnits ::= ε | LowUnits **I**

   The language of Roman numerals is subject to context constraints that the number of X's in LowTens and I's in LowUnits can be no more than three. Remember ε represents the empty string.

   Provide semantic functions and semantic equations for a denotational definition of Roman numerals that furnishes the numeric value of each string in the language. Assume that the context constraints have been verified by other means.

## 9.2 A CALCULATOR LANGUAGE

In this section we develop the denotational semantics for the language of the simple three-function calculator shown in Figure 9.3. A "program" on this calculator consists of a sequence of keystrokes generally alternating between operands and operators. The concrete syntax in Figure 9.4 gives those combinations that we call legal on the calculator. For instance,

**6 + 33 x 2 =**

produces the value **78** on the display of the calculator. Observe that unlike more complex calculators, keystrokes are entered and processed from left to right, so that the addition is performed before the multiplication.

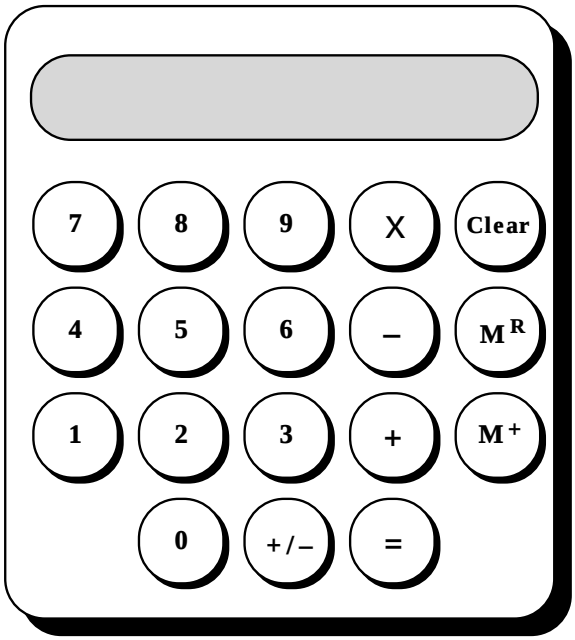


*Figure 9.3*: A Three-Function Calculator

In fact calculators usually accept any sequence of key presses, but we have limited our syntax to those collections that a user is most likely to employ. We outlaw combinations such as

**5 + + 6 =** and **88 x $^{+}$/- 11 + M$^{R}$ M$^{R}$**

that provide no new meaningful calculations although many real calculators allow them. The **Clear** key occurs in several productions because a user is likely to press it at almost any time. The plus-minus key $^{+}$/- changes the sign of the displayed value.

```
<program> ::= <expression sequence>
<expression sequence> ::= <expression>| <expression> <expression sequence>
<expression> ::= <term> | <expression> <operator> <term>
                | <expression> <answer> | <expression> <answer> +/-
<term> ::= <numeral> | MR | Clear | <term> +/-
<operator> ::= + | – | x
<answer> ::= M+ | =
<numeral> ::= <digit> | <numeral> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

*Figure 9.4*: Concrete Syntax for the Calculator Language

To simplify the definition of the denotational semantics for the calculator, the abstract syntax in Figure 9.5 considerably reduces the complexity of the notation while still representing all the key sequences allowed by the concrete syntax. Since $^{+}/-$ acts in much the same way as the answer keys, it has been included with them in the abstract syntax.

**Abstract Syntactic Domains**

P : Program  O : Operator  N : Numeral
S : ExprSequence  A : Answer  D : Digit
E : Expression

**Abstract Production Rules**

Program ::= ExprSequence
ExprSequence ::= Expression | Expression ExprSequence
Expression ::= Numeral | $\mathbf{M^R}$ | **Clear** | Expression Answer
| Expression Operator Expression
Operator ::= **+** | **–** | **x**
Answer ::= $\mathbf{M^+}$ | **=** | $^{+}/-$
Numeral ::= see Figure 9.1

*Figure 9.5*: Abstract Syntax for the Calculator Language

Following the concrete syntax for the calculator language, given the sequence of keystrokes

$$\mathbf{10\ M^+ + 6\ {}^{+}/-\ =\ x\ M^R\ =}$$

a parser will construct the abstract syntax tree shown in Figure 9.6. Notice that most operations associate to the left because of the way keystrokes are processed from left to right.

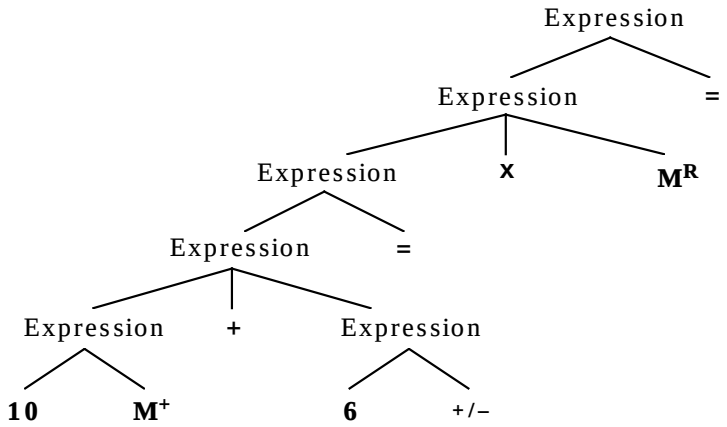


*Figure 9.6*: An Abstract Syntax Tree for $\mathbf{10\ M^+ + 6\ {}^{+}/-\ =\ x\ M^R\ =}$

The syntax of the calculator language encompasses six syntactic domains if we ignore the structure of numerals. In providing semantics for the calculator we define a semantic function for each of these domains. But before these functions can be specified, we need to describe the semantic domains into which they map, and to do so requires that we understand the operation of the calculator.

## Calculator Semantics

The description presented here for the meaning of calculator expressions is slightly more complicated than it needs to be so that some extensions of the meaning can be implemented easily. See the exercises at the end of this section for an alternative model of the calculator. To define the semantics of the calculator, we use a state that maintains four registers or values to capture its internal working:

1. An internal accumulator maintains a running total value reflecting the operations that have been carried out so far.
2. An operator flag indicates the pending operation that will be executed when another operand is made available.
3. The current display portrays the latest numeral that has been entered, the memory value that has been recalled using $\mathbf{M^R}$, or the total computed so far whenever the **=** or $\mathbf{M^+}$ key has been pressed.
4. The memory of the calculator contains a value, initially zero; it is controlled by the $\mathbf{M^+}$ and $\mathbf{M^R}$ keys.

Calculator arithmetic will be modeled by several auxiliary functions that carry out the three basic binary operations and an "identity" operation for the case when no operator is pending:

$$plus : \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$$

$$minus : \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$$

$$times : \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$$

$$nop : \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \quad \text{where } nop(a,d) = d.$$

We use the same names for the values of the operator flag and for the auxiliary operations, assuming an implicit function that identifies the flag values with the auxiliary functions. To understand how the state varies under the control of the keystrokes, consider the calculation in Figure 9.7 and how it affects these four values.

| Keystroke | Accumulator | Operator Flag | Display | Memory |
|---|---|---|---|---|
| | 0 | *nop* | 0 | 0 |
| **12** | 0 | *nop* | 12 | 0 |
| **+** | 12 | *plus* | 12 | 0 |
| **5** | 12 | *plus* | 5 | 0 |
| **+/-** | 12 | *plus* | -5 | 0 |
| = | 12 | *nop* | 7 | 0 |
| **x** | 7 | *times* | 7 | 0 |
| **2** | 7 | *times* | 2 | 0 |
| **M+** | 7 | *nop* | 14 | 14 |
| **123** | 7 | *nop* | 123 | 14 |
| **M+** | 7 | *nop* | 123 | 137 |
| **MR** | 7 | *nop* | 137 | 137 |
| **+/-** | 7 | *nop* | -137 | 137 |
| **–** | -137 | *minus* | -137 | 137 |
| **25** | -137 | *minus* | 25 | 137 |
| **=** | -137 | *nop* | -162 | 137 |
| **+** | -162 | *plus* | -162 | 137 |
| **MR** | -162 | *plus* | 137 | 137 |
| **=** | -162 | *nop* | -25 | 137 |

*Figure 9.7*: Sample Calculation of **12 + 5 +/- = x 2 M+ 123 M+ MR +/- – 25 = + MR =**

Although the meaning of a calculator program will be the final integer value shown on the display, we are also interested in the behavior of the calculator in response to individual keystrokes and partial results. This meaning depends on the following semantic domains:

Integer = { ... , -2, -1, 0, 1, 2, 3, 4, ... }

Operation = { *plus*, *minus*, *times*, *nop* }

State = Integer x Operation x Integer x Integer.

The Operation domain can be compared to an enumerated type in Pascal. These four values act as flags inside the calculator by saving the last operator keystroke. Integer represents the abstract collection of mathematical integers, and State takes values that are quadruples embodying the internal accumulator, the pending operation, the display, and the memory. In particular, observe which entries change the various values in the State tuple:

| State tuple value | Tokens that may alter the value |
|---|---|
| Accumulator | **Clear**, **+**, **–**, and **x** |
| Operator Flag | **Clear**, **+**, **–**, **x**, **=**, and $\mathbf{M^+}$ |
| Display | **Clear**, numeral, **=**, $\mathbf{M^+}$, $\mathbf{M^R}$, and $^+$**/-** |
| Memory | **Clear** and $\mathbf{M^+}$ |

The trace in Figure 9.7 shows the changing state in response to various keystrokes.

## Semantic Functions

The denotational semantics for this calculator has a semantic function for each of the syntactic domains.

*meaning* : Program → Integer
*perform* : ExprSequence → (State → State)
*evaluate* : Expression → (State → State)
*compute* : Operator → (State → State)
*calculate* : Answer → (State → State)
*value* : Numeral → Integer -- uses only nonnegative integers

Semantic equations specifying the functions are defined in Figure 9.8, with one equation for each production rule in the abstract syntax. Inspection of the semantics for individual keystrokes will provide an understanding of the calculator operation. The semantic function *meaning* calls *perform* on a sequence of one or more expressions that makes up a program, giving *perform* an initial state (0,*nop*,0,0) as its argument. An expression sequence is evaluated one expression at a time by composing executions of the *evaluate* function. Finally, *meaning* returns the display value given as a result of evaluating the last expression in the sequence.

The semantic function *evaluate* produces a function in State → State as its result when applied to an expression. The functions *compute* and *calculate* give meaning to operators and "totaling" keys. For example, **+** computes the pending operation with the accumulator and display, updating the accumulator and display but leaving the display unchanged. Moreover, *plus* becomes the new pending operation. In contrast, **=** places the computed value into the display with *nop* signaling that there is no longer a pending operation.

Observe that $\mathbf{M^R}$ and $^+$**/-** act only on the display. Compound keystrokes are handled as compositions, eliminating the need to give the argument tuple. The semantic equation, given here as a composition,

$$\mathit{evaluate}\ [\![\text{E A}]\!] = \mathit{calculate}\ [\![\text{A}]\!] \circ \mathit{evaluate}\ [\![\text{E}]\!]$$

is equivalent to writing

$$\mathit{evaluate}\ [\![\text{E A}]\!]\ (a,op,d,m) = \mathit{calculate}\ [\![\text{A}]\!]\ (\mathit{evaluate}\ [\![\text{E}]\!]\ (a,op,d,m)).$$

*meaning* ⟦P⟧ = d where *perform* ⟦P⟧(0,*nop*,0,0) = (a,op,d,m)
*perform* ⟦E S⟧ = *perform* ⟦S⟧ ∘ *evaluate* ⟦E⟧
*perform* ⟦E⟧ = *evaluate* ⟦E⟧
*evaluate* ⟦N⟧ (a,op,d,m) = (a,op,v,m) where v = *value* ⟦N⟧
*evaluate* ⟦$\mathbf{M^R}$⟧ (a,op,d,m) = (a,op,m,m)
*evaluate* ⟦**Clear**⟧ (a,op,d,m) = (0,*nop*,0,0)
*evaluate* ⟦$E_1$ O $E_2$⟧ = *evaluate* ⟦$E_2$⟧ ∘ *compute* ⟦O⟧ ∘ *evaluate* ⟦$E_1$⟧
*evaluate* ⟦E A⟧ = *calculate* ⟦A⟧ ∘ *evaluate* ⟦E⟧
*compute* ⟦**+**⟧ (a,op,d,m) = (op(a,d),*plus*,op(a,d),m)
*compute* ⟦**–**⟧ (a,op,d,m) = (op(a,d),*minus*,op(a,d),m)
*compute* ⟦**x**⟧ (a,op,d,m) = (op(a,d),*times*,op(a,d),m)
*calculate* ⟦**=**⟧ (a,op,d,m) = (a,*nop*,op(a,d),m)
*calculate* ⟦$\mathbf{M^+}$⟧ (a,op,d,m) = (a,*nop*,v,*plus*(m,v)) where v = op(a,d)
*calculate* ⟦$\mathbf{^+/-}$⟧ (a,op,d,m) = (a,op,*minus*(0,d),m)
*value* ⟦N⟧ = see Figure 9.1

*Figure 9.8*: Semantic Equations for the Calculator Language

Denotational definitions commonly use this technique of factoring out arguments to semantic equations whenever possible. It was for this reason that the syntax of *evaluate* and the other semantic functions are given in a curried form (see Chapter 5 for a discussion of curried functions)

*evaluate* : Expression → (State → State)

instead of as an uncurried function acting on a tuple

*evaluate* : (Expression x State) → State.

## A Sample Calculation

As an example of an evaluation according to the definition, consider the series of keystrokes "**2 + 3 =**". The meaning of the sequence is given by

*meaning* ⟦**2 + 3 =**⟧ = d where *perform* ⟦**2 + 3 =**⟧(0,*nop*,0,0) = (a,op,d,m).

The evaluation proceeds as follows:

*perform* ⟦**2 + 3 =**⟧(0,*nop*,0,0)

= *evaluate* ⟦**2 + 3 =**⟧(0,*nop*,0,0)

= (*calculate* ⟦**=**⟧ ∘ *evaluate* ⟦**2 + 3**⟧) (0,*nop*,0,0)

= (*calculate* ⟦**=**⟧ ∘ *evaluate* ⟦**3**⟧ ∘ *compute* ⟦**+**⟧ ∘ *evaluate* ⟦**2**⟧) (0,*nop*,0,0)

= *calculate* ⟦=⟧ (*evaluate* ⟦**3**⟧ (*compute* ⟦+⟧ (*evaluate* ⟦**2**⟧ (0,*nop*,0,0))))

= *calculate* ⟦=⟧ (*evaluate* ⟦**3**⟧ (*compute* ⟦+⟧ (0,*nop*,2,0))), since *value* ⟦**2**⟧ = 2

= *calculate* ⟦=⟧ (*evaluate* ⟦**3**⟧ (2,*plus*,2,0)), since *nop*(0,2) = 2

= *calculate* ⟦=⟧ (2,*plus*,3,0), since *value* ⟦**3**⟧ = 3

= (2,*nop*,5,0), since *plus*(2,3) = 5.

Therefore *meaning* ⟦**2 + 3 =**⟧ = 5.

A similar evaluation corresponding to the computation in Figure 9.7 will provide a useful example of elaborating the semantics of the calculator—namely, to demonstrate that

*meaning* ⟦**12 + 5 $^{+}$/- = x 2 M$^{+}$ 123 M$^{+}$ M$^{R}$ $^{+}$/- – 25 = + M$^{R}$ =**⟧ = -25.

Remember that the ambiguity in the abstract syntax is resolved by viewing the keystrokes from left to right.

Real calculators have two conditions that produce errors when evaluating integer arithmetic: arithmetic overflow and division by zero. Our calculator has no division so that we can avoid handling the problem of division by zero as a means of reducing the complexity of the example. Furthermore, we assume unlimited integers so that overflow is not a problem.

## Exercises

1. Draw the abstract syntax tree that results from parsing the series of keystrokes

   **12 + 5 $^{+}$/- M$^{+}$ M$^{+}$ - 55 =**.

   Remember, keystrokes are entered and evaluated from left to right.

2. Evaluate the semantics of these combinations of keystrokes using the denotational definition in this section:

   a) **8 $^{+}$/- + 5 x 3 =**

   b) **7 x 2 M$^{+}$ M$^{+}$ M$^{+}$ – 15 + M$^{R}$ =**

   c) **10 - 5 $^{+}$/- M$^{+}$ 6 x M$^{R}$ M$^{+}$ =**

3. Prove that for any expression E, *perform*⟦E **Clear**⟧ = *perform*⟦**Clear**⟧.

4. Some calculators treat **=** differently from the calculator in this section, repeating the most recent operation, so that "**2 + 5 = =**" leaves 12 on the display and "**2 + 5 = = =**" leaves 17. Describe the changes that must be made in the denotational semantics to model this alternative interpretation.

5. Prove that for any expression E, *meaning* [[E **=** $\mathbf{M}^{+}$]] = *meaning* [[E $\mathbf{M}^{+}$ **=**]].

6. Add to the calculator a key **sqr** that computes the square of the value in the display. Alter the semantics to model the action of this key. Its syntax should be similar to that of the $^{+}$**/-** key.

7. Alter the calculator semantics so that **Clear** leaves the memory unchanged. Modify the semantic equations to reflect this change.

8. Explain how the *evaluate* function for the semantics of the calculator language can be thought of as a homomorphism.

9. Rewrite the denotational definition of the calculator semantics taking the state to be State = Integer x Integer x (*clear* + *unclear*), representing the display, memory, and a "clear" flag. Delete the semantic functions *compute* and *calculate*, and use the following abstract syntax:

**Abstract Syntactic Domains**

P : Program E : Expression D : Digit

S : ExprSequence N : Numeral

**Abstract Production Rules**

Program ::= ExprSequence

ExprSequence ::= Expression | Expression ExprSequence

Expression ::= Numeral | $\mathbf{M}^{R}$ | **Clear** | Expression $\mathbf{M}^{+}$
| Expression **=** | Expression $^{+}$**/-**
| Expression **+** Expression | Expression **-** Expression
| Expression **x** Expression

## 9.3 THE DENOTATIONAL SEMANTICS OF WREN

The programming language Wren exemplifies a class of languages referred to as **imperative**. Several properties characterize imperative programming languages:

1. Programs consist of commands, thereby explaining the term "imperative".

2. Programs operate on a global data structure, called a store, in which results are generally computed by incrementally updating values until a final result is produced.

3. The dominant command is the assignment instruction, which modifies a location in the store.

4. Program control entails sequencing, selection, and iteration, represented by the semicolon, the **if** command, and the **while** command in Wren.

The abstract syntax for Wren appears in Figure 9.9. Compare this version with the one in Figure 1.18. Note that lists of commands are handled somewhat differently. Instead of using the postfix operator "$^+$", a command is allowed to be a pair of commands that by repetition produces a sequence of commands. However, the definition still provides abstract syntax trees for the same collection of programs. As a second change, input and output have been omitted from Wren for the time being. This simplifies our initial discussion of denotational semantics. The issues involved with defining input and output will be considered later.

**Abstract Syntactic Domains**

| | | |
|---|---|---|
| P : Program | C : Command | N : Numeral |
| D : Declaration | E : Expression | I : Identifier |
| T : Type | O : Operator | |

**Abstract Production Rules**

Program ::= **program** Identifier **is** Declaration* **begin** Command **end**

Declaration ::= **var** Identifier$^+$ **:** Type **;**

Type ::= **integer** | **boolean**

Command ::= Command **;** Command | Identifier **:=** Expression
| **skip** | **if** Expression **then** Command **else** Command
| **if** Expression **then** Command | **while** Expression **do** Command

Expression ::= Numeral | Identifier | **true** | **false** | **-** Expression
| Expression Operator Expression | **not(** Expression**)**

Operator ::= **+** | **–** | * | **/** | **or** | **and** | **<=** | **<** | **=** | **>** | **>=** | **<>**

*Figure 9.9*: Abstract Syntax for Wren

## Semantic Domains

To provide a denotational semantics for Wren, we need to specify semantic domains into which the syntactic constructs map. Wren uses two primitive domains that can be described by listing (or suggesting) their values:

Integer = { ..., -2, -1, 0, 1, 2, 3, 4, ... }

Boolean = { true, false }.

Primitive domains are combined into more complex structures by certain mathematical constructions. The calculator language uses two of these structures, the Cartesian product and the function domain. The State in the se-

mantics of the calculator is a Cartesian product of four primitive domains, so that each element of the State is a quadruple. Although we do not name the function domains, we use them in the semantic functions—for example, *evaluate* maps an Expression into a set of functions State → State. The notation for a function domain A → B agrees with normal mathematical notation. We view this expression as representing the set of functions from A into B; that the function f is a member of this set can be described by f : A → B using a colon as the symbol for membership.

Wren without input and output needs no Cartesian product for its semantics, but function domains are essential. The Store (memory) is modeled as a function from Identifiers to values,

$$\text{Store} = \text{Identifier} \rightarrow (\text{SV} + \textit{undefined}),$$

where SV represents the collection of values that may be placed in the store, the so-called **storable values**, and *undefined* is a special value indicating that an identifier has not yet been assigned a value. The constant function mapping each identifier to *undefined* serves as the initial store provided to a Wren program. Wren allows integers and Boolean values to be storable. To specify the domain of storable values, we take the union of the primitive domains Integer and Boolean. The notion of set union will not keep the sets separate if they contain common elements. For this reason, we use the notion of **disjoint union** or **disjoint sum** that requires tags on the elements from each set so that their origin can be determined. We exploit the notation

$$\text{SV} = \textit{int}(\text{Integer}) + \textit{bool}(\text{Boolean})$$

for the disjoint union of Integer and Boolean, where the tag *int* indicates the integer values and the tag *bool* specifies the Boolean values. Typical elements of SV are *int*(5), *int*(-99), and *bool*(true). Such elements can be viewed as Prolog structures where the function symbols provide the tags or as items in a Standard ML datatype. The important feature of disjoint unions is that we can always determine the origin of an element of SV by inspecting its tag. In the disjoint sum for Store, *undefined* is a tagged value with no data field. Chapter 10 provides a more formal definition of the structure of disjoint unions.

We assume several properties about the store that make it an abstraction of the physical memory of a computer—namely, that it has an unbounded number of locations and that each location will be large enough to contain any storable value. Formal semantics usually does not concern itself with implementation restrictions imposed when executing programs on an actual computer.

## Language Constructs in Wren

Structurally, Wren includes three main varieties of language constructs: declarations, commands, and expressions. In programming languages, declarations define bindings of names (identifiers) to objects such as memory locations, literals (constants), procedures, and functions. These bindings are recorded in a structure, called an **environment**, that is active over some part of a program known as the **scope** of the bindings. Since a Wren program has only one scope, the entire program, environments can be ignored in its semantics. Thus the environment of a Wren program is constant, in effect determined at the start of the program, and need not be modeled at all in the dynamic semantics of Wren. The declarations in Wren act solely as part of the context-sensitive syntax that we assume has already been verified by some other means, say an attribute grammar. Later we show that denotational semantics can be used to verify context conditions. For now, we assume that any Wren program to be analyzed by our denotational semantics has no inconsistency in its use of types. At this stage, we also ignore the program identifier, taking it as documentation only.

Expressions in a programming language produce values. An important defining trait of a language is the sorts of values that expressions can produce, called the **expressible values** or the **first-class values** of the language. The expressible values in Wren are the same as the storable values:

$$\mathrm{EV} = \mathit{int}(\mathrm{Integer}) + \mathit{bool}(\mathrm{Boolean}).$$

The value of an expression will depend on the values associated with its identifiers in the store. Therefore the semantic function *evaluate* for expressions has as its signature

$$\mathit{evaluate} : \mathrm{Expression} \to (\mathrm{Store} \to \mathrm{EV}).$$

The syntax of *evaluate* can also be given by

$$\mathit{evaluate} : \mathrm{Expression} \times \mathrm{Store} \to \mathrm{EV},$$

but we prefer the first (curried) version since it allows partial evaluation of the semantic function. The object $\mathit{evaluate}\ [\![\mathrm{I}]\!]$ makes sense as a function from the store to an expressible value when we use the curried version. This approach to defining functions sometimes allows us to factor out rightmost arguments (see command sequencing in the denotational definition given later in this section).

Commands may modify the store, so we define the meaning of a command to be a function from the current store to a new store. We mentioned earlier that the store is global and implied that only one store exists. When we speak of the “current store” and the “new store”, we mean snapshots of the same store. The signature of the semantic function *execute* for commands is given by

$$\mathit{execute} : \mathrm{Command} \to (\mathrm{Store} \to \mathrm{Store}).$$

The meaning of a command is thus a function from Store to Store.

As for primitive syntactic domains, Numerals will be handled by the semantic function *value* as with the calculator language, and the Boolean values **true** and **false** will be defined directly by *evaluate*. Note that the syntactic domain Identifier is used in defining the semantic domain Store. To make sense of this mixing of syntactic and semantic worlds, we assume that the denotational semantics of Wren has an implicit semantic function that maps each Identifier in the syntactic world to a value in the semantic world that is the identifier itself, as the attribute *Name* did in the attribute grammars for Wren. We can pretend that we have an invisible semantic function defined by *id* ⟦I⟧ = I.

Since input and output commands have been omitted from Wren for the time being, we consider the final values of the variables of the program to be the semantics of the program. So the signature of *meaning* is

*meaning* : Program → Store.

The semantic domains for Wren and the signatures of the semantic functions are summarized in Figure 9.10. Remember that the semantic domain Store allows its maps to take the special value *undefined* to represent identifiers that have not yet been assigned a value and that in the disjoint sum, *undefined* stands for a tag with an empty "value". By the way, in constructing semantic domains we assume that disjoint sum "+" has a higher precedence than the forming of a function domain, so some parentheses may be omitted—for example, those around SV + *undefined* in the definiton of Store.

**Semantic Domains**

Integer = { ... , -2, -1, 0, 1, 2, 3, 4, ... }
Boolean = { true, false }
EV = *int*(Integer) + *bool*(Boolean) -- expressible values
SV = *int*(Integer) + *bool*(Boolean) -- storable values
Store = Identifier → SV + *undefined*

**Semantic Functions**

*meaning* : Program → Store
*execute* : Command → (Store → Store)
*evaluate* : Expression → (Store → EV)
*value* : Numeral → EV

*Figure 9.10*: Semantic Domains and Functions for Wren

## Auxiliary Functions

To complete our denotational definition of Wren, we need several auxiliary functions representing the normal operations on the primitive semantic domains and others to make the semantic equations easier to read. Since Wren is an algorithmic language, its operations must map to mathematical operations in the semantic world. Here we use the semantic operations *plus*, *minus*, *times*, *divides*, *less*, *lesseq*, *greater*, *greatereq*, *equal*, *neq* defined on the integers with their normal mathematical meanings. The relational operators have syntax following the pattern

$$less : \text{Integer} \times \text{Integer} \rightarrow \text{Boolean}.$$

Operations that update and access the store can be threaded into the definitions of the semantic functions, but we get a cleaner specification by factoring these operations out of the semantic equations, thereby treating the store as an abstract data type. In defining these auxiliary functions and Wren's semantic functions, we make use of semantic metavariables in a way similar to the syntactic domains. We use "sto" for elements of Store and "val" for values in either EV or SV. Three auxiliary functions manipulate the store:

*emptySto* : Store

*emptySto* I = *undefined* or *emptySto* = $\lambda$ I . *undefined*

*updateSto* : Store x Identifier x SV $\rightarrow$ Store

*updateSto*(sto,I,val) $I_1$ = (if I = $I_1$ then val else sto($I_1$))

*applySto* : Store x Identifier $\rightarrow$ SV + *undefined*

*applySto*(sto,I) = sto(I)

The definition of *updateSto* means that *updateSto*(sto,I,val) is the function Identifier $\rightarrow$ SV + *undefined* that is identical to sto except that I is bound to val. Denotational definitions frequently use lambda notation to describe functions, as seen above in the definition of *emptySto*. See Chapter 5 for an explanation of the lambda calculus.

## Semantic Equations

The semantic equations for the denotational semantics of Wren are listed in Figure 9.11. Notice how we simply ignore the declarations in the first equation by defining the meaning of a program to be the store that results from executing the commands of the program starting with the store in which all identifiers are undefined. Command sequencing follows the pattern shown in the calculator language. Observe that *execute* **[[skip]]** is the identity function on Store. Some semantic equations for expressions are omitted in Figure 9.11—they follow the pattern given for the operations addition and less than.

*meaning* ⟦**program** I **is** D **begin** C **end**⟧ = *execute* ⟦C⟧ *emptySto*

*execute* ⟦$C_1$ **;** $C_2$⟧ = *execute* ⟦$C_2$⟧ ∘ *execute* ⟦$C_1$⟧

*execute* ⟦**skip**⟧ sto = sto

*execute* ⟦I **:=** E⟧ sto = *updateSto*(sto, I, (*evaluate* ⟦E⟧ sto))

*execute* ⟦**if** E **then** C⟧ sto = if p then *execute* ⟦C⟧ sto else sto
where *bool*(p) = *evaluate* ⟦E⟧ sto

*execute* ⟦**if** E **then** $C_1$ **else** $C_2$⟧ sto =
if p then *execute* ⟦$C_1$⟧ sto else *execute* ⟦$C_2$⟧ sto
where *bool*(p) = *evaluate* ⟦E⟧ sto

*execute* ⟦**while** E **do** C⟧ = loop
where loop sto = if p then loop(*execute* ⟦C⟧ sto) else sto
where *bool*(p) = *evaluate* ⟦E⟧ sto

*evaluate* ⟦I⟧ sto = if val=*undefined* then *error* else val
where val = *applySto*(sto, I)

*evaluate* ⟦N⟧ sto = *int*(*value* ⟦N⟧)

*evaluate* ⟦**true**⟧ sto = *bool*(true) *evaluate* ⟦**false**⟧ sto = *bool*(false)

*evaluate* ⟦$E_1$ **+** $E_2$⟧ sto = *int*(*plus*(m,n))
where *int*(m) = *evaluate* ⟦$E_1$⟧ sto and *int*(n) = *evaluate* ⟦$E_2$⟧ sto

:

*evaluate* ⟦$E_1$ **/** $E_2$⟧ sto = if n=0 then *error* else *int*(*divides*(m,n))
where *int*(m) = *evaluate* ⟦$E_1$⟧ sto and *int*(n) = *evaluate* ⟦$E_2$⟧ sto

*evaluate* ⟦$E_1$ **<** $E_2$⟧ sto = if *less*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦$E_1$⟧ sto and *int*(n) = *evaluate* ⟦$E_2$⟧ sto

**:**

*evaluate* ⟦$E_1$ **and** $E_2$⟧ sto = if p then *bool*(q) else *bool*(false)
where *bool*(p) = *evaluate* ⟦$E_1$⟧ sto and *bool*(q) = *evaluate* ⟦$E_2$⟧ sto

*evaluate* ⟦$E_1$ **or** $E_2$⟧ sto = if p then *bool*(true) else *bool*(q)
where *bool*(p) = *evaluate* ⟦$E_1$⟧ sto and *bool*(q) = *evaluate* ⟦$E_2$⟧ sto

*evaluate* ⟦**-** E⟧ sto = *int*(*minus*(0,m)) where *int*(m) = *evaluate* ⟦E⟧ sto

*evaluate* ⟦**not**(E)⟧ sto = if p then *bool*(false) else *bool*(true)
where *bool*(p) = *evaluate* ⟦E⟧ sto

*Figure 9.11*: Semantic Equations for Wren

An assignment evaluates the expression on the right and updates the store accordingly. To illustrate the change in the store caused by the *updateSto* operation, we use the notation "{x |→*int*(3), y |→*int*(5), z |→*int*(8)}" to represent a store with those three bindings where every other identifier maps to *undefined*. Observe that for any Wren program, no matter how long or complex,

the current store function will always be finite in the sense that all but a finite set of identifiers will map to *undefined*. We also use the notation "{x |→*int*(25) y |→*int*(-1)}sto" to stand for the store that is identical to sto except that x has the value 25 and y the value -1. Therefore we can write

$$\begin{aligned} &\mathit{updateSto}(\mathit{updateSto}(\mathit{emptySto}, \text{a}, \mathit{int}(0)), \text{b}, \mathit{int}(1)) \\ &\quad = \mathit{updateSto}(\{\text{a} \mid\!\!\rightarrow \mathit{int}(0)\}\mathit{emptySto}, \text{b}, \mathit{int}(1)) \\ &\quad = \{\text{a} \mid\!\!\rightarrow \mathit{int}(0),\ \text{b} \mid\!\!\rightarrow \mathit{int}(1)\}\mathit{emptySto} \\ &\quad = \{\text{a} \mid\!\!\rightarrow \mathit{int}(0),\ \text{b} \mid\!\!\rightarrow \mathit{int}(1)\}. \end{aligned}$$

Incidentally, this store is the meaning of *execute* ⟦a **:= 0;** b **:= 1**⟧ *emptySto*. To tie these two notations together, we view {a |→*int*(0), b |→*int*(1)}as an abbreviation for {a |→int(0), b |→*int*(1)}*emptySto*.

One other point needs to be made about our functional notation. It has been implicitly assumed that function application associates to the left as with the lambda calculus, so that

*execute* ⟦a **:= 0;** b **:= 1**⟧ *emptySto* = (*execute* ⟦a **:= 0;** b **:= 1**⟧) *emptySto*.

Furthermore, the arrow forming a function domain associates to the right, so that

*execute* : Command → Store → Store

means

*execute* : Command → (Store → Store).

Surprisingly, these two conventions, associating application to the left and → to the right, agree, as shown by the following signatures:

*execute* : Command → Store → Store

*execute* ⟦a **:= 0;** b **:= 1**⟧ : Store → Store

*execute* ⟦a **:= 0;** b **:= 1**⟧ *emptySto* : Store.

Also remember that composition "∘" is an associative operation so that no convention is required to disambiguate f∘g∘h.

When we inspect primitive values, as in the selection (**if**) commands, we must account for the tags provided by the disjoint sum. Maintaining correct tags is also an important part of defining the *evaluate* semantic function.

The **while** command presents special difficulties in a denotational definition. A naive approach to its meaning follows its operational description—namely, to evaluate the test and, if its value is true, to execute the body of the **while** and repeat the entire command, whereas if it is false, to do nothing (more). The corresponding semantic equation can be written:

*execute* ⟦**while** E **do** C⟧ sto =
  if p then *execute* ⟦**while** E **do** C⟧(*execute* ⟦C⟧ sto) else sto
    where *bool*(p) = *evaluate* ⟦E⟧ sto.

Although this equation captures the operational explanation, it fails to adhere to a fundamental tenet of denotational semantics—namely, that each semantic equation be compositional. The meaning of the **while** command is defined in terms of itself, not just its constituent parts. Using a technique common to functional programming we transform this equation into a definition that is compositional:

*execute* ⟦**while** E **do** C⟧ = loop
    where loop sto = if p then loop(*execute* ⟦C⟧ sto) else sto
                    where *bool*(p) = *evaluate* ⟦E⟧ sto.

In this definition we have factored out a function embodying the effect of the meaning of a **while** command; "loop: Store → Store" is a function that models *execute* ⟦**while** E **do** C⟧ compositionally as a recursive function defined on stores. This approach will be justified in Chapter 10, where we also ensure that recursive definitions of functions are really describing mathematical objects.

The meaning of expressions is straightforward, consisting of evaluating the operands and then passing the values to auxiliary functions in the semantic world. The Boolean operations **and**, **or**, and **not** are defined directly as conditional expressions in the metalanguage.

## Error Handling

We take a simple approach to dynamic errors in Wren, automatically adding a special element *error* to each of the semantic domains and assuming that all semantic functions produce *error* when given *error* as an argument; that is, errors propagate. In an actual programming language, a program aborts when a dynamic (or runtime) error occurs, but the kind of denotational semantics described in this section—namely, **direct denotational semantics**—makes this sort of termination very difficult to define.

Nontermination of a **while** command is also not modeled directly by our semantics, but it will be considered when we study semantic domains more carefully in Chapter 10. We tolerate an operational point of view in the sense that a nonterminating **while** command gives no value at all under the *execute* semantic function, making *execute* a partial function. For example, we consider *execute* ⟦**while true do skip**⟧ to be an undefined function on any store.

The semantic equations in Figure 9.11 are heavily dependent on pattern matching, such as "*int*(m) = *evaluate* ⟦E⟧ sto", for their definition. The question may arise as to whether this pattern matching can fail—for example, what if "*evaluate* ⟦E⟧ sto" in a numeric expression produces the value *bool*(true)? Since we assume that programs that are analyzed by the

denotational semantics have already been verified as syntactically valid according to both the context-free and context-sensitive syntax of Wren, a numeric expression cannot produce a Boolean value. If, on the other hand, such an expression produces *error*, say by accessing an undefined variable, then the *error* value is propagated through the equations.

## Semantic Equivalence

Denotational semantics provides a method for formulating the equivalence of two language phrases.

**Definition**: Two language constructs are **semantically equivalent** if they share the same denotation according to their denotational definition. ∎

For example, for any command C, since

*execute* ⟦C**; skip**⟧ sto = *execute* ⟦**skip**⟧ (*execute* ⟦C⟧ sto) = *execute* ⟦C⟧ sto,

we conclude that "C**; skip**" is semantically equivalent to C.

Furthermore, we can show that the following denotations are mathematically the same function:

*execute* ⟦a **:= 0;** b **:= 1**⟧ sto = {a |→*int*(0), b |→*int*(1)}sto

*execute* ⟦b **:= 1;** a **:=** b–b⟧ sto = *execute* ⟦a **:=** b–b⟧ {b |→*int*(1)}sto
= {b |→*int*(1), a |→*int*(0)}sto,

since 0 = *minus*(1,1). Therefore "a **:= 0;** b **:= 1**" is semantically equivalent to "b **:= 1;** a **:=** b–b".

As a consequence of this definition of semantic equivalence, we cannot distinguish nonterminating computations since they have no denotation. Hence, the commands "**while true do** m:=m+1" and "**while true do skip**" are semantically equivalent. (Since we consider only abstract syntax trees when analyzing syntax, we omit "**end while**" from these commands.)

## Input and Output

Remember, Wren allows only integer values for input and output. The **read** and **write** commands permit Wren to communicate with entities outside of programs—namely, files of integers. We model these files as semantic domains that are sets of finite lists where any particular file is an element of one of these sets:

Input = Integer*
Output = Integer*.

At each point during the execution of a program, the values in these lists influence the current computation and the final result of the program. We define the meaning of a program as a function between two files taken from Input and Output:

*meaning* : Program → (Input → Output).

Since input and output are performed by commands, the semantic function for them must encompass the values of the input and output files. We describe the state of a machine executing a Wren program with input and output as a semantic domain containing the store and two lists of integers:

State = Store x Input x Output.

The signature of the *execute* semantic function for commands becomes

*execute* : Command → State → State.

Again, we rely on auxiliary functions to handle the manipulation of the input and output files to simplify the semantic equations. We represent an arbitrary list of integers by the notation $[n_1,n_2, ..., n_k]$ where $k \geq 0$. A list with $k=0$ is empty and is represented as [ ]. We need four auxiliary functions that are similar to those found in a list-processing language such as Scheme (see Chapter 6).

*head* : Integer* → Integer

*head* $[n_1,n_2, ..., n_k] = n_1$ provided $k \geq 1$.

*tail* : Integer* → Integer*

*tail* $[n_1,n_2, ..., n_k] = [n_2, ..., n_k]$ provided $k \geq 1$.

*null* : Integer* → Boolean

*null* $[n_1,n_2, ..., n_k] = (k=0)$

*affix* : Integer* x Integer → Integer*

*affix* $([n_1,n_2, ..., n_k],m) = [n_1,n_2, ..., n_k,m]$.

Although all the semantic equations for *meaning* and *execute* must be altered to reflect the new signature, the changes for most commands are merely cosmetic and are left to the reader as an exercise. We list only those semantic equations that are totally new, using "inp" and "outp" as metavariables ranging over Input and Output:

*meaning* ⟦**program** I **is** D **begin** C **end**⟧ inp = outp
where (sto, $inp_1$, outp) = *execute* ⟦C⟧ (*emptySto*, inp, [ ])

*execute* ⟦**read** I⟧ (sto,inp,outp) =
if *null*(inp) then *error*
else (*updateSto*(sto,I,*int*(*head*(inp))), *tail*(inp), outp)

*execute* ⟦**write** E⟧ (sto,inp,outp) = (sto, inp, *affix*(outp,val))
where *int*(val) = *evaluate* ⟦E⟧ sto.

In the next section where Wren is implemented in the laboratory, we develop a prototype implementation of Wren based on its denotational semantics. We consider two methods for implementing input and output, one based on these denotational definitions, and one that ignores the denotational approach, handling input and output interactively.

## Elaborating a Denotational Definition

The denotational semantics for Wren supplies a meaning to each Wren program. Here we apply the semantic functions defined for Wren to give meaning to the following Wren program that contains both input and output. This example illustrates that a complete denotational description of even a small program entails a considerable amount of patience and attention to detail.

```
program sample is
        var sum,num : integer;
    begin
        sum := 0;
        read num;
        while num>=0 do
            if num>9 and num<100
                    then sum := sum+num
            end if;
            read num
        end while;
        write sum
    end
```

The meaning of the program is defined by

*meaning* ⟦**program** I **is** D **begin** C **end**⟧ inp = outp
where (sto, inp1, outp) = *execute* ⟦C⟧ (*emptySto*, inp, [ ]).

The semantic equations for *execute* must be altered to reflect the use of states. For example, the meaning of an assignment command is defined as

*execute* ⟦I **:=** E⟧ (sto,inp,outp) =
(*updateSto*(sto,I,(*evaluate* ⟦E⟧ sto)),inp,outp).

Let the input list be [5,22,-1]. To simplify the work, the elaboration employs several abbreviations.

d = **var** sum,num : **integer**

$c_1$ = sum := 0

$c_2$ = **read** num

$c_3$ = **while** num>=0 **do** $c_{3.1}$ ; $c_{3.2}$
  $c_{3.1}$ = **if** num>9 **and** num<100 **then** sum := sum+num
  $c_{3.2}$ = **read** num
$c_4$ = **write** sum

Using these abbreviations for the abstract syntax trees that make up the program, the meaning of sample unfolds as follows:

*meaning* ⟦**program** sample **is** d **begin** $c_1$ ; $c_2$ ; $c_3$ ; $c_4$ **end**⟧ [5,22,-1] = outp
  where (sto, $inp_1$, outp) = *execute* ⟦$c_1$ ; $c_2$ ; $c_3$ ; $c_4$⟧ (*emptySto*, [5,22,-1], [ ]).

In Wren the semantics of a program reduces to the meaning of its sequence of commands.

*execute* ⟦$c_1$ ; $c_2$ ; $c_3$ ; $c_4$⟧ (*emptySto*, [5,22,-1], [ ])
  = (*execute* ⟦$c_4$⟧ ∘ *execute* ⟦$c_3$⟧ ∘ *execute* ⟦$c_2$⟧ ∘ *execute* ⟦$c_1$⟧)
      (*emptySto*, [5,22,-1], [ ])
  = *execute* ⟦$c_4$⟧ (*execute* ⟦$c_3$⟧ (*execute* ⟦$c_2$⟧ (*execute* ⟦$c_1$⟧
      (*emptySto*, [5,22,-1], [ ])))).

The commands are executed from the inside out, starting with $c_1$.

*execute* ⟦sum := 0⟧ (*emptySto*, [5,22,-1], [ ])
  = (*updateSto*(*emptySto*, sum, (*evaluate* ⟦0⟧ *emptySto*)), [5,22,-1], [ ])
  = (*updateSto*(*emptySto*, sum, *int*(0)), [5,22,-1], [ ])
  = ({sum |→*int*(0)}, [5,22,-1], [ ]).

*execute* ⟦**read** num⟧ ({sum |→*int*(0)}, [5,22,-1], [ ])
  = (*updateSto*({sum |→*int*(0)},num,*int*(5)), [22,-1], [ ])
  = ({sum |→*int*(0),num |→*int*(5)}, [22,-1], [ ]).

Let $sto_{0,5}$ = {sum |→*int*(0),num |→*int*(5)}.

*execute* ⟦**while** num>=0 **do** $c_{3.1}$ ; $c_{3.2}$ ⟧ ($sto_{0,5}$, [22,-1], [ ])
    = loop ($sto_{0,5}$, [22,-1], [ ])
  where loop (sto,in,out) =
    if p then loop(*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ (sto,in,out)) else (sto,in,out)
      where *bool*(p) = *evaluate* ⟦num>=0⟧ sto.

We work on the Boolean expression first.

*evaluate* ⟦num⟧ $sto_{0,5}$ = *applySto*($sto_{0,5}$, num) = *int*(5).

*evaluate* ⟦0⟧ $sto_{0,5}$ = *int*(0).

*evaluate* ⟦num>=0⟧ $sto_{0,5}$
  = if *greatereq*(m,n) then *bool*(true) else *bool*(false)
    where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,5}$
      and *int*(n) = *evaluate* ⟦-0⟧ $sto_{0,5}$
  = if *greatereq*(5,0) then *bool*(true) else *bool*(false)
  = *bool*(true).

Now we can execute loop for the first time.

loop ($sto_{0,5}$, [22,-1], [ ])
= if p then loop(*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ ($sto_{0,5}$, [22,-1], [ ]))
else ($sto_{0,5}$, [22,-1], [ ])
where *bool*(p) = *evaluate* ⟦num>=0⟧ $sto_{0,5}$
= if true then loop(*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ ($sto_{0,5}$, [22,-1], [ ]))
else ($sto_{0,5}$, [22,-1], [ ])
= loop(*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ ($sto_{0,5}$, [22,-1], [ ])).

To complete the execution of loop, we need to execute the body of the **while** command.

*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ ($sto_{0,5}$, [22,-1], [ ])
= *execute* ⟦**read** num⟧
(*execute* ⟦**if** num>9 **and** num<100 **then** sum := sum+num⟧
($sto_{0,5}$, [22,-1], [ ])).

We need the value of the Boolean expression in the **if** command next.

*evaluate* ⟦num>9⟧ $sto_{0,5}$
= if *greater*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,5}$
and *int*(n) = *evaluate* ⟦9⟧ $sto_{0,5}$
= if *greater*(5,9) then *bool*(true) else *bool*(false)
= *bool*(false)

*evaluate* ⟦num<100⟧ $sto_{0,5}$
= if *less*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,5}$
and *int*(n) = *evaluate* ⟦100⟧ $sto_{0,5}$
= if *less*(5,100) then *bool*(true) else *bool*(false)
= *bool*(true)

*evaluate* ⟦num>9 **and** num<100⟧ $sto_{0,5}$
= if p then *bool*(q) else *bool*(false)
where *bool*(p) = *evaluate* ⟦num>9⟧ $sto_{0,5}$
and *bool*(q) = *evaluate* ⟦num<100⟧ $sto_{0,5}$
= if false then *bool*(true) else *bool*(false)
= *bool*(false).

Continuing with the **if** command, we get the following.

*execute* ⟦**if** num>9 **and** num<100 **then** sum := sum+num⟧ ($sto_{0,5}$, [22,-1], [ ])
= if p then *execute* ⟦sum := sum+num⟧ ($sto_{0,5}$, [22,-1], [ ])
else ($sto_{0,5}$, [22,-1], [ ])
where *bool*(p) = *evaluate* ⟦num>9 **and** num<100⟧ $sto_{0,5}$
= if false then *execute* ⟦sum := sum+num⟧ ($sto_{0,5}$, [22,-1], [ ])
else ($sto_{0,5}$, [22,-1], [ ])
= ($sto_{0,5}$, [22,-1], [ ]).

After finishing with the **if** command, we proceed with the second command in the body of the **while**.

$$\begin{aligned}&\textit{execute}\ [\![\textbf{read}\ \text{num}]\!]\ (\text{sto}_{0,5}, [22,\text{-}1], [\,])\\ &\qquad = (\textit{updateSto}(\text{sto}_{0,5},\text{num},\textit{int}(22)), [\text{-}1], [\,])\\ &\qquad = (\{\text{sum} \mid\!\!\rightarrow \textit{int}(0), \text{num} \mid\!\!\rightarrow \textit{int}(22)\}, [\text{-}1], [\,]).\end{aligned}$$

Let $\text{sto}_{0,22} = \{\text{sum} \mid\!\!\rightarrow \textit{int}(0), \text{num} \mid\!\!\rightarrow \textit{int}(22)\}$.

Summarizing the execution of the body of the **while** command, we have the result

$$\textit{execute}\ [\![\text{c}_{3.1}\ ;\ \text{c}_{3.2}]\!]\ (\text{sto}_{0,5}, [22,\text{-}1], [\,]) = (\text{sto}_{0,22}, [\text{-}1], [\,]).$$

This completes the first pass through loop.

$$\begin{aligned}&\text{loop}\ (\text{sto}_{0,5}, [22,\text{-}1], [\,])\\ &\quad = \text{loop}\ (\textit{execute}\ [\![\text{c}_{3.1}\ ;\ \text{c}_{3.2}]\!]\ (\text{sto}_{0,5}, [22,\text{-}1], [\,]))\\ &\quad = \text{loop}\ (\text{sto}_{0,22}, [\text{-}1], [\,]).\end{aligned}$$

Again, we work on the Boolean expression from the **while** command first.

$$\textit{evaluate}\ [\![\text{num}]\!]\ \text{sto}_{0,22} = \textit{applySto}(\text{sto}_{0,22}, \text{num}) = \textit{int}(22).$$

$$\textit{evaluate}\ [\![0]\!]\ \text{sto}_{0,22} = \textit{int}(0).$$

$$\begin{aligned}&\textit{evaluate}\ [\![\text{num}>=0]\!]\ \text{sto}_{0,22}\\ &\quad = \text{if}\ \textit{greatereq}(\text{m},\text{n})\ \text{then}\ \textit{bool}(\text{true})\ \text{else}\ \textit{bool}(\text{false})\\ &\qquad \text{where}\ \textit{int}(\text{m}) = \textit{evaluate}\ [\![\text{num}]\!]\ \text{sto}_{0,22}\\ &\qquad\quad \text{and}\ \textit{int}(\text{n}) = \textit{evaluate}\ [\![0]\!]\ \text{sto}_{0,22}\\ &\quad = \text{if}\ \textit{greatereq}(22,0)\ \text{then}\ \textit{bool}(\text{true})\ \text{else}\ \textit{bool}(\text{false})\\ &\quad = \textit{bool}(\text{true}).\end{aligned}$$

Now we can execute loop for the second time.

$$\begin{aligned}&\text{loop}\ (\text{sto}_{0,22}, [\text{-}1], [\,])\\ &\quad = \text{if p then loop}(\textit{execute}\ [\![\text{c}_{3.1}\ ;\ \text{c}_{3.2}]\!]\ (\text{sto}_{0,22}, [\text{-}1], [\,]))\\ &\qquad\quad \text{else}\ (\text{sto}_{0,22}, [\text{-}1], [\,])\\ &\qquad\qquad \text{where}\ \textit{bool}(\text{p}) = \textit{evaluate}\ [\![\text{num}>=0]\!]\ \text{sto}_{0,22}\\ &\quad = \text{if true then loop}(\textit{execute}\ [\![\text{c}_{3.1}\ ;\ \text{c}_{3.2}]\!]\ (\text{sto}_{0,22}, [\text{-}1], [\,]))\\ &\qquad\quad \text{else}\ (\text{sto}_{0,22}, [\text{-}1], [\,])\\ &\quad = \text{loop}(\textit{execute}\ [\![\text{c}_{3.1}\ ;\ \text{c}_{3.2}]\!]\ (\text{sto}_{0,22}, [\text{-}1], [\,])).\end{aligned}$$

Again we execute the body of the **while** command.

$$\begin{aligned}&\textit{execute}\ [\![\text{c}_{3.1}\ ;\ \text{c}_{3.2}]\!]\ (\text{sto}_{0,22}, [\text{-}1], [\,])\\ &\quad = \textit{execute}\ [\![\textbf{read}\ \text{num}]\!]\\ &\qquad (\textit{execute}\ [\![\textbf{if}\ \text{num}>9\ \textbf{and}\ \text{num}<100\\ &\qquad\qquad \textbf{then}\ \text{sum} := \text{sum}+\text{num}]\!]\ (\text{sto}_{0,22}, [\text{-}1], [\,])).\end{aligned}$$

The Boolean expression in the **if** command must be evaluated again.

*evaluate* ⟦num>9⟧ $sto_{0,22}$
= if *greater*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,22}$
and *int*(n) = *evaluate* ⟦9⟧ $sto_{0,22}$
= if *greater*(22,9) then *bool*(true) else *bool*(false)
= *bool*(true)

*evaluate* ⟦num<100⟧ $sto_{0,22}$
= if *less*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{0,22}$
and *int*(n) = *evaluate* ⟦100⟧ $sto_{0,22}$
= if *less*(22,100) then *bool*(true) else *bool*(false)
= *bool*(true)

*evaluate* ⟦num>9 **and** num<100⟧ $sto_{0,22}$
= if p then *bool*(q) else *bool*(false)
where *bool*(p) = *evaluate* ⟦num>9⟧ $sto_{0,22}$
and *bool*(q) = *evaluate* ⟦num<100⟧ $sto_{0,22}$
= if true then *bool*(true) else *bool*(false)
= *bool*(true).

This time we execute the **then** clause in the **if** command.

*execute* ⟦**if** num>9 **and** num<100 **then** sum := sum+num⟧ ($sto_{0,22}$, [-1], [ ])
= if p then *execute* ⟦sum := sum+num⟧ ($sto_{0,22}$, [-1], [ ])
else ($sto_{0,22}$, [-1], [ ])
where *bool*(p) = *evaluate* ⟦num>9 **and** num<100⟧ $sto_{0,22}$
= if true then *execute* ⟦sum := sum+num⟧ ($sto_{0,22}$, [-1], [ ])
else ($sto_{0,22}$, [-1], [ ])
= *execute* ⟦sum := sum+num⟧ ($sto_{0,22}$, [-1], [ ]).

Therefore we need the value of the right side of the assignment command.

*evaluate* ⟦sum+num⟧ $sto_{0,22}$
= *int*(*plus*(m,n))
where *int*(m) = *evaluate* ⟦sum⟧ $sto_{0,22}$
and *int*(n) = *evaluate* ⟦num⟧ $sto_{0,22}$
= *int*(*plus*(0,22)) = *int*(22).

Completing the assignment provides the state produced by the **if** command.

*execute* ⟦sum := sum+num⟧ ($sto_{0,22}$, [-1], [ ])
= (*updateSto*($sto_{0,22}$, sum, (*evaluate* ⟦sum+num⟧ $sto_{0,22}$)), [-1], [ ])
= (*updateSto*($sto_{0,22}$, sum, *int*(22)), [-1], [ ])
= ({sum $\mid\!\rightarrow$*int*(22),num $\mid\!\rightarrow$*int*(22)}, [-1], [ ]).

Let $sto_{22,22}$ = {sum $\mid\!\rightarrow$*int*(22),num $\mid\!\rightarrow$*int*(22)}.

Continuing with the body of the **while** command for its second pass yields a state with a new store after executing the **read** command.

*execute* ⟦**read** num⟧ ($sto_{22,22}$, [-1], [ ]))
= (*updateSto*($sto_{22,22}$,num,*int*(-1)), [ ], [ ])
= ({sum |→*int*(22),num |→*int*(-1)}, [ ], [ ]).

Let $sto_{22,-1}$ = {sum |→*int*(22),num |→*int*(-1)}.

Summarizing the second execution of the body of the **while** command, we have the result

*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ ($sto_{0,22}$, [-1], [ ]) = ($sto_{22,-1}$, [ ], [ ]).

This completes the second pass through loop.

loop ($sto_{0,22}$, [-1], [ ])
= loop (*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ ($sto_{0,22}$, [-1], [ ]))
= loop($sto_{22,-1}$, [ ], [ ]).

Again we work on the Boolean expression from the **while** command first.

*evaluate* ⟦num⟧ $sto_{22,-1}$ = *applySto*($sto_{22,-1}$, num) = *int*(-1).

*evaluate* ⟦0⟧ $sto_{22,-1}$ = *int*(0).

*evaluate* ⟦num>=0⟧ $sto_{22,-1}$
= if *greatereq*(m,n) then *bool*(true) else *bool*(false)
where *int*(m) = *evaluate* ⟦num⟧ $sto_{22,-1}$
and *int*(n) = *evaluate* ⟦0⟧ $sto_{22,-1}$
= if *greatereq*(-1,0) then *bool*(true) else *bool*(false)
= *bool*(false).

When we execute loop for the third time, we exit the **while** command.

loop ($sto_{22,-1}$, [ ], [ ])
= if p then loop(*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ ($sto_{22,-1}$, [ ], [ ]))
else ($sto_{22,-1}$, [ ], [ ])
where *bool*(p) = *evaluate* ⟦num>=0⟧ $sto_{22,-1}$
= if false then loop(*execute* ⟦$c_{3.1}$ ; $c_{3.2}$⟧ ($sto_{22,-1}$, [ ], [ ]))
else ($sto_{22,-1}$, [ ], [ ])
= ($sto_{22,-1}$, [ ], [ ]).

Recapping the execution of the **while** command, we conclude:

*execute* ⟦**while** num>=0 **do** $c_{3.1}$ ; $c_{3.2}$ ⟧ ($sto_{0,5}$, [22,-1], [ ])
= loop ($sto_{0,5}$, [22,-1], [ ])
= ($sto_{22,-1}$, [ ], [ ]).

Now we continue with the fourth command in the program.

*evaluate* ⟦sum⟧ $sto_{22,-1}$ = *applySto*($sto_{22,-1}$, sum) = *int*(22).

*execute* ⟦**write** sum⟧ ($sto_{22,-1}$, [ ], [ ])
= ($sto_{22,-1}$, [ ], *affix*([ ],val)) where *int*(val) = *evaluate* ⟦sum⟧ $sto_{22,-1}$
= ($sto_{22,-1}$, [ ], [22])).

Finally, we summarize the execution of the four commands to obtain the meaning of the program.

*execute* ⟦$c_1$ ; $c_2$ ; $c_3$ ; $c_4$⟧ (*emptySto*, [5,22,-1], [ ]) = ($sto_{22,-1}$, [ ], [22])).

and so

*meaning* ⟦**program** sample **is** d **begin** $c_1$ ; $c_2$ ; $c_3$ ; $c_4$ **end**⟧ [5,22,-1] = [22].

## Exercises

1. Add these language constructs to Wren and provide their denotational semantics.

   a) repeat-until command

   Command ::= ... | **repeat** Command **until** Expression

   b) conditional expression

   Expression ::= ... | **if** Expression **then** Expression **else** Expression

   Use your definition to prove the semantic equivalence of

   m := **if** $E_1$ **then** $E_2$ **else** $E_3$ and **if** $E_1$ **then** m:=$E_2$ **else** m:=$E_3$.

   c) expression with side effects

   Expression ::= ... | **begin** Command **return** Expression **end**.

   d) case command

   Command ::= **case** IntegerExpr **of** (**when** $\text{Numeral}^+$ **=>** Command)$^+$

2. Express the denotational meaning of this code fragment as a function Store → Store using the notation described in this section for representing stores:

```
switch := true; sum := 0; k := 1;
while k<4 do
      switch := not(switch);
      if switch then sum := sum+k end if;
      k := k+1
end while
```

3. Modify the remaining semantic equations for *execute* to reflect the inclusion of input and output in Wren.

4. Carefully prove: *execute* ⟦m:=5; n:=m+3⟧ = *execute* ⟦n:=8; m:=n-3⟧.

5. Prove the semantic equivalence of these language phrases:

   a) **while** E **do** C and **if** E **then** (C**; while** E **do** C) **else skip**

   b) **if** E **then** $C_1$ **else** $C_2$ and **if not(**E**) then** $C_2$ **else** $C_1$

   c) x := 5; y := 2*x and y := 10; x := y/2

   d) $E_1 + E_2$ and $E_2 + E_1$

   e) **if** E **then** (**if** E **then** $C_1$ **else** $C_2$) **else** $C_3$ and **if** E **then** $C_1$ **else** $C_3$

   f) (**while** E **do** $C_1$); **if** E **then** $C_2$ **else** $C_3$ and (**while** E **do** $C_1$); $C_3$

6. Elaborate the denotational meaning of the following Wren program using the function, meaning: Program → Input → Output, taking [5,22,-1] as the input list:

```
program bool is
  var a,b : boolean;
begin
  a := true;     b := true;
  while a or b do
        write 5;
        if not(a) then b := not(b) end if;
        if b then a := not(a) end if
  end while
end
```

7. Discuss the ambiguity in binary operations that occurs when expressions can have side effects—for example, the expressions in exercise 1c or in a language with function subprograms. Give an example of this ambiguity. Where is this issue dealt with in a denotational definition?

8. A vending machine takes nickels, dimes, and quarters and has buttons to select gum (30¢), a candy bar (50¢), or a brownie (65¢), or to return the coins entered. After entering a sequence of coins and pressing a button, the user receives the selected item (or nothing) and the change from the purchase. When the value of the coins is insufficient for the button pressed, the outcome is the same as from return.

   The following two examples show how the vending machine might be used:

   "dime, dime, dime, quarter, candy bar button"
   produces a candy bar and 5 cents in change.

   "quarter, nickel, return" or "quarter, nickel, candy"
   produce nothing and 30 cents in change.

The language of the vending machine has the following abstract syntax:

Program ::= CoinSeq Button
CoinSeq ::= ε | Coin CoinSeq
Coin ::= Nickel | Dime | Quarter
Button ::= Gum | Candy | Brownie | Return

Using the semantic domains

Result = { gum, candy, brownie, naught } and
Number = { 0, 1, 2, 3, 4, ... },

provide a denotational semantics for the language of the vending machine.

9. Consider the language of propositional logic, which contains symbols from the following syntactic domains:

| | |
|---|---|
| var : Var = { p, q, r, $p_1$, $q_1$, $r_1$, $p_2$, $q_2$, ... } | Propositional variables |
| con : Con = { **t**, **f** } | Propositional constants |
| uop : Uop = { ~ } | Unary operation |
| bop : Bop = { $\wedge$, $\vee$, $\supset$, $\leftrightarrow$ } | Binary operations |

a) Give a BNF grammar for the concrete syntax of the language (well-formed formulas) of propositional logic.

b) Describe an abstract syntax of this language in terms of the syntactic variables for the syntactic domains.

c) Provide a denotational definition that gives meaning to the formulas in the language of propositional logic, specifying the semantic domain(s), the syntax of the semantic function(s), and the semantic equations that define the semantics of the language. One parameter to the semantic functions will be a function assigning Boolean values to the propositional variables.

## 9.4 LABORATORY: IMPLEMENTING DENOTATIONAL SEMANTICS

In Chapter 2 we developed a scanner and parser that take a text file containing a Wren program and produce an abstract syntax tree. Now we continue, creating a prototype implementation of Wren based on its denotational semantics. The semantic equations are translated into Prolog clauses that carry out the denotational definition when executed.

We illustrate the result of this exercise with a Wren program that tests whether positive integers are prime. It expects a list of integers terminated by a negative number or zero as input and returns those integers that are prime and

zero for those that are not. A sample execution of a version of the interpreter using interactive (nondenotational) input and output is shown below:

```
>>> Interpreting Wren via Denotational Semantics <<<
Enter name of source file: prime.wren
    program prime is
      var num,div : integer;
      var done : boolean;
    begin
      read num;
      while num>0 do
        div := 2; done := false;
        while div<= num/2 and not(done) do
          done := num = div*(num/div);
          div := div+1
         end while;
        if done then write 0
                else write num
        end if;
        read num
      end while
    end
Scan successful
Parse successful
Input: 23
Output = 23
Input: 91
Output = 0
Input: 149
Output = 149
Input: 0
Final store:
      num   int(0)
      div   int(75)
      done  bool(false)
yes
```

If the denotational approach to input and output is followed using a state containing an input list, an output list, and the store (see section 9.3), the results look like this:

```
Enter input list followed by a period:
[23,79,91,129,149,177,0].
```

```
Output = [23,79,0,0,149,0]
yes
```

We consider the version with interactive input and output in this section, leaving the denotational version as an exercise. To implement a denotational definition in Prolog, we translate semantic functions into relations given by Prolog predicates. Since functions *are* relations, this approach works nicely. For example, the *execute* function

$$execute : \text{Command} \to \text{Store} \to \text{Store}$$

becomes the predicate execute(Cmd, Sto, NewSto). In the abstract syntax tree produced by the parser, command sequencing is handled by using a Prolog list of commands. In the semantic equations, execute processes the first command producing a temporary store that is given to another application of execute on the rest of the commands. Executing an empty list results in the identity relation on stores.

```
execute([Cmd|Cmds],Sto,NewSto) :- execute(Cmd,Sto,TempSto),
                                  execute(Cmds,TempSto,NewSto).
execute([ ],Sto,Sto).
```

Both the **if** and **while** commands require auxiliary predicates in their Prolog versions. We illustrate the **while** command since it is a bit more complex:

```
execute(while(Test,Body),Sto,NewSto) :- loop(Test,Body,Sto,NewSto).

loop(Test,Body,Sto,NewSto) :- evaluate(Test,Sto,Val),
                              iterate(Val,Test,Body,Sto,NewSto).

iterate(bool(true),Test,Body,Sto,NewSto) :- execute(Body,Sto,TempSto),
                                            loop(Test,Body,TempSto,NewSto).
iterate(bool(false),Test,Body,Sto,Sto).
```

Before considering the assignment command, we need to discuss how to model the finite function that comprises the store. We portray the store as a Prolog structure of the form

```
sto(a, int(3), sto(b, int(8), sto(c, bool(false), nil)))
```

for the store $\{a \mid\!\to int(3),\ b \mid\!\to int(8),\ c \mid\!\to bool(\text{false})\}$. The empty store is given by the Prolog atom nil. The auxiliary functions for manipulating the store become predicates defined as follows:

```
updateSto(sto(Ide,V,Sto),Ide,Val,sto(Ide,Val,Sto)).

updateSto(sto(I,V,Sto),Ide,Val,sto(I,V,NewSto)) :-
                                        updateSto(Sto,Ide,Val,NewSto).

updateSto(nil,Ide,Val,sto(Ide,Val,nil)).
```

The predicate updateSto(Sto,Ide,Val,NewSto) searches the current store for a match with Ide. If the identifier is found, its binding is changed to Val in the new store. If Ide is not found, the binding Ide $\mapsto$Val is inserted at the end of the store. A value binding for an identifier is found using the predicate applySto.

```
applySto(sto(Ide,Val,Sto),Ide,Val).
applySto(sto(I,V,Sto),Ide,Val) :- applySto(Sto,Ide,Val).
applySto(nil,Ide,undefined) :- write('Undefined variable'), nl, abort.
```

Note that when an identifier cannot be found in the store, applySto prints an error message and aborts the execution of the denotational interpreter to indicate the runtime error.

The assignment command evaluates the expression on the right and updates the identifier in the store:

```
execute(assign(Ide,Exp),Sto,NewSto) :- evaluate(Exp,Sto,Val),
                                       updateSto(Sto,Ide,Val,NewSto).
```

The *evaluate* function, *evaluate* : Expression $\rightarrow$ Store $\rightarrow$ EV, takes an expression and the current store and produces an expressible value. For literals, we use the value given by the scanner and attach the appropriate tag:

```
evaluate(num(N),Sto,int(N)).
evaluate(true,Sto,bool(true)).
evaluate(false,Sto,bool(false)).
```

For identifiers, evaluate simply fetches a value from the store:

```
evaluate(ide(Ide),Sto,Val) :- applySto(Sto,Ide,Val).
```

Numeric binary operations are handled by evaluating the two operands using evaluate and calling a predicate compute that carries out the operations using the native arithmetic in Prolog. We illustrate a few of the operations:

```
evaluate(exp(Opr,E1,E2),Sto,Val) :- evaluate(E1,Sto,V1), evaluate(E2,Sto,V2),
                                    compute(Opr,V1,V2,Val).
compute(times,int(M),int(N),int(R)) :- R is M*N.
compute(divides,int(M),int(0),int(0)) :- write('Division by zero'), nl, abort.
compute(divides,int(M),int(N),int(R)) :- R is M//N.
```

Observe how a division-by-zero error causes the interpreter to abort. This action does not follow the denotational definition but avoids the problem of propagating errors in the prototype interpreter.

Comparisons and Boolean operations can be dealt with in a similar manner, except that some operations are implemented using pattern matching:

```
evaluate(bexp(Opr,E1,E2),Sto,Val) :-  evaluate(E1,Sto,V1), evaluate(E2,Sto,V2),
                                      compute(Opr,V1,V2,Val).

compute(equal,int(M),int(N),bool(true)) :- M =:= N.
compute(equal,int(M),int(N),bool(false)).

compute(neq,int(M),int(N),bool(false)) :- M =:= N.
compute(neq,int(M),int(N),bool(true)).

compute(lteq,int(M),int(N),bool(true)) :- M =< N.
compute(lteq,int(M),int(N),bool(false)).

compute(and,bool(true),bool(true),bool(true)).
compute(and,bool(P),bool(Q),bool(false)).
```

For an entire program, meaning calls execute with an empty store and returns the final store, which is printed by the predicate controlling the system.

```
meaning(prog(Dec,Cmd),Sto) :- execute(Cmd,nil,Sto).
```

We now consider the two approaches to input and output. For the interactive version, we disregard the denotational definitions for **read** and **write** and simply rely on Prolog to fetch an input value from the keyboard and print an integer on the screen. Executing the **read** command this way requires an auxiliary predicate readnum that can be based on the part of the scanner for processing integers.

```
execute(read(Ide),Sto,NewSto) :-  write('Input: '), nl, readnum(N),
                                  updateSto(Sto,Ide,int(N),NewSto).

execute(write(Exp),Sto,Sto) :- evaluate(Exp,Sto,Val), Val=int(M),
                               write('Output = '), write(M), nl.
```

The denotational approach complicates the semantic equations for execute, as was discussed in section 9.3. Then the **read** and **write** commands act directly on the input and output lists in the state, modeled by a Prolog structure, state(Sto,Inp,Outp).

```
execute(read(Ide),state(Sto,[H|T],Outp),state(NewSto,T,Outp)) :-
                                  updateSto(Sto,Ide,int(H),NewSto).

execute(read(Ide),state(Sto,[ ],Outp),state(NewSto,[ ],Outp)) :-
                                  write('Attempt to read empty input'), nl, abort.

execute(write(Exp),state(Sto,Inp,Outp),state(Sto,Inp,Outp1)) :-
                        evaluate(Exp,Sto,Val), int(M)=Val, concat(Outp,[M],Outp1).
```

Note that in both versions of **write**, we use a variable as the third parameter of evaluate, and then use unification, denoted by =, to pull the integer out of the structure. This convention ensures that if evaluate involves a store lookup

that fails, the failure comes in the body of the third clause applySto and not because undefined in the head of that clause failed to pattern match with int(M).

The top-level meaning predicate calls execute with the original input list and produces the output list as its result. We depend on a predicate go to request the input and print the output.

```
meaning(prog(Dec,Cmd),Inp,Outp)) :-
                    execute(Cmd,state(nil,Inp,[ ]),state(Sto,Inp1,Outp)).

go :- nl, write('>>> Interpreting Wren <<<'), nl, nl,
      write('Enter name of source file: '), nl, getfilename(FileName), nl,
      see(FileName), scan(Tokens), seen, write('Scan successful'), nl, !,
      program(Parse,Tokens,[eop]), write('Parse successful'), nl, !,
      write('Enter input list followed by a period: '), nl, read(Inp), nl,
      meaning(Parse,Inp,Outp),nl,write('Output = '), write(Outp), nl.
```

All of the clauses defining execute must correctly maintain the state; for example, the assignment command has no effect on the input or output list but needs access to the store argument inside the state:

```
execute(assign(Ide,Exp),state(Sto,Inp,Outp),state(Sto1,Inp,Outp)) :-
                    evaluate(Exp,Sto,Val), updateSto(Sto,Ide,Val,Sto1).
```

## Exercises

1. Supply Prolog definitions for the remaining commands: **skip** and **if**.

2. Supply Prolog definitions for subtraction, multiplication, **or**, unary minus, **not**, and the remaining relations. Be careful in handling the tags on the values.

3. Write a Prolog predicate readnum that accepts a string of digits from the terminal and forms an integer. Modify the predicate to accept an optional minus sign immediately preceding the digits.

4. Complete the Prolog definition of the prototype interpreter for both interactive input and output and for the denotational version.

5. Extend the prototype interpreter to include the following language constructs:

   a) repeat-until commands

      Command ::= ... | **repeat** Command **until** Expression

   b) conditional expressions

      Expression ::= ... | **if** Expression **then** Expression **else** Expression

c) expressions with side effects

Expression ::= ... | **begin** Command **return** Expression **end**

6. Write a scanner, parser, and denotational interpreter for the calculator language described in section 9.2.

## 9.5 DENOTATIONAL SEMANTICS WITH ENVIRONMENTS

In this section we extend Wren to a programming language in which declarations contribute to the semantics as well as the context-sensitive syntax of the language. The addition of procedures significantly enlarges the language, and we thus take the P from procedure to give it a new name, Pelican. Figure 9.12 contains a definition of its abstract syntax. Notice the features that make Pelican different from Wren:

**Abstract Syntactic Domains**

| | | |
|---|---|---|
| P : Program | C : Command | N : Numeral |
| B : Block | E : Expression | I : Identifier |
| D : Declaration | O : Operator | L : Identifier$^+$ |
| T : Type | | |

**Abstract Production Rules**

Program ::= **program** Identifier **is** Block

Block ::= Declaration **begin** Command **end**

Declaration ::= ε | Declaration Declaration
| **const** Identifier = Expression
| **var** Identifier : Type | **var** Identifier, Identifier$^+$ : Type
| **procedure** Identifier **is** Block
| **procedure** Identifier **(**Identifier **:** Type**) is** Block

Type ::= **integer** | **boolean**

Command ::= Command **;** Command | Identifier **:=** Expression
| **skip** | **if** Expression **then** Command **else** Command
| **if** Expression **then** Command | **while** Expression **do** Command
| **declare** Block | Identifier | Identifier**(**Expression**)**
| **read** Identifier | **write** Expression

Expression ::= Numeral | Identifier | **true** | **false** | **-** Expression
| Expression Operator Expression | **not(**Expression**)**

Operator ::= **+** | **–** | ***** | **/** | **or** | **and** | **<=** | **<** | **=** | **>** | **>=** | **<>**

*Figure 9.12*: Abstract Syntax for Pelican

1. A program may consist of several scopes corresponding to the syntactic domain Block that occurs in the main program, in anonymous blocks headed by **declare**, and in procedures.

2. Each block may contain constant declarations indicated by **const** as well as variable declarations.

3. Pelican permits the declaration of procedures with zero or one value parameter and their use as commands. We limit procedures to no more than one parameter for the sake of simplicity. Multiple parameters will be left as an exercise.

We have slightly modified the specification of lists of declarations to make the semantic equations easier to define. In particular, the nonempty lists of identifiers in variable declarations are specified with two clauses: the first handles a basis case of one identifier, and the second manages lists of two or more identifiers. In addition, we specify Pelican without the **read** and **write** commands, whose definitions are left as an exercise at the end of this section.

## Environments

In a block structured language with more than one scope, such as Pelican, the same identifier can refer to different objects in different parts of the program. The region where an identifier has a unique meaning is called the **scope** of the identifier, and this meaning is recorded in a structure called an **environment**. Therefore in Pelican the bindings between a variable identifier and a value split into two parts: (1) a binding between the identifier and a location, modeling a memory address, and (2) a binding of the location to its value in the store. The record of bindings between identifiers and locations as well as bindings of other sorts of objects, such as literals and procedures, to identifiers is maintained in the environment. Those values that are bindable to identifiers are known as **denotable values**, and in Pelican they are given by the semantic domain

DV = *int*(Integer) + *bool*(Boolean) + *var*(Location) + Procedure,

where Procedure represents the domain of procedure objects in Pelican. We defer the discussion of Pelican procedures until later. The first two terms in the disjoint sum for DV provide the literal values that can be bound to identifiers by a **const** declaration.

As with stores, we use auxiliary functions to manipulate environments, thereby treating them as an abstract data type:

*emptyEnv* : Environment

*emptyEnv* I = *unbound*

*extendEnv* : Environment x Identifier x DV → Environment
*extendEnv*(env,I,dval) $I_1$ = (if I = $I_1$ then dval else env($I_1$))

*applyEnv* : Environment x Identifier → DV + *unbound*
*applyEnv*(env,I) = env(I).

## Stores

A store in Pelican becomes a function from locations, here modeled by the natural numbers, to the storable values, augmented by two special values: (1) *unused* for those locations that have not been bound to an identifier by a declaration, and (2) *undefined* for locations that have been associated with a variable identifier but do not have a value yet. Locations serve as an abstraction of memory addresses and should not be confused with them. In fact, any ordinal set can be used to model locations; we take the natural numbers for convenience.

The auxiliary functions for stores now include operations for allocating and deallocating memory locations at block entry and exit:

*emptySto* : Store
*emptySto* loc = *unused*

*updateSto* : Store x Location x (SV + *undefined* + *unused*) → Store
*updateSto*(sto,loc,val) $loc_1$ = (if loc = $loc_1$ then val else sto($loc_1$))

*applySto* : Store x Location → SV + *undefined* + *unused*
*applySto*(sto,loc) = sto(loc)

*allocate* : Store → Store x Location
*allocate* sto = (*updateSto*(sto,loc,*undefined*),loc)
where loc = *minimum* { k | sto(k) = *unused* }

*deallocate* : Store x Location → Store
*deallocate*(sto,loc) = *updateSto*(sto,loc,*unused*).

The semantic domains for Pelican are summarized in Figure 9.13. We save the explanation of the Procedure domain until later in this section.

Figures 9.14 and 9.15 show how a Pelican program with multiple scopes influences environments and the store. We use a notation for environments that is similar to the one employed to display the store. The expression "[a |→*int*(5),b |→*var*(0)]" indicates that the identifier a is bound to the constant 5 and b is bound to the location 0, while all other identifiers are *unbound*. Furthermore, "[x |→*var*(3),y |→*var*(4)]env" denotes the environment that is identical to env except that x and y have new bindings.

**Semantic Domains**

Integer = { ... , -2, -1, 0, 1, 2, 3, 4, ... }
Boolean = { true, false }
EV = *int*(Integer) + *bool*(Boolean) -- expressible values
SV = *int*(Integer) + *bool*(Boolean) -- storable values
DV = EV + *var*(Location) + Procedure -- denotable values
Location = Natural Number = { 0, 1 2, 3, 4, ... }
Store = Location → SV + *unused* + *undefined*
Environment = Identifier → DV + *unbound*
Procedure = *proc0*(Store → Store) + *proc1*(Location → Store → Store)

**Semantic Functions**

*meaning* : Program → Store
*perform* : Block → Environment → Store → Store
*elaborate* : Declaration → Environment → Store → Environment x Store
*execute* : Command → Environment → Store → Store
*evaluate* : Expression → Environment → Store → EV
*value* : Numeral → EV

*Figure 9.13*: Semantic Domains and Functions for Pelican

In the representation of the stores in Figure 9.15, locations that are unused are simply omitted, whereas locations that are allocated but without a meaningful value are indicated within the brackets as bound to *undefined*. So the empty store with all locations bound to *unused* can be depicted by { }. Note that seven different locations are allocated for the seven variables declared in the program.

## Semantic Functions

The main alteration in the semantic functions for Pelican is to add Environment as an argument and to include functions providing meaning to Blocks and Declarations. The semantic function *elaborate* constructs a new environment on top of the given environment reflecting the declarations that are processed in the current block. Observe that *elaborate* produces a new store as well as a new environment, since the declaration of variable identifiers requires the allocation of locations, which thereby changes the state of the store. On the other hand, constant and procedure declarations have no effect on the store. The function *perform* has the same signature as *execute*, but it elaborates the declarations in the block before its commands are executed. The signatures of the semantic functions may be found in Figure 9.13.

| **program** scope **is** | **Environment** |
|---|---|
| **const** c = 9**;** | [ c $\mid\!\rightarrow$ *int*(9) ] |
| **var** a **: integer;** | [ a $\mid\!\rightarrow$ *var*(0), c $\mid\!\rightarrow$ *int*(9) ] |
| **var** b **: boolean;** | [ b $\mid\!\rightarrow$ *var*(1), a $\mid\!\rightarrow$ *var*(0), c $\mid\!\rightarrow$ *int*(9) ] |
| **begin** | |
| a **:=** 10; | $env_1$ |
| b **:=** a>0; | $env_1$ |
| **declare** | |
| **const** a **=** 0**;** | [ a $\mid\!\rightarrow$ *int*(0) ] $env_1$ |
| **var** x,y **: integer;** | [ y $\mid\!\rightarrow$ *var*(3), x $\mid\!\rightarrow$ *var*(2), a $\mid\!\rightarrow$ *int*(0) ] $env_1$ |
| **begin** | |
| x **:=** a; | $env_2$ |
| y **:=** c**–**2**;** | $env_2$ |
| **declare** | |
| **var** x **: boolean;** | [ x $\mid\!\rightarrow$ *var*(4) ] $env_2$ |
| **var** c,d **: integer;** | [ d $\mid\!\rightarrow$ *var*(6), c $\mid\!\rightarrow$ *var*(5), x $\mid\!\rightarrow$ *var*(4) ] $env_2$ |
| **begin** | |
| x := **not(**b**);** | $env_3$ |
| c **:=** y**+**5**;** | $env_3$ |
| d **:=** 17 | $env_3$ |
| **end;** | |
| x **:= –**c | $env_2$ |
| **end;** | |
| a **:=** a**+**5 | $env_1$ |
| **end** | |

where $env_1$ = [ b $\mid\!\rightarrow$ *var*(1), a $\mid\!\rightarrow$ *var*(0), c $\mid\!\rightarrow$ *int*(9) ]

$env_2$ = [ y $\mid\!\rightarrow$ *var*(3), x $\mid\!\rightarrow$ *var*(2), a $\mid\!\rightarrow$ *int*(0) ] $env_1$

= [ y $\mid\!\rightarrow$ *var*(3), x $\mid\!\rightarrow$ *var*(2), a $\mid\!\rightarrow$ *int*(0), b $\mid\!\rightarrow$ *var*(1), c $\mid\!\rightarrow$ *int*(9) ]

$env_3$ = [ d $\mid\!\rightarrow$ *var*(6), c $\mid\!\rightarrow$ *var*(5), x $\mid\!\rightarrow$ *var*(4) ] $env_2$

= [ d $\mid\!\rightarrow$ *var*(6), c $\mid\!\rightarrow$ *var*(5), x $\mid\!\rightarrow$ *var*(4), y $\mid\!\rightarrow$ *var*(3), a $\mid\!\rightarrow$ *int*(0), b $\mid\!\rightarrow$ *var*(1) ]

*Figure 9.14*: Environments for the Program “scope”

| | Store |
|---|---|
| **program** scope **is** | |
| **const** c = 9**;** | { } |
| **var** a **: integer;** | { 0 $\mid\!\rightarrow ud$ } |
| **var** b **: boolean;** | { 0 $\mid\!\rightarrow ud$, 1 $\mid\!\rightarrow ud$ } |
| **begin** | |
| a **:=** 10; | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow ud$ } |
| b **:=** a>0; | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$ } |
| **declare** | |
| **const** a = 0**;** | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$ } |
| **var** x,y **: integer;** | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow ud$, 3 $\mid\!\rightarrow ud$ } |
| **begin** | |
| x **:=** a; | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(0)$, 3 $\mid\!\rightarrow ud$ } |
| y **:=** c–2**;** | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(0)$, 3 $\mid\!\rightarrow int(7)$ } |
| **declare** | |
| **var** x **: boolean;** | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(0)$, 3 $\mid\!\rightarrow int(7)$, 4 $\mid\!\rightarrow ud$ } |
| **var** c,d **: integer;** | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(0)$, 3 $\mid\!\rightarrow int(7)$, 4 $\mid\!\rightarrow ud$, 5 $\mid\!\rightarrow ud$, 6 $\mid\!\rightarrow ud$ } |
| **begin** | |
| x := **not(**b**);** | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(0)$, 3 $\mid\!\rightarrow int(7)$, 4 $\mid\!\rightarrow bool(\text{false})$, 5 $\mid\!\rightarrow ud$, 6 $\mid\!\rightarrow ud$ } |
| c **:=** y+5**;** | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(0)$, 3 $\mid\!\rightarrow int(7)$, 4 $\mid\!\rightarrow bool(\text{false})$, 5 $\mid\!\rightarrow int(12)$, 6 $\mid\!\rightarrow ud$ } |
| d **:=** 17 | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(0)$, 3 $\mid\!\rightarrow int(7)$, 4 $\mid\!\rightarrow bool(\text{false})$, 5 $\mid\!\rightarrow int(12)$, 6 $\mid\!\rightarrow int(17)$ } |
| **end;** | |
| x **:=** –c | { 0 $\mid\!\rightarrow int(10)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(-9)$, 3 $\mid\!\rightarrow int(7)$, 4 $\mid\!\rightarrow bool(\text{false})$, 5 $\mid\!\rightarrow int(12)$, 6 $\mid\!\rightarrow int(17)$) } |
| **end;** | |
| a **:=** a+5 | { 0 $\mid\!\rightarrow int(15)$, 1 $\mid\!\rightarrow bool(\text{true})$, 2 $\mid\!\rightarrow int(-9)$, 3 $\mid\!\rightarrow int(7)$, 4 $\mid\!\rightarrow bool(\text{false})$, 5 $\mid\!\rightarrow int(12)$, 6 $\mid\!\rightarrow int(17)$ } |
| **end** | |
| where $ud$ = *undefined* | |

*Figure 9.15*: The Store for the Program “scope”

## Semantic Equations

Many of the semantic equations are straightforward extensions of those for Wren, especially for those language constructs that are defined in the denotational semantics of Wren. Figure 9.16 shows the semantic equations except that we have omitted many of the *evaluate* equations since they all follow the pattern established by the addition operation. We focus on those functions that are entirely new—namely, *perform* and *elaborate*. The function *perform* is invoked by *meaning* for the whole program and by *execute* for an anonymous block (**declare**). It also is encapsulated in the procedure objects declared in a program, but these will be considered in detail later.

---

*meaning* ⟦**program** I **is** B⟧ = *perform* ⟦B⟧ *emptyEnv emptySto*

*perform* ⟦D **begin** C **end**⟧ env sto = *execute* ⟦C⟧ $env_1$ $sto_1$
where ($env_1$, $sto_1$) = *elaborate* ⟦D⟧ env sto

*elaborate* ⟦ε⟧ env sto = (env, sto)
*elaborate* ⟦$D_1$ $D_2$⟧ env sto = *elaborate* ⟦$D_2$⟧ $env_1$ $sto_1$
where ($env_1$, $sto_1$) = *elaborate* ⟦$D_1$⟧ env sto
*elaborate* ⟦**const** I **=** E⟧ env sto = (*extendEnv*(env,I,*evaluate* ⟦E⟧ env sto), sto)
*elaborate* ⟦**var** I : T⟧ env sto = (*extendEnv*(env,I,*var*(loc)), $sto_1$)
where ($sto_1$, loc) = *allocate* sto
*elaborate* ⟦**var** I, L : T⟧ env sto = *elaborate* ⟦**var** L : T⟧ $env_1$ $sto_1$
where ($env_1$,$sto_1$) = *elaborate* ⟦**var** I : T⟧ env sto
*elaborate* ⟦**procedure** I **is** B⟧ env sto = ($env_1$, sto)
where $env_1$ = *extendEnv*(env,I,*proc0*(proc))
and proc = *perform* ⟦B⟧ $env_1$
*elaborate* ⟦**procedure** $I_1$($I_2$ : T) **is** B⟧ env sto = ($env_1$, sto)
where $env_1$ = *extendEnv*(env,$I_1$,*proc1*(proc))
and proc loc = *perform* ⟦B⟧ *extendEnv*($env_1$,$I_2$,*var*(loc))

---

*Figure 9.16*: Semantic Equations for Pelican (Part 1)

Compare the equation for elaborating a sequence of declarations with that for executing a pair of commands. Since *elaborate* produces a pair of values, Environment and Store, the composition operator cannot be used. Of course, an empty declaration leaves the environment and store unchanged.

```
execute ⟦C1 ; C2⟧ env sto = execute ⟦C2⟧ env (execute ⟦C1⟧ env sto)
execute ⟦skip⟧ env sto = sto
execute ⟦I := E⟧ env sto = updateSto(sto, loc, (evaluate ⟦E⟧ env sto))
      where var(loc) = applyEnv(env,I)
execute ⟦if E then C⟧ env sto = if p then execute ⟦C⟧ env sto else sto
      where bool(p) = evaluate ⟦E⟧ env sto
execute ⟦if E then C1 else C2⟧ env sto =
      if p then execute ⟦C1⟧ env sto else execute ⟦C2⟧ env sto
         where bool(p) = evaluate ⟦E⟧ env sto
execute ⟦while E do C⟧ = loop
      where loop env sto = if p then loop env (execute ⟦C⟧ env sto) else sto
                           where bool(p) = evaluate ⟦E⟧ env sto
execute ⟦declare B⟧ env sto = perform ⟦B⟧ env sto
execute ⟦I⟧ env sto = proc sto
      where proc0(proc) = applyEnv(env,I)
execute ⟦I(E)⟧ env sto = proc loc updateSto(sto1,loc,evaluate ⟦E⟧ env sto)
      where proc1(proc) = applyEnv(env,I)  and  (sto1,loc) = allocate sto

evaluate ⟦I⟧ env sto =
      if dval = int(n) or dval = bool(p) then dval
                else if dval = var(loc)
                        then if applySto(sto,loc) = undefined
                                then error
                                else applySto(sto,loc)
                where dval = applyEnv(env,I)
evaluate ⟦N⟧ env sto = int(value ⟦N⟧)
evaluate ⟦true⟧ env sto = bool(true)
evaluate ⟦false⟧ env sto = bool(false)
evaluate ⟦E1 + E2⟧ env sto = int(plus(m,n))
      where int(m) = evaluate ⟦E1⟧ env sto and int(n) = evaluate ⟦E2⟧ env sto
      :
evaluate ⟦E1 / E2⟧ env sto = if n=0 then error else int(divides(m,n))
      where int(m) = evaluate ⟦E1⟧ env sto and int(n) = evaluate ⟦E2⟧ env sto
      :
```

*Figure 9.16*: Semantic Equations for Pelican (Part 2)

The declaration of a list of variable identifiers is reduced to declarations of individual variable identifiers by elaborating the single identifier (the head of the list) to produce a new environment and store and then by elaborating the list of identifiers (the tail of the list) using that environment and store. In the equation for "**var** I **:** T", *allocate* produces a new state of the store because a location with the value *undefined* has been reserved for the variable. Recall that L is the metavariable for nonempty lists of identifiers. The equation for "**const** I = E" simply binds the current value of E to the identifier I in the environment, leaving the store unchanged. Note that these constant identifiers are bound to dynamic expressions whose values may not be known until run-time.

Observe that when a sequence of commands is executed, both commands receive the same environment; only the store is modified by the commands. The semantic equations may also be written using compositon:

$$execute\ [\![C_1\ ;\ C_2]\!]\ env = (execute\ [\![C_2]\!]\ env) \circ (execute\ [\![C_1]\!]\ env)$$

An assignment command depends on the environment for the location of the target variable and on the store for the value of the expression on the right side. Executing an assignment results in a modification of the store using *updateSto*.

Because we assume Pelican programs have already been checked for syntax errors (both context-free and context-sensitive), only syntactically correct programs are submitted for semantic analysis. Therefore identifiers used in assignment commands, in expression, and in procedure calls are bound to values of the appropriate type. The following semantic decisions need to be handled in the semantic equations (in the absence of the **read** command):

1. Whether an identifier in an expression represents a constant or a variable.
2. Whether the location bound to a variable identifier has a value when it is accessed (whether it is defined).
3. Whether the second operand to a divides operation is zero.

## Procedures

A procedure declaration assembles a new binding in the environment. We consider procedures without parameters first.

$$\begin{aligned} &elaborate\ [\![\textbf{procedure}\ I\ \textbf{is}\ B]\!]\ env\ sto = (env_1,\ sto) \\ &\qquad \text{where } env_1 = (extendEnv(env, I, proc0(proc)) \\ &\qquad \text{and } proc = perform\ [\![B]\!]\ env_1. \end{aligned}$$

The procedure object proc, constructed to complete the binding, encapsulates a call of *perform* on the body of the procedure in the environment now being defined, thus ensuring two important properties:

1. Since a procedure object carries along an extension of the environment in effect at its definition, we get **static scoping**. That means nonlocal variables in the procedure will refer to variables in the scope of the declaration, not in the scope of the call of the procedure (dynamic scoping). A procedure object constructed this way is an example of a closure (see Section 8.2).
2. Since the environment $env_1$ inserted into the procedure object contains the binding of the procedure identifier with this object, recursive references to the procedure are permitted. If recursion is forbidden, the procedure object can be defined by

$$\text{proc} = \textit{perform}\ [\![\text{B}]\!]\ \text{env}.$$

When such a procedure object is invoked by "*execute* ⟦I⟧ env sto", the object is found by accessing the current environment and is executed by passing the current store to it, after first removing the tag *proc0*.

*execute* ⟦I⟧ env sto = proc sto
where *proc0*(proc) = *applyEnv*(env,I).

For procedures that take a parameter, the object defined in the declaration is a function of a location corresponding to the formal parameter that will be provided at procedure invocation time when the value of the actual parameter passed to the procedure is stored in the location.

*elaborate* ⟦**procedure** $I_1$**(**$I_2$ **:** T**) is** B⟧ env sto = ($env_1$, sto)
where $env_1$ = *extendEnv*(env,$I_1$,*proc1*(proc))
and proc loc = *perform* ⟦B⟧ *extendEnv*($env_1$,$I_2$,*var*(loc)).

The environment encapsulated with the procedure object includes a binding of the unspecified location "loc" to the formal parameter $I_2$. Note that the actual location must be allocated at the point of call, thus providing "call by value" semantics for the parameter.

*execute* ⟦**I(**E**)**⟧ env sto = proc loc *updateSto*($sto_1$,loc,*evaluate* ⟦E⟧ env sto)
where *proc1*(proc) = *applyEnv*(env,I) and ($sto_1$,loc) = *allocate* sto.

The procedure object then executes with a store having the allocated location loc bound to the value of the actual parameter "*evaluate* ⟦E⟧ env sto". Again the environment $env_1$, provided to the procedure object proc, contains the binding of the procedure name, so that recursion can take place.

Figure 9.17 shows a Pelican program with the declaration of a recursive procedure along with the environments at points of the program. Because Pelican adheres to static scoping, we can identify the bound identifiers at each position in the program. Since the procedure has no local variables other than the formal parameter, the procedure object proc disregards the elaboration of its (empty) declarations. The four calls, sum(3), sum(2), sum(1), and

| | Environment |
|---|---|
| **program** summation **is** | |
| **var** s **: integer;** | [ s $\mid\rightarrow$ *var*(0) ] |
| **procedure** sum(n:integer) **is** | [ s $\mid\rightarrow$ *var*(0), sum $\mid\rightarrow$ *proc1*(proc) ] |
| **begin** | |
| **if** n>0 | $env_{2,loc}$ |
| **then** s **:=** s+n**;** | $env_{2,loc}$ |
| sum(n–1) **end if** | $env_{2,loc}$ |
| **end;** | |
| **begin** | |
| s **:=** 0**;** | $env_1$ |
| sum(3) | $env_1$ |
| **end** | |

where

proc =

$\lambda$ loc . *execute* ⟦**if** n>0 **then** s **:=** s+n**;** sum(n–1)⟧ *extendEnv*($env_1$,n,var(loc))

$env_1$ = [ s $\mid\rightarrow$ *var*(0), sum $\mid\rightarrow$ *proc1*(proc) ]

$env_{2,1}$ = [ s $\mid\rightarrow$ *var*(0), sum $\mid\rightarrow$ *proc1*(proc), n $\mid\rightarrow$ *var*(1) ]

$env_{2,2}$ = [ s $\mid\rightarrow$ *var*(0), sum $\mid\rightarrow$ *proc1*(proc), n $\mid\rightarrow$ *var*(2) ]

$env_{2,3}$ = [ s $\mid\rightarrow$ *var*(0), sum $\mid\rightarrow$ *proc1*(proc), n $\mid\rightarrow$ *var*(3) ]

$env_{2,4}$ = [ s $\mid\rightarrow$ *var*(0), sum $\mid\rightarrow$ *proc1*(proc), n $\mid\rightarrow$ *var*(4) ]

*Figure 9.17*: A Procedure Declaration in Pelican

sum(0), result in four environments $env_{2,1}$, $env_{2,2}$, $env_{2,3}$, and $env_{2,4}$, respectively, as new locations are allocated.

In Figure 9.18 we describe the status of the store as the declarations and commands of the program summation are processed according to the denotational semantics of Pelican, showing the store only when it changes. Each time sum is invoked, a new location is allocated and bound to n as the value of loc in the procedure object proc. Therefore $env_{2,loc}$ stands for four different environments for the body of the procedure depending on the current value of loc—namely, 1, 2, 3, or 4. These four locations in the store correspond to the four activation records that an implementation of Pelican creates when executing this program.

| | Store |
|---|---|
| **var** s **: integer;** | { 0 \|→*undefined* } |
| **procedure** sum(n:integer) **is**... | |
| s **:=** 0**;** | { 0 \|→*int*(0) } |
| sum(3) | { 0 \|→*int*(0), 1 \|→*int*(3) } |
| **if** n>0 ... | |
| s **:=** s+n**;** | { 0 \|→*int*(3), 1 \|→*int*(3) } |
| sum(2) | { 0 \|→*int*(3), 1 \|→*int*(3), 2 \|→*int*(2) } |
| **if** n>0 ... | |
| s **:=** s+n**;** | { 0 \|→*int*(5), 1 \|→*int*(3), 2 \|→*int*(2) } |
| sum(1) | { 0 \|→*int*(5), 1 \|→*int*(3), 2 \|→*int*(2), 3 \|→*int*(1) } |
| **if** n>0 ... | |
| s **:=** s+n**;** | { 0 \|→*int*(6), 1 \|→*int*(3), 2 \|→*int*(2), 3 \|→*int*(1) } |
| sum(0) | { 0 \|→*int*(6), 1 \|→*int*(3), 2 \|→*int*(2), 3 \|→*int*(1), 4 \|→*int*(0) } |
| **if** n>0 ... | is false causing termination. |

*Figure 9.18*: The Store While Executing "summation"

## Exercises

1. Although we defined an auxiliary function *deallocate*, we made no use of it in the denotational semantics of Pelican. Extend the denotational definition to provide for the deallocation of store locations at block exit.

   *Hint*: Use *perform* ⟦D **begin** C **end**⟧ env sto =
   *release* ⟦D⟧ $env_1$ (*execute* ⟦C⟧ $env_1$ $sto_1$)
   where ($env_1$,$sto_1$) = *elaborate* ⟦D⟧ env sto

   and define the semantic function *release*.

2. Provide denotational definitions for the **read** and **write** commands. Use triples of the form (sto,inp,outp) to represent the state of a computation.

3. Modify Pelican so that the parameter is passed by

   a) reference

   b) value-result (an **in-out** parameter in Ada)

   c) constant (a read-only parameter known as an **in** parameter in Ada).

4. Modify Pelican so that it uses dynamic scoping to resolve nonlocal variable references.

5. Modify Pelican so that a procedure may have an arbitrary number of parameters.

6. Trace the environment and store in the manner of Figures 9.14, 9.15, 9.17, and 9.18 for the following Pelican programs:

a)
```
program trace1 is
    var a : integer;
    procedure q1 is
        begin a := 5 end;
    procedure q2 is
        var a : integer;
        begin a := 3; q1 end;
begin
    a := 0; q2; write a
end
```

b)
```
program trace2 is
    var n,f : integer;
    procedure fac(n : integer) is
        procedure mul(m : integer) is begin f := f*m end;
        begin
            if n=0 then f:=1
                   else fac(n-1); mul(n) end if
        end;
    begin
        n := 4; fac(n); write f
    end
```

7. Carefully explain and contrast the following terms used to classify the semantic domains in a denotational definition of a programming language:
   - expressible values
   - storable values
   - denotable values.

   Use examples from real programming languages to illustrate a sort of values that is

   a) expressible, storable, and denotable

   b) denotable but neither storable nor expressible

   c) expressible and storable but not denotable

   d) expressible but neither storable nor denotable.

8. Suppose that Pelican is extended to include functions of one parameter, passed by value. The abstract syntax now has productions of the form:

   Declaration ::= ...
   | **function** Identifier **(** Identifier **:** Type **) :** Type **is** Declaration
   **begin** Command **return** Expression **end**

   Expressions ::= ... | Identifier **(** Expression **)**

   Make all the necessary changes in the denotational definition of Pelican to incorporate this new language construct.

9. Some programming languages allow functions with no parameters but require an empty parameter list at the time of the call, as in f(). Why do these languages have this requirement?

10. Remove the assignment command, parameter passing, the **read** command, and the **while** command from Pelican, calling the new language BabyPelican. Also consider the values *unused* and *undefined* as the same. Use structural induction (see Chapter 8) to prove that every command in BabyPelican is semantically equivalent to **skip**.

11. Construct a prototype denotational interpreter for Pelican in Prolog.

## 9.6 CHECKING CONTEXT-SENSITIVE SYNTAX

In Chapter 3 we developed an attribute grammar to check the context constraints imposed by the declarations and type regime of a programming language. Here we solve the same problem in the framework of denotational semantics, mapping a program into the semantic domain Boolean in such a way that the resulting truth value records whether the program satisfies the requirements of the context-sensitive syntax. We assume that programs analyzed in this way already agree with the context-free syntax of the language.

A slightly modified Pelican serves as an example for illustrating this process of verifying context conditions. We leave procedure declarations and calls to be handled in an exercise and include the **read** and **write** commands now. The context conditions for Pelican are listed in Figure 9.19, including those for procedures that will be treated in an exercise.

The context checker has radically simplified denotational semantics since runtime behavior need not be modeled. In particular, we drop the store and register only the types of objects in environments, not their values. Figure 9.20 lists the semantic domains and the signatures of the semantic functions. The semantic function *elaborate* enters the type information given by declarations into the environment in much the same way the attribute grammar constructed a symbol table in Chapter 3. *Typify* produces the type of an expression given the types of the identifiers recorded in the (type) environment.

1. The program name identifier lies in a scope outside the main block.
2. All identifiers that appear in a block must be declared in that block or in an enclosing block.
3. No identifier may be declared more than once at the top level of a block.
4. The identifier on the left side of an assignment command must be declared as a variable, and the expression on the right side must be of the same type.
5. An identifier occurring as an (integer) element must be an integer variable or an integer constant.
6. An identifier occurring as a Boolean element must be a Boolean variable or a Boolean constant.
7. An identifier occurring in a read command must be an integer variable.
8. An identifier used in a procedure call must be defined in a procedure declaration with the same (zero or one) number of parameters.
9. The identifier defined as the formal parameter in a procedure declaration is considered to belong to the top level declarations of the block that forms the body of the procedure.
10. The expression in a procedure call must match the type of the formal parameter in the procedure's declaration.

*Figure 9.19*: Context Conditions for Pelican

Since environments map identifiers to types, we need a semantic domain Sort to assemble the possible types. Note that we distinguish between constants (*integer* and *boolean*) and variables (*intvar* and *boolvar*). It is important to remember that every domain is automatically augmented with an *error* value, and every semantic function and auxiliary function propagates *error*.

**Semantic Domains**

Boolean = { true, false }

Sort = { *integer*, *boolean*, *intvar*, *boolvar*, *program*, *unbound* }

Environment = Identifier → Sort

**Semantic Functions**

*validate* : Program → Boolean

*examine* : Block → Environment → Boolean

*elaborate* : Declaration → (Environment x Environment) → (Environment x Environment)

*check* : Command → Environment → Boolean

*typify* : Expression → Environment → Sort

*Figure 9.20*: Semantic Domains and Functions for Context Checking

We need two environments to elaborate each block:

1. One environment (locenv) holds the identifiers local to the block so that duplicate identifier declarations can be detected. It begins the block as an empty envirnoment with no bindings.
2. The other environment (env) collects the accumulated bindings from all of the enclosing blocks. This environment is required so that the expressions in constant declarations can be typified.

Both type environments are built in the same way by adding a new binding using *extendEnv* as each declaration is elaborated. The auxiliary functions for maintaining environments are listed below (see section 9.5 for definitions):

*emptyEnv* : Environment

*extendEnv* : Environment x Identifier x Sort → Environment

*applyEnv* : Environment x Identifier → Sort

*type* : Type → Sort

*type*(**integer**) = *intvar*

*type*(**boolean**) = *boolvar*.

The semantic equations in Figure 9.21 show that each time a block is initialized, we build a local type environment starting with the empty environment. The first equation indicates that the program identifier is viewed as lying in a block of its own, and so it does not conflict with any other occurrences of identifiers. This alteration in the context conditions for program identifiers as compared to Wren makes the denotational specification much simpler.

---

*validate* ⟦**program** I **is** B⟧ =

    *examine* ⟦B⟧ *extendEnv*(*emptyEnv*,I,*program*)

*examine* ⟦D **begin** C **end**⟧ env = *check* ⟦C⟧ $env_1$

    where ($locenv_1$, $env_1$) = *elaborate* ⟦D⟧ (*emptyEnv*, env)

*elaborate* ⟦ε⟧ (locenv, env) = (locenv, env)

*elaborate* ⟦$D_1$ $D_2$⟧ = (*elaborate* ⟦$D_2$⟧) ∘ (*elaborate* ⟦$D_1$⟧)

*elaborate* ⟦**const** I **=** E⟧ (locenv, env) = if *applyEnv*(locenv,I) = *unbound*

    then (*extendEnv*(locenv,I,*typify* ⟦E⟧ env),*extendEnv*(env,I,*typify* ⟦E⟧ env))

    else *error*

*elaborate* ⟦**var** I **:** T⟧ (locenv, env) = if *applyEnv*(locenv,I) = *unbound*

    then (*extendEnv*(locenv,I,*type* (T)),*extendEnv*(env,I,*type* (T)))

    else *error*

*elaborate* ⟦**var** I, L **:** T⟧ = (*elaborate* ⟦**var** L **:** T⟧) ∘ (*elaborate* ⟦**var** I **:** T⟧)

---

*Figure 9.21*: Checking Context Constraints in Pelican (Part 1)

As declarations are processed, the environment for the current local block (locenv) and the cumulative environment (env) are constructed incrementally, adding a binding of an identifier to a type for each individual declaration while checking for multiple declarations of an identifier in the local en-

```
check ⟦C1 ; C2⟧ env = (check ⟦C1⟧ env) and (check ⟦C2⟧ env)
check ⟦skip⟧ env = true
check ⟦I := E⟧ env =
            (applyEnv (env,I) = intvar and typify ⟦E⟧ env = integer)
          or (applyEnv (env,I) = boolvar and typify ⟦E⟧ env = boolean)
check ⟦if E then C⟧ env = (typify ⟦E⟧ env = boolean) and (check ⟦C⟧ env)
check ⟦if E then C1 else C2⟧ env =
    (typify ⟦E⟧ env = boolean) and (check ⟦C1⟧ env) and (check ⟦C2⟧ env)
check ⟦while E do C⟧ env = (typify ⟦E⟧ env = boolean) and (check ⟦C⟧ env)
check ⟦declare B⟧ env = examine ⟦B⟧ env
check ⟦read I⟧ env = (applyEnv(I, env) = intvar)
check ⟦write E⟧ env = (typify ⟦E⟧ env = integer)
typify ⟦I⟧ env = case applyEnv(env,I) of
                    intvar, integer    : integer
                    boolvar, boolean   : boolean
                    program            : program
                    unbound            : error
typify ⟦N⟧ env = integer
typify ⟦true⟧ env = boolean
typify ⟦false⟧ env = boolean
typify ⟦E1 + E2⟧ env =
          if (typify ⟦E1⟧ env = integer) and (typify ⟦E2⟧ env = integer)
                    then integer else error
    :
typify ⟦E1 and E2⟧ env =
          if (typify ⟦E1⟧ env = boolean) and (typify ⟦E2⟧ env = boolean)
                    then boolean else error
    :
typify ⟦E1 < E2⟧ env =
          if (typify ⟦E1⟧ env = integer) and (typify ⟦E2⟧ env = integer)
                    then boolean else error
    :
```

*Figure 9.21*: Checking Context Constraints in Pelican (Part 2)

vironment. If an attempt is made to declare an identifier that is not *unbound* locally, the *error* value results. We assume that all semantic functions propagate the *error* value.

Checking commands involves finding Boolean or integer expressions where required and recursively checking sequences of commands that might occur. The semantic function *check* applied to a **declare** command just calls the *examine* function for the block. Simple expressions have their types determined directly. When we *typify* a compound expression, we must verify that its operands have the proper types and then specify the appropriate result type. If any part of the verification fails, *error* becomes the type value to be propagated.

A program satisfies the context-sensitive syntax of Pelican if *validate* produces true when applied to it. A final value of false or *error* means that the program does not fulfill the context constraints of the programming language.

The elaboration of the following Pelican program suggests the need for the local environment for context checking. Observe the difference if the Boolean variable is changed to "b". Note that the expressions "m+21" cannot be typified without access to the global environment, env.

| | **locnv** | **env** |
|---|---|---|
| **program** p **is** | [ ] | [ p \|→*program* ] |
| **const** m = 34; | [ m \|→*integer* ] | [ m \|→*integer*, p \|→*program* ] |
| **begin** | | |
| **declare** | [ ] | [ m \|→*integer*, p \|→*program* ] |
| **var** c : **boolean**; | [ c \|→*boolvar* ] | [ c \|→*boolvar*, m \|→*integer*, p \|→*program* ] |
| **const** c = m+21; | *error* | |
| **begin** | | |
| **write** m+c; | | |
| **end** | | |
| **end** | | |

## Exercises

1. Apply the *validate* semantic function to these Pelican programs and elaborate the definitions that check the context constraints for Pelican.

a)
```
program a is
  const c = 99;
  var n : integer;
begin
  read n;
  n := c-n;
  write c+1;
  write n
end
```

b)
```
program b is
  const c = 99;
  var b : boolean;
begin
  b := false;
  if b and true
     then b := c end if;
  b := c>0
end
```

c)
```
program c is
  var x,y,z : integer;
begin
  read x;
  y := z;
  declare
    var x,z : integer;
  begin
    while x>0 do
      x := x-1 end while;
    declare
      var x,y : boolean;
      const y = false;
    begin
      skip
    end
  end
end
```

d)
```
program d is
  var b : boolean;
  const c = true;
begin
  b := not(c) or false;
  read b;
  write 1109
end
```

e)
```
program e is
  var m,n : integer;
begin
  read m;
  n := m/5;
  write n+k
end
```

2. Extend the denotational semantics for context checking Pelican to include procedure declarations and calls.

3. Extend the result in exercise 2 to incorporate procedures with an arbitrary number of parameters.

4. Reformulate the denotational semantics for context checking Pelican using false in place of *error* and changing the signature of *elaborate* to

   *elaborate* : Declaration → Environment x Environment
   → Environment x Environment x Boolean

   Let *typify* applied to an expression with a type error or an unbound identifier take the value *unbound*.

5. Following the denotational approach in this section, implement a context checker for Pelican in Prolog.

## 9.7 CONTINUATION SEMANTICS

All the denotational definitions studied so far in this chapter embody what is known as **direct denotational semantics**. With this approach, each semantic equation for a language construct describes a transformation of argument domain values, such as environment and store, directly into results in some semantic domain, such as a new environment, an updated store, or an expressible value. Furthermore, the results from one construct pass directly to the language construct that immediately follows it physically in the code. The semantic equation for command sequencing shows this property best:

$$execute\ [\![C_1\ \mathbf{;}\ C_2]\!]\ sto = execute\ [\![C_2]\!]\ (execute\ [\![C_1]\!] sto).$$

Observe that this semantic equation has the second command $C_2$ working directly on the store produced by the first command. We return to Wren for these examples since the points to be made here do not depend on environments as found in Pelican. As an example, consider the following Wren program fragment processed by the semantic function *execute* : Command → Store → Store.

```
s := 0; n := 5;
while n>1 do
        s := s+n; n := n–2
end while;
mean := s/2
```

Figure 9.22 outlines the modifications through which the store progresses as *execute* is applied to this sequence of commands. At the same time we suppress the applications of *evaluate* that must be carried out in the analysis—for example, *evaluate* [[5]] and *evaluate* [[n>1]]. Our purpose is to illustrate the flow of data through commands in direct denotational semantics.

Although many language constructs have perspicuous descriptions in direct denotational semantics, two problems reduce the applicability of this approach:

1. When an error occurs in determining the meaning of a language construct, the error must propagate through all of the remaining denotational transformations in the definition of the construct in a program. If semantic equations detail all the aspects of this propagation, they become cluttered with error testing. We avoided such confusion in our semantic equations by informally describing the nature of error propagation at the cost

of lost precision in the definitions. Furthermore, most programming language implementations do not propagate errors in this manner; they abort (terminate) execution on finding a dynamic error. Of course, denotational definitions do not have to adhere to real implementations, but aborting execution is an easier way to handle errors, if we only have a way of describing it.

2. Most programing languages allow radical transfers of control during the execution of a program—in particular, by means of the **goto** command. Such constructs cannot be modeled easily with direct denotational semantics.

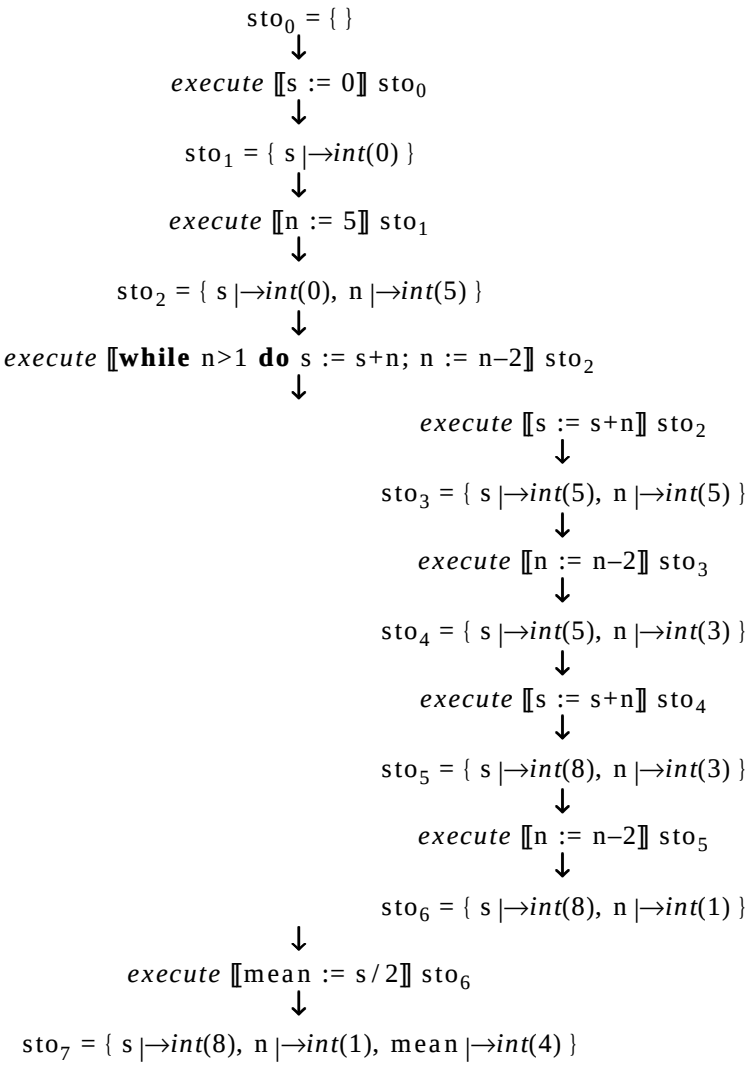


*Figure 9.22*: Passing the Store through a Denotational Analysis

Consider again the meaning of command sequencing, "*execute* $[\![C_1 \; ; \; C_2]\!]$". Direct denotational semantics assumes that $C_2$ will be executed immediately following $C_1$ and that it depends on receiving a store from "*execute* $[\![C_1]\!]$ sto". But what happens if $C_1$ does not pass control on to $C_2$, producing a new store for $C_2$ to act on? The reasons that $C_1$ may not be immediately followed by $C_2$ include the occurrence of a dynamic error in $C_1$, or because $C_1$ may belong to a class of commands, called **sequencers**, including **goto**, **stop**, **return**, **exit** (Ada or Modula-2), **break** (C), **continue** (C), **raise** (a language with exceptions), and **resume** (a language with coroutines). Sequencers have the property that computation generally does not proceed with the next command in the physical text of the program.

Returning to a concrete example, regard a sequence (block) of four labeled commands:

**begin** $L_1 : C_1;\ L_2 : C_2;\ L_3 : C_3;\ L_4 : C_4$ **end**.

With direct semantics, the sequence has as its meaning

*execute* $[\![C_4]\!] \circ$ *execute* $[\![C_3]\!] \circ$ *execute* $[\![C_2]\!] \circ$ *execute* $[\![C_1]\!]$,

if we ignore the denotations of the labels for now. As a store transformation, the sequence can be viewed as follows:

$\text{sto}_0 \rightarrow$ *execute* $[\![C_1]\!] \rightarrow$ *execute* $[\![C_2]\!] \rightarrow$ *execute* $[\![C_3]\!] \rightarrow$ *execute* $[\![C_4]\!] \rightarrow \text{sto}_{\text{final}}$.

But what if $C_3$ is the command "**goto** $L_1$"? Then the store transformation must develop as follows:

$\text{sto}_0 \rightarrow$ *execute* $[\![C_1]\!] \rightarrow$ *execute* $[\![C_2]\!] \rightarrow$ *execute* $[\![C_3]\!] \rightarrow$ *execute* $[\![C_1]\!] \rightarrow \ldots$.

To handle these two possibilities, "ex*ecute* $[\![C_3]\!]$" must be able to make the choice of sending its result, a store, on to "*execute* $[\![C_4]\!]$" or to somewhere else, such as "*execute* $[\![C_1]\!]$". Before describing how this choice can be made, we need to establish the meaning of a label. We take the label $L_k$, for k=1, 2, 3, or 4, to denote the computation starting with the command $C_k$ and running to the termination of the program. This meaning is encapsulated as a function from the current store to a final store for the entire program. Such functions are known as **continuations**, and a denotational definition involving them is called **continuation semantics** or **standard semantics**.

## Continuations

A continuation describes the change of state (store in this case) that occurs as a result of executing the program from a particular point until the program terminates; that is, a continuation models the **remainder of the program** from a point in the code. The semantic domain of continuations,

$$\text{Continuation} = \text{Store} \rightarrow \text{Store}$$

is included with the denotable values since they can be bound to identifiers (labels). Each label in a program is bound to a continuation in the environment of the block containing that label.

For the previous block with four commands and *no sequencers*, we have an environment env with the following bindings:

| Identifier | Denotable Value |
|---|---|
| $L_1$ | $cont_1$ = *execute* $[\![C_1; C_2; C_3; C_4]\!]$ env |
| $L_2$ | $cont_2$ = *execute* $[\![C_2; C_3; C_4]\!]$ env |
| $L_3$ | $cont_3$ = *execute* $[\![C_3; C_4]\!]$ env |
| $L_4$ | $cont_4$ = *execute* $[\![C_4]\!]$ env |

Note that each continuation depends on an environment that contains the bindings of all the labels being elaborated so that jumps anywhere in the block can be made when we allow sequencers. Therefore the signature of *execute* includes an environment domain:

*execute* : Command → Environment → Store → Store

This is not, however, the final signature for *execute* since we have one more issue to deal with.

Suppose that $C_2$ is not a sequencer. Then "*execute* $[\![C_2]\!]$" passes its resulting store to the rest of the computation starting at $C_3$—namely, the "normal" continuation $cont_3$. On the other hand, if $C_3$ is "**goto** $L_1$", it passes its resulting store to the continuation bound to $L_1$—namely, $cont_1$. To allow these two possibilities, we make the normal continuation an argument to "*execute* $[\![C_k]\!]$" to be executed with normal program flow (as with $C_2$) or to be discarded when a sequencer occurs (as with $C_3$). Therefore the final signature of *execute* has the form

*execute* : Command → Environment → Continuation → Store → Store,

and the corresponding semantic equation for command sequencing becomes

*execute* $[\![C_1$ **;** $C_2]\!]$ env cont sto = *execute* $[\![C_1]\!]$ env {*execute* $[\![C_2]\!]$ env cont} sto

where "*execute* $[\![C_2]\!]$ env cont" is the normal continuation for $C_1$. The continuation given to the execution of $C_1$ encapsulates the execution of $C_2$ followed by the execution of the original continuation. Traditionally, braces are used to delimit this constructed continuation. Observe the functionality of this normal continuation:

*execute* $[\![C_2]\!]$ env cont : Store → Store.

The semantic equation for the **goto** command shows that the continuation bound to the label comes from the environment and is executed with the store passed as a parameter,

*execute* $[\![$**goto** L$]\!]$ env cont sto = *applyEnv*(env,L) sto

with the effect that the normal continuation is simply discarded. In the previous example, where $C_2$ is **skip** and $C_3$ is "**goto** $L_1$",

*execute* $[\![C_2]\!]$ env $cont_3$ = $cont_3$

and *execute* $[\![C_3]\!]$ env $cont_4$ = *applyEnv*(env,$L_1$).

Observe that the store argument has been factored out of both of these semantic equations.

## The Programming Language Gull

We illustrate continuation semantics with Gull (G for **goto**), a programming language that is similar to Wren but contains two sequencers, **goto** and **stop**. Figure 9.23 provides the abstract syntax for Gull.

**Syntactic Domains**

| | | |
|---|---|---|
| P : Program | L : Label | O : Operator |
| S : Series | I : Identifier | N : Numeral |
| C : Command | E : Expression | |

**Abstract Production Rules**

Program ::= **program** Identifier **is begin** Series **end**

Series ::= Command

Command ::= Command **;** Command | Identifier **:=** Expression
| **if** Expression **then** Series **else** Series
| **while** Expression **do** Series | **skip** | **stop**
| **goto** Label | **begin** Series **end** | Label **:** Command

Expression ::= Identifier | Numeral | **-** Expression
| Expression Operator Expression

Operator ::= **+** | **–** | ***** | **/** | **=** | **<=** | **<** | **>** | **>=** | **<>**

Label ::= Identifier

*Figure 9.23*: Abstract Syntax of Gull

Gull permits only integer variables and has simply the **if-then-else** selection command for economy. The syntactic domain "Series" acts as the syntactic domain that corresponds to blocks, thus making the bodies of **if** and **while** commands into local scoping regions. A local environment for labels can also be created by an anonymous block using "**begin** Series **end**". Although a series is only a command in the abstract syntax, it serves as a separate syntactic category to allow for the elaboration of labels in the command. The syntax of Gull must have context constraints that forbid multiple labels with the same identifier in a series and a jump to an undefined label.

We need to elaborate labels at the top level and also inside compound commands, ensuring correct denotations for language constructs, such as those found in the program shown below. This poorly written program is designed to illustrate the environments created by labels in Gull. Figure 9.24 displays the nesting of the five environments that bind the labels in the program.

```
program labels is
  begin
        s:= 0; m := 1;                                   env1
  outer : if m<=5
              then  n := 1;                              env2
          inner :   if n<= 4
                      then  s := s+m *n;                 env3
                            goto next;
                    next :  n := n+1;
                            goto inner
                      else  m := m+1;                    env4
                            goto outer                   end if
              else  goto done                            env5   end if;
       done : answer := s
  end
```

*Figure 9.24*: A Gull Program

Figure 9.25 provides the semantic domains and the signatures of the semantic functions for the continuation semantics of Gull. As already mentioned, continuations are functions from store to store, and environments map labels to continuations, the only denotable values in Gull. Therefore we do not bother to put tags on them. The semantic functions indicate that only commands depend on continuations, sometimes called **command continuations**. A deeper investigation of continuation semantics would study **expression continuations** and **declaration continuations**. An expression continuation encapsulates the rest of the computation following the expression being evaluated. These are the continuations in Scheme, which allows the manipulation of such continuations as first-class objects in the same way as any other

functions. Declaration continuations are necessary to model an escape from the elaboration of a declaration, say because of an error. These other kinds of continuations are beyond the scope of this text. They are covered in other books (see the further readings at the end of the chapter).

**Semantic Domains**

EV = *int*(Integer) + *bool*(Boolean)

SV = *int*(Integer)

DV = Continuation

Store = Identifier → SV + *undefined*

Continuation = Store → Store

Environment = Label → Continuation + *unbound*

**Semantic Functions**

*meaning* : Program → Store

*perform* : Series → Environment → Continuation → Store → Store

*execute* : Command → Environment → Continuation → Store → Store

*evaluate* : Expression → Store → EV

*Figure 9.25*: Semantic Domains and Semantic Functions of Gull

## Auxiliary Functions

The semantic equations defining the semantic functions require a number of auxiliary functions that have been presented previously. When we specify the meaning of a program, an initial environment, *emptyEnv*, an initial store, *emptySto*, and an initial continuation, *identityCont*, must be supplied. The initial continuation is a function that takes the store resulting at the end of the execution of the entire program and produces the final "answer". We take the final store as the answer of a program, since Gull has no output. The initial continuation can thus be the identity function.

*emptySto* : Store

*updateSto* : Store x Identifier x SV → Store

*applySto* : Store x Identifier → SV

*emptyEnv* : Environment

*extendEnv* : Environment x Label$^+$ x Continuation$^+$ → Environment

*applyEnv* : Environment x Label → Continuation

*identityCont* : Continuation

*identityCont* = λ sto . sto.

Since the denotational semantics of Gull elaborates all the labels in a series in one semantic equation, *extendEnv* takes lists of labels and continuations as arguments. An exercise asks the reader to define this auxiliary function.

## Semantic Equations

The semantic equations for Gull are detailed in Figure 9.26. We examine the specification of *execute* first, assuming that the environment argument already contains bindings for all visible labels. Command sequencing has already been described. Executing the **skip** command makes no change in the store, so the current store is passed to the current continuation to continue the execution of the program. The **stop** command abandons the current continuation and returns the current store, which thereby becomes the final store terminating the denotational analysis. The **if** and **while** commands are analogues of those for direct semantics, passing the current continuation to appropriate series. The only exception occurs when the **while** test is false and the effect is like a **skip** command. The assignment command calls the current continuation with a store reflecting the new value that has been stored.

Since our denotational specification of Gull ignores expression continuations, the semantic equations for *evaluate* remain the same as those for Wren.

The function *perform*, specifying the meaning of a series, assumes a list of n commands, all possessing labels. It proceeds by binding each label to the appropriate continuation that encapsulates the rest of the code from the point of the label in the program together with an environment that includes all the bindings being established in the series. Observe that each defined continuation executes one command with the continuation that follows the command. The last continuation $cont_n$ executes the last command with the continuation that was originally passed to the series, representing the rest of the program following the series. Once the labels have been elaborated, the first continuation $cont_1$, which embodies the entire list of commands, is invoked.

## The Error Continuation

Gull does not handle errors any better than Wren, even though we suggested earlier that continuation semantics allows us to abort execution when a dynamic error occurs. To treat errors properly, we need expression continuations, so that when division by zero or accessing an undefined variable happens, an error continuation can be called at that point. Our specification of Gull has to inspect the result of evaluating an expression at the command

level and call an error (command) continuation there. The semantic equation for the assignment command then takes the form

*execute* ⟦I **:=** E⟧ env cont sto =
    if *evaluate* ⟦E⟧ sto=*error*
        then *errCont* sto else cont *updateSto*(sto,I,*evaluate* ⟦E⟧ sto).

---

meaning ⟦program I is begin S end⟧ =
        perform ⟦S⟧ *emptyEnv identityCont emptySto*

perform ⟦$L_1$: $C_1$**;** $L_2$**:** $C_2$**;** ... **;** $L_n$**:** $C_n$ ⟧ env cont = $cont_1$
    where $cont_1$ = *execute* ⟦$C_1$⟧ $env_1$ $cont_2$
        $cont_2$ = *execute* ⟦$C_2$⟧ $env_1$ $cont_3$
                :
        $cont_n$ = *execute* ⟦$C_n$⟧ $env_1$ cont
    and $env_1$ = *extendEnv*(env,[$L_1$, $L_2$, ... , $L_n$],[$cont_1$, $cont_2$, ... , $cont_n$])

*execute* ⟦I **:=** E⟧ env cont sto = cont *updateSto*(sto,I,*evaluate* ⟦E⟧ sto)
*execute* ⟦**skip**⟧ env cont sto = cont sto
*execute* ⟦**stop**⟧ env cont sto = sto
*execute* ⟦**if** E **then** $S_1$ **else** $S_2$ ⟧ env cont sto =
        if p then *perform* ⟦$S_1$⟧ env cont sto else *perform* ⟦$S_2$⟧ env cont sto
                where *bool*(p) = *evaluate* ⟦E⟧ sto
*execute* ⟦**while** E **do** S⟧ env cont sto = loop
    where loop env cont sto = if p then *perform* ⟦S⟧ env {loop env cont} sto
                else cont sto
            where *bool*(p) = *evaluate* ⟦E⟧ sto
*execute* ⟦$C_1$ **;** $C_2$⟧ env cont sto = *execute* ⟦$C_1$⟧ env {*execute* ⟦$C_2$⟧ env cont} sto
*execute* ⟦**begin** S **end**⟧ env cont sto = *perform* ⟦S⟧ env cont sto
*execute* ⟦**goto** L⟧ env cont sto = *applyEnv*(env,L) sto
*execute* ⟦L : C⟧ = *execute* ⟦C⟧

*evaluate* ⟦I⟧ sto = *applySto*(sto,I)
*evaluate* ⟦N⟧ sto = *value* ⟦N⟧
*evaluate* ⟦-E⟧ = *int*(*minus*(0,m)) where *int*(m) = *evaluate* ⟦$E_1$⟧ sto
*evaluate* ⟦$E_1$ + $E_2$⟧ sto = *int*(*plus*(m,n))
        where *int*(m) = *evaluate* ⟦$E_1$⟧ sto and *int*(n) = *evaluate* ⟦$E_2$⟧ sto
    **:**

---

*Figure 9.26*: Semantic Equations for Gull

For the **if** command, the value of "*evaluate* ⟦E⟧ sto" must be examined before executing one of the branches.

*execute* ⟦**if** E **then** $S_1$ **else** $S_2$⟧ env cont sto =
    if *evaluate* ⟦E⟧ sto=*error*
        then *errCont* sto
        else if p then *perform* ⟦$S_1$⟧ env cont sto
            else *perform* ⟦$S_2$⟧ env cont sto
                where *bool*(p) = *evaluate* ⟦E⟧ sto.

The error continuation *errCont* performs in the same way as the identity continuation, except that it should signal an error condition some way, say by displaying an error message. For our purposes, we simply take *errCont* = *identityCont*.

## Exercises

1. Give a definition of *extendEnv* for lists of identifiers and continuations.

2. Describe the continuations used in analyzing the following program denotationally, and give the bindings in its environment:

```
program fact is
begin
          f := 1; n := 6;
  start : if n>=1 then goto rest else stop end if;
   rest : f := f*n; n := n–1; goto start
end
```

3. Add an **exit** command to Gull and provide a denotational definition for its semantics. Executing an **exit** causes control to transfer to the point following the closest enclosing **begin**-**end** construct. Explain why it is not sufficient to exit from the enclosing series.

4. Define the denotational semantics of a programming language that combines the features of Gull and Pelican.

5. Construct a denotational interpreter for Gull in Prolog. See the further readings for a suggestion on handling continuations as Prolog structures.

## 9.8 FURTHER READING

Denotational semantics grew out of the tradition of mathematical logic, and early versions were characterized by single-letter identifiers, the Greek alphabet, and a heavy use of concise and sometimes cryptic mathematical notation. Under the influence of the principles of good software design, more recent expositions of denotational semantics possess enhanced readability as a result of the use of meaningful identifiers and the concepts of data abstraction.

One of the best descriptions of denotational semantics can be found in David Schmidt's book [Schmidt88], which covers most of the material that we describe in this chapter as well as additional material and examples treating compound data structures, applicative languages, expression continuations, and concurrency.

The traditional reference for denotational semantics has been the book by Joseph Stoy [Stoy77]. Many of the later books were based on his work. The books by Michael Gordon [Gordon79], Frank Pagan [Pagan81], and Lloyd Allison [Allison86] contain short but thorough explanations of denotational semantics with many good examples, although the dense notation in them requires careful reading. The books by Allison and Gordon have clear presentations of continuation semantics. Moreover, Allison discusses at some length the possibility of implementing a denotational definition using an imperative programming language to construct interpreters. His examples were one of the inspirations for our denotational interpreter of Wren written in Prolog.

Several textbooks published in the past few years provide additional examples of denotational specifications and a look at the various notational conventions employed in denotational semantics. These books, [Meyer90], [Watt91], and [Nielson92], are written at about the same level as our presentation. The Nielson book discusses implementing denotational definitions using the functional language Miranda. David Watt's very readable text uses notational conventions that are close to ours. Watt also suggests using denotational semantics to verify the context constraints on programming languages. His text contains a complete denotational specification of an imperative programming language called Triangle. He suggests using Standard ML as a vehicle for constructing a denotational interpreter for Triangle based on the specification. Watt coined the term **semantic prototyping** for the process of implementing formal specifications of programming languages. Susan Stepney gives a denotational semantics for a small imperative programming language and a hypothetical machine language (using continuation semantics). After describing a compiler from one the other, she verifies it relative to the formal specifications and implements the system in Prolog [Stepney93].

More formal treatments of denotational semantics can be found in [Mosses90], [Tennent91], and [Winskel93]. The book by Tennent contains an interesting discussion of compositionality. He suggests [Janssen86] for a historical review of the notion of compositional definitions. Tennent has also written an undergraduate textbook on the concepts of programming languages that is based on denotational principles [Tennent81].

Most of the descriptions of implementing denotational semantics have avoided the problems inherent in continuation semantics. For a short presentation of denotational interpreters that handle continuations, see [Slonneger93], where implementations in Standard ML and Prolog are explained and compared.