Chapter 4

Basic Structure of Denotational Definitions

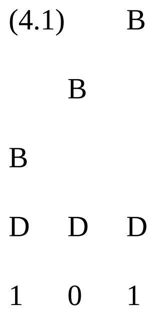
This chapter presents the format for denotational definitions. We use the abstract syntax and semantic algebra formats to define the appearance and the meaning of a language. The two are connected by a function called the *valuation function.* After giving an informal presentation of an application of the valuation function, we present the denotational semantics of two simple languages.

4.1 THE VALUATION FUNCTION

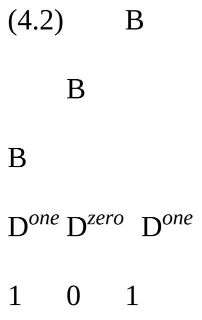
The valuation function maps a language's abstract syntax structures to meanings drawn from semantic domains. The domain of a valuation function is the set of derivation trees of a language. The valuation function is defined structurally. It determines the meaning of a derivation tree by determining the meanings of its subtrees and combining them into a meaning for the entire tree.

Some illustrations will make this point better than just words. A sentence in the language of binary numerals is depicted in Diagram 4.1.

(4.1)



(4.2)



The tree's internal nodes represent nonterminals in the language's abstract syntax definition:

B∈ Binary-numeral
D∈ Binary-digit
  B ::= BD | D
  D ::= 0 | 1

For this example, we take the somewhat artificial view that the individual binary digits are the ''words'' of a binary numeral ''sentence.''

The valuation function assigns a meaning to the tree by assigning meanings to its subtrees. We will actually use two valuation functions: **D**, which maps binary digits to their meanings, and **B**, which maps binary numerals to their meanings. The distinct valuation functions make the semantic definition easier to formulate and read.

Let's determine the meaning of the tree in Diagram 4.1 in a ‘‘bottom-up’’ fashion. First, the meaning of the digit subtree:

D

0

is the number *zero.* We might state this as:

D(D) = *zero*

0

That is, the **D** valuation function maps the tree to its meaning, *zero.* Similarly, the meanings of the other binary digits in the tree are *one*; that is:

D (D) = *one*

1

We will find it convenient to represent these two-dimensional equations in one-dimensional form, and we write:

$\mathbf{D}[\![0]\!] = \mathit{zero}$
$\mathbf{D}[\![1]\!] = \mathit{one}$

The double brackets surrounding the subtrees are used to clearly separate the syntax pieces from the semantic notation. The linearized form omits the D nonterminal. This isn't a problem, as the **D** valuation function maps only binary digits to meanings— D's presence is implied by **D**'s.

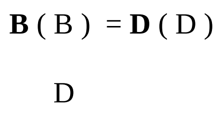
To help us note our progress in determining the meaning of the tree, Diagram 4.2 shows the meanings placed next to the nonterminal nodes. Now that we know the meanings of the binary digit subtrees, we next determine the meanings of the binary numeral trees. Looking at the leftmost B-tree, we see it has the form:

B

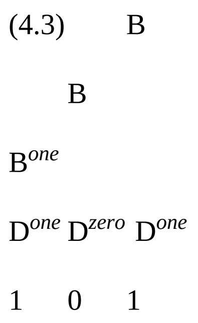
D^{one}

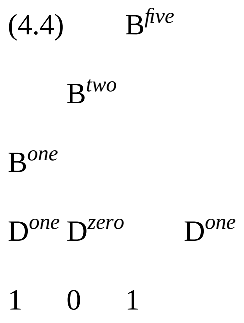
1

The meaning of this tree is just the meaning of its D-subtree, that is, *one.* In general, for any unary binary numeral subtree:

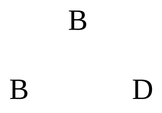


that is, **B**⟦D⟧ = **D**⟦D⟧. Diagram 4.3 displays the new information.

(4.3)


(4.4)


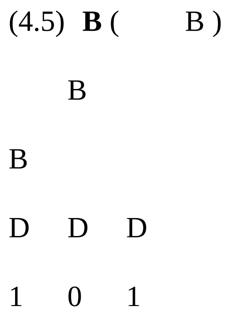
The other form of the binary numeral tree is:

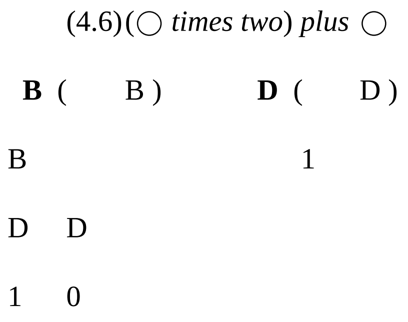


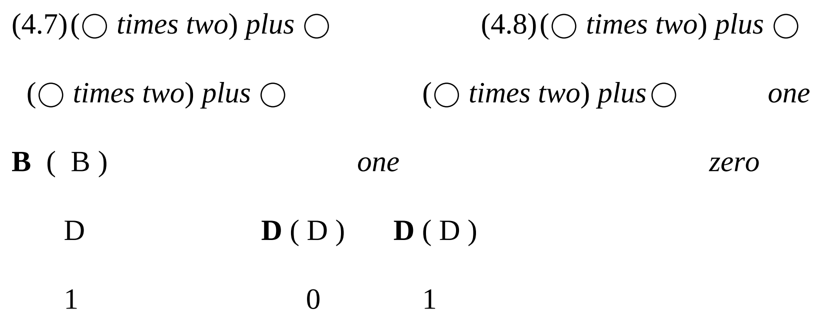
The principle of binary arithmetic dictates that the meaning of this tree must be the meaning of the left subtree doubled and added to the meaning of the right subtree. We write this as **B**⟦BD⟧ = (**B**⟦B⟧ *times two*) *plus* **D**⟦D⟧. Using this definition we complete the calculation of the meaning of the tree. The result, *five,* is shown in Diagram 4.4.

Since we have defined the mappings of the valuation functions on all of the options listed in the BNF rules for binary numerals, the valuation functions are completely defined.

We can also determine the meaning of the tree in Diagram 4.1 in a ‘‘top-down’’ fashion. The valuation functions are applied to the tree in Diagrams 4.5 through 4.8 and again show that its meaning is *five.*

(4.5)


(4.6)




4.2 FORMAT OF A DENOTATIONAL DEFINITION

A *denotational definition* of a language consists of three parts: the abstract syntax definition of the language, the semantic algebras, and the valuation function. As we saw in the previous section, the valuation function is actually a collection of functions, one for each syntax domain. A valuation function **D** for a syntax domain D is listed as a set of equations, one per option in the corresponding BNF rule for D.

Figure 4.1 gives the denotational definition of binary numerals.

The syntax domains are the ones we saw in the previous section. Only one semantic algebra is needed— the algebra of natural numbers *Nat.* Operations *minus* and *div* are not listed in the algebra, because they aren't used in the valuation functions.

It is instructive to determine once again the meaning of the tree in Diagram 4.1. We represent the tree in its linear form ⟦101⟧, using the double brackets to remind us that it is indeed a tree. We begin with:

B⟦101⟧ = (**B**⟦10⟧ *times two*) *plus* **D**⟦1⟧

The **B**⟦BD⟧ equation of the **B** function divides ⟦101⟧ into its subparts. The linear representation of ⟦101⟧ may not make it clear how to split the numeral into its two subparts. When in doubt, check back with the derivation tree! Checking back, we see that the division was performed correctly. We continue:

(**B**⟦10⟧ *times two*) *plus* **D**⟦1⟧
= (((**B**⟦1⟧ *times two*) *plus* **D**⟦0⟧) *times two*) *plus* **D**⟦1⟧
= (((**D**⟦1⟧ *times two*) *plus* **D**⟦0⟧) *times two*) *plus* **D**⟦1⟧
= (((*one times two*) *plus zero*) *times two*) *plus one*
= *five*

The derivation mimics the top-down tree transformation seen earlier.

The ‘‘bottom-up’’ method also maps ⟦101⟧ to *five.* We write a system of equations that defines the meanings of each of the subtrees in the tree:

D⟦0⟧ = *zero*
D⟦1⟧ = *one*

Figure 4.1

Abstract syntax:

$\mathrm{B} \in$ Binary-numeral
$\mathrm{D} \in$ Binary-digit

B ::= BD | D
D ::= 0 | 1

Semantic algebras:

I. Natural numbers
Domain $Nat = \mathbb{N}$
Operations

$zero,\ one,\ two,\ \cdots : Nat$
$plus,\ times : Nat \times Nat \to Nat$

Valuation functions:

B: Binary-numeral $\to Nat$

$\mathbf{B}[\![\mathrm{BD}]\!] = (\mathbf{B}[\![\mathrm{B}]\!]\ times\ two)\ plus\ \mathbf{D}[\![\mathrm{D}]\!]$
$\mathbf{B}[\![\mathrm{D}]\!] = \mathbf{D}[\![\mathrm{D}]\!]$

D: Binary-digit $\to Nat$

$\mathbf{D}[\![0]\!] = zero$
$\mathbf{D}[\![1]\!] = one$

$\mathbf{B}[\![1]\!] = \mathbf{D}[\![1]\!]$
$\mathbf{B}[\![10]\!] = (\mathbf{B}[\![1]\!]\ times\ two)\ plus\ \mathbf{D}[\![0]\!]$
$\mathbf{B}[\![101]\!] = (\mathbf{B}[\![10]\!]\ times\ two)\ plus\ \mathbf{D}[\![1]\!]$

If we treat each $\mathbf{D}[\![\mathrm{d}]\!]$ and $\mathbf{B}[\![\mathrm{b}]\!]$ as a variable name as in algebra, we can solve the simultaneous set of equations:

$\mathbf{D}[\![0]\!] = zero$
$\mathbf{D}[\![1]\!] = one$
$\mathbf{B}[\![1]\!] = one$
$\mathbf{B}[\![10]\!] = two$
$\mathbf{B}[\![101]\!] = five$

Again, we see that the meaning of the tree is *five*.

4.3 A CALCULATOR LANGUAGE

A calculator is a good example of a processor that accepts programs in a simple language as input and produces simple, tangible output. The programs are entered by pressing buttons on the device, and the output appears on a display screen. Consider the calculator pictured in Figure 4.2. It is an inexpensive model with a single ‘‘memory cell’’ for retaining a numeric value. There is also a conditional evaluation feature, which allows the user to enter a form of if-then-else expression.

A sample session with the calculator might go:

```
press    ON
press    ( 4 + 1 2 ) * 2
press    TOTAL                  (the calculator prints 32)
press    1 + LASTANSWER
press  TOTAL                    (the calculator prints 33)
press    IF LASTANSWER + 1 , 0 , 2 + 4
press    TOTAL                  (the calculator prints 6)
press    OFF
```

The calculator’s memory cell automatically remembers the value of the previous expression calculated so the value can be used in a later expression. The IF and , keys are used to build a conditional expression that chooses its second or third argument to evaluate based upon whether the value of the first is zero or nonzero. An excellent way of understanding the proper use of the calculator is to study the denotational semantics of its input language, which

Figure 4.2

is given in Figure 4.3.

The abstract syntax indicates that a session with the calculator consists of pressing the ON key and entering an expression sequence. An *expression sequence* is one or more expressions, separated by occurrences of TOTAL, terminated by the OFF key. The syntax for an expression follows the usual abstract syntax for arithmetic. Since the words of the language are numerals, no BNF rule is given for the syntax domain Numeral.

The semantic algebras show that the calculator reasons with two kinds of semantic objects: truth values and natural numbers. The phrase $n \in Nat$ in the *Nat* algebra's definition reminds us that all occurrences of identifier *n* in the valuation equations stand for an element of domain *Nat*. The same holds for the phrase $t \in Tr$. This convention is used in all the remaining semantic definitions in this book.

We can learn much about the calculator language from its semantic algebras. Apparently, the members of *Nat* will be the meanings of numerals and expressions in the calculator language, but *Tr* has no obvious representation of its elements in the syntax. This suggests that the calculator has some internal mechanism for doing logical reasoning, but the full power of that mechanism is not given to the user. This is confirmed by the presence of the *equals* operation in the algebra for *Nat*; the calculator can do arithmetic comparisons, but no comparison operator is included in the syntax. Therefore, we must take care to understand the syntactic construct whose semantic equation utilizes the *equals* operation.

There are four valuation functions for the language:

$\mathbf{P}$: Program$\rightarrow Nat^*$

$\mathbf{S}$: Expr-sequence$\rightarrow Nat \rightarrow Nat^*$

$\mathbf{E}$: Expression$\rightarrow Nat \rightarrow Nat$

$\mathbf{N}$: Numeral$\rightarrow Nat$

A good part of the calculator's semantics can be understood from studying the functionalities of the valuation functions. The **P** function maps a program to its meaning, which is a list of natural numbers. The reason for using the codomain Nat^* is found from the syntax: a program has the form [[ON S]], where [[S]] stands for a sequence of expressions. If each expression has a numeric value, then a sequence of them is list of numbers. The list represents the sequence of outputs displayed by the calculator during a session. This is confirmed by the functionality of **S**, which maps an expression sequence and a number to the desired number list. But what is the extra number used for? Recall that the calculator has a memory cell, which retains the value of the most recently evaluated expression. The number is the value in the memory cell. Perhaps the functionality should be written:

$\mathbf{S}$: Expr-sequence$\rightarrow Memory\text{-}cell \rightarrow Nat^*$, where $Memory\text{-}cell = Nat$

Two important features of denotational definitions are expressed in **S**'s functionality. First, the global data structures in a language's processor can be modelled as arguments to the valuation functions. There are no ''global variables'' for functions, so all such structures must be specified as arguments so that they are of use in the semantics. Second, the meaning of a syntactic construct can be a function. **S**'s functionality states that ''the meaning of an expression sequence is a function from a memory cell to a list of numbers.'' This seems confusing, for we might think that the meaning of an expression sequence is a list of numbers itself and not a function. The point is that the content of the memory cell is needed to evaluate the

Figure 4.3

Abstract syntax:

P∈ Program
S∈ Expr-sequence
E∈ Expression
N∈ Numeral

P ::= ON S

S ::= E TOTAL S | E TOTAL OFF

E ::= E_1+E_2 | E_1∗E_2 | IF E_1 , E_2 , E_3 | LASTANSWER | (E) | N

Semantic algebras:

I. Truth values
Domain $t \in Tr = \mathbb{B}$
Operations
true, *false*: *Tr*

II. Natural numbers
Domain $n \in Nat$
Operations
zero, *one*, *two*, $\cdots$: *Nat*
plus, *times* : $Nat \times Nat \rightarrow Nat$
equals : $Nat \times Nat \rightarrow Tr$

Valuation functions:

P: Program→ Nat^*

$\mathbf{P}[\![\text{ON S}]\!] = \mathbf{S}[\![\text{S}]\!](zero)$

S: Expr-sequence→ $Nat \rightarrow Nat^*$

$\mathbf{S}[\![\text{E TOTAL S}]\!](n) = \text{let } n' = \mathbf{E}[\![\text{E}]\!](n) \text{ in } n' \; cons \; \mathbf{S}[\![\text{S}]\!](n')$

$\mathbf{S}[\![\text{E TOTAL OFF}]\!](n) = \mathbf{E}[\![\text{E}]\!](n) \; cons \; nil$

E: Expression→ $Nat \rightarrow Nat$

$\mathbf{E}[\![\text{E}_1\text{+E}_2]\!](n) = \mathbf{E}[\![\text{E}_1]\!](n) \; plus \; \mathbf{E}[\![\text{E}_2]\!](n)$

$\mathbf{E}[\![\text{E}_1 * \text{E}_2]\!](n) = \mathbf{E}[\![\text{E}_1]\!](n) \; times \; \mathbf{E}[\![\text{E}_2]\!](n)$

$\mathbf{E}[\![\text{IF E}_1 \text{ , E}_2 \text{ , E}_3]\!](n) = \mathbf{E}[\![\text{E}_1]\!](n) \; equals \; zero \rightarrow \mathbf{E}[\![\text{E}_2]\!](n) \; [] \; \mathbf{E}[\![\text{E}_3]\!](n)$

$\mathbf{E}[\![\text{LASTANSWER}]\!](n) = n$

$\mathbf{E}[\![\text{(E)}]\!](n) = \mathbf{E}[\![\text{E}]\!](n)$

$\mathbf{E}[\![\text{N}]\!](n) = \mathbf{N}[\![\text{N}]\!]$

N: Numeral→ *Nat* (omitted— maps numeral N to corresponding $n \in Nat$)

sequence— the value in the cell may be accessed via the ‘‘LASTANSWER’’ key. The functionality notes the dependence of the value of the expression sequence upon the memory cell.

Let’s consider the semantic equations. The equation for $\mathbf{P}[\![\text{ON S}]\!]$ states that the meaning of a program session follows from the meaning of the expression sequence $[\![\text{S}]\!]$. The equation also says that the memory cell is initialized to *zero* when the calculator is turned on. The cell’s value is passed as an argument to the valuation function for the sequence.

As indicated by the functionality for **S**, an expression sequence uses the value of the memory cell to compute a list of numbers. The equation for $\mathbf{S}[\![\text{E TOTAL S}]\!]$ describes the meaning of a sequence of two or more expressions: the meaning of the first one, $[\![\text{E}]\!]$, is appended to the front of the list of values that follows from $[\![\text{S}]\!]$. We can list the corresponding actions that the calculator would take:

1. Evaluate $[\![\text{E}]\!]$ using cell n, producing value n'.
2. Print n' out on the display.
3. Place n' into the memory cell.
4. Evaluate the rest of the sequence $[\![\text{S}]\!]$ using the cell.

Note how each of these four steps are represented in the semantic equation:

1. is handled by the expression $\mathbf{E}[\![\text{E}]\!](n)$, binding it to the variable n'.
2. is handled by the expression n' *cons* $\cdots$.
3. and 4. are handled by the expression $\mathbf{S}[\![\text{S}]\!](n')$.

Nonetheless, the right-hand side of $\mathbf{S}[\![\text{E TOTAL S}]\!]$ is a mathematical value. Note that the same value is represented by the expression:

$$\mathbf{E}[\![\text{E}]\!](n) \; \textit{cons} \; \mathbf{S}[\![\text{S}]\!]\,(\mathbf{E}[\![\text{E}]\!](n))$$

which itself suggests that $[\![\text{E}]\!]$ be evaluated *twice.* This connection between the structure of function expressions and operational principles will be examined in detail in later chapters. In the meantime, it can be used to help understand the meanings denoted by the function expressions.

The meaning of $\mathbf{S}[\![\text{E TOTAL OFF}]\!]$ is similar. Since $[\![\text{E}]\!]$ is the last expression to be evaluated, the list of subsequent outputs is just *nil.*

Of the semantic equations for expressions, the ones for $[\![\text{LASTANSWER}]\!]$ and $[\![\text{IF E}_1, \text{E}_2, \text{E}_3]\!]$ are of interest. The $[\![\text{LASTANSWER}]\!]$ operator causes a lookup of the value in the memory cell. The meaning of the IF expression is a conditional. The *equals* operation is used here. The test value, $[\![\text{E}_1]\!]$, is evaluated and compared with *zero*. If it equals *zero*, $\mathbf{E}[\![\text{E}_2]\!](n)$ is taken as the value of the conditional, else $\mathbf{E}[\![\text{E}_3]\!](n)$ is used. Hence, the expression $[\![\text{E}_1]\!]$ in the first position of the conditional takes on a logical meaning in addition to its numeric one. This is a source of confusion in the calculator language and is a possible area for improvement of the language.

One last remark: equations such as $\mathbf{E}[\![(\text{E})]\!](n) = \mathbf{E}[\![\text{E}]\!](n)$ may also be written as $\mathbf{E}[\![(\text{E})]\!] = \lambda n.\, \mathbf{E}[\![\text{E}]\!](n)$, making use of the abstraction notation, or even as $\mathbf{E}[\![(\text{E})]\!] = \mathbf{E}[\![\text{E}]\!]$ (why?). This will be done in later examples.

A simplification of a sample calculator program is instructional:

$$\mathbf{P}[\![\text{ON 2+1 TOTAL IF LASTANSWER , 2 , 0 TOTAL OFF}]\!]$$
$$= \mathbf{S}[\![\text{2+1 TOTAL IF LASTANSWER , 2 , 0 TOTAL OFF}]\!](\textit{zero})$$

= let $n' =$ **E**⟦2+1⟧(*zero*)
  in n' *cons* **S**⟦IF LASTANSWER , 2 , 0 TOTAL OFF⟧(n')

Simplifying **E**⟦2+1⟧(*zero*) leads to the value *three*, and we have:

let $n' =$ *three* in n' *cons* **S**⟦IF LASTANSWER , 2 , 0 TOTAL OFF⟧(n')
= *three cons* **S**⟦IF LASTANSWER , 2 , 0 TOTAL OFF⟧(*three*)
= *three cons* (**E**⟦IF LASTANSWER , 2 , 0⟧(*three*) *cons nil*)

If we work on the conditional, we see that:

E⟦IF LASTANSWER , 2 , 0⟧(*three*)
= **E**⟦LASTANSWER⟧(*three*) *equals zero* → **E**⟦2⟧(*three*) [] **E**⟦0⟧(*three*)
= *three equals zero* → *two* [] *zero*
= *false* → *two* [] *zero* = *zero*

This gives as the final result the list:

three cons (*zero cons nil*)

Each of the simplification steps preserved the meaning of **P**⟦2+1 TOTAL IF LASTANSWER , 2 , 0 TOTAL OFF⟧. The purpose of simplification is to produce an equivalent expression whose meaning is more obvious than the original's. The simplification process is of interest in itself, for it shows how the calculator operates on input. If the denotational definition is used as a *specification* for the calculator, the definition plus simplification strategy show a possible *implementation* of the calculator. A simple-minded implementation would use functions, parameters, and an evaluation strategy corresponding to the simplification sequence just seen. It only takes a bit of insight, however, to notice that the numeral argument can be converted into a global memory cell. The derivation of a processor for a language from its denotational specification will be repeatedly touched upon in future chapters.

SUGGESTED READINGS

Jones 1982a; Gordon 1979; Milne & Strachey 1976; Pagan 1981; Stoy 1977; Tennent 1977, 1981

EXERCISES

1. Use the binary numeral semantics in Figure 4.1 to determine the meanings of the following derivation trees:

   a. ⟦0011⟧
   b. ⟦000⟧

c. $[\![111]\!]$

2. Here is an alternative abstract syntax for the language of binary numerals:

   N∈ Numeral
   B∈ Bin-Numeral
   D∈ Bin-digit
   N ::= B
   B ::= DB | D
   D ::= 0 | 1

   Define the valuation function that maps a binary numeral to its value. (Hint: define $\mathbf{P}$: Numeral $\rightarrow$ *Nat* and $\mathbf{B}$: Bin-numeral $\rightarrow$ (*Value* $\times$ *Scale*), where *Value* = *Nat* is the value of the numeral, and *Scale* = { *one*, *two*, *four*, *eight*, $\cdots$ } remembers the scale (physical size) of the numeral.)

3. a. In a fashion similar to that in Figure 4.1, define a denotational semantics for the language of base 8 numerals, Octal. Let **E** be the valuation function **E** : Octal $\rightarrow$ *Nat*.
   b. Prove the following equivalence: $\mathbf{E}[\![015]\!] = \mathbf{B}[\![1101]\!]$.
   c. Construct an algorithm that maps an octal numeral to binary form. Use the respective denotational semantics for Octal and Binary-numeral to prove that your algorithm is correct.

4. Simplify these calculator programs to their meanings in Nat^*:

   a. [[ON 1+(IF LASTANSWER , 4 , 1) TOTAL LASTANSWER TOTAL
   5∗2 TOTAL OFF]]
   b. [[ON 5 TOTAL 5 TOTAL 10 TOTAL OFF]]
   c. [[ON LASTANSWER TOTAL OFF]]

5. Augment the calculator so that it can compare two values for equality: add an = button to its panel and augment the BNF rule for Expression to read: E ::= $\cdots$ | E_1=E_2

   a. Write the semantic equation for $\mathbf{E}[\![E_1{=}E_2]\!]$.
   b. What changes must be made to the other parts of the denotational definition to accommodate the new construct? Make these changes. Do you think the new version of the calculator is an improvement over the original?

6. Alter the calculator semantics in Figure 4.3 so that the memory cell argument to **S** and **E** becomes a memory *stack*; that is, use Nat^* in place of *Nat* as an argument domain to **S** and **E**.

   a. Adjust the semantics so that the last answer is *pushed* onto the memory stack and the LASTANSWER button accesses the top value on the stack.
   b. Augment the syntax of the calculator language so that the user can explicitly pop values off the memory stack.

7. Use the denotational definition in Figure 4.3 to guide the coding of a test implementation of the calculator in Pascal (or whatever language you choose). What do the semantic algebras become in the implementation? How are the valuation equations realized? What does the memory cell become? What questions about the implementation *doesn't* the denotational definition answer?

8. Design, in the following stages, a calculator for manipulating character string expressions:

   a. List the semantic algebras that the calculator will need.
   b. List the operations that the calculator will provide to the user.
   c. Define the abstract syntax of these operations.
   d. Define the valuation functions that give meaning to the abstract syntax definition.

   Can these four steps be better accomplished in another order? Is the order even important?

9. If you are familiar with attribute grammars, describe the relationship between a denotational definition of a language and its attribute grammar definition. What corresponds to inherited attributes in the denotational definition? What are the synthesized attributes? Define attribute grammars for the binary numerals language and the calculator language.

10. Consider the compiler-oriented aspects of a denotational definition: if there existed a machine with hardwired instructions *plus*, *times*, and representations of numbers, the definition in Figure 4.1 could be used as a syntax-directed translation scheme for binary numbers to machine code. For example:

    $$\mathbf{B}[\![101]\!] = (((\mathit{one\ times\ two})\ \mathit{plus\ zero})\ \mathit{times\ two})\ \mathit{plus\ one}$$

    is the ‘‘compiled code’’ for the input program [[101]]; the syntax pieces are mapped to their denotations, but no simplifications are performed. The machine would evaluate this program to *five*. With this idea in mind, propose how a compiler for the calculator language of Figure 4.3 might be derived from its denotational definition. Propose a machine for executing compiled calculator programs.