THE CALCULATOR

After looking at the basics of the denotational method with the binary numerals we will take Schmidt’s example of a calculator to bring us one step closer to looking at the semantics of a real language. The calculator we shall study is essentially a two-function calculator that operates on natural numbers only (no negatives). It is unusual in that it has a conditional built into it, and it can handle numbers of any size. How it does this is a mystery, but we are only interested in semantics, not in engineering. It looks like this:

At the top is a display with two scroll buttons (for those long numbers). There are ON and OFF buttons, the usual digit buttons, 0-9, and two operator buttons for addition and multiplication. The IF and the comma buttons work together as we shall see. TOTAL makes the calculator compute the result of the previously input expression. The LASTANSWER button recalls the value of the last calculation and inserts the value into the current one.

SYNTAX

A real calculator has all kinds of problems to solve when the user pushes the wrong buttons. For instance what happens the + button is pushed twice in a row? What happens if the parentheses are not matched? What happens if the IF form is not used correctly? We are going to ignore all of these complications, and instead only allow correctly formed calculations. We can do this because our first step in syntax is to describe correct forms, not the incorrect ones. The abstract syntax is then:

\[
\begin{align*}
P & \in \text{Program} \\
S & \in \text{Expression-sequence} \\
E & \in \text{Expression} \\
N & \in \text{Numeral} \\
D & \in \text{Digit} \\
P & ::= \text{ON } S \\
S & ::= E \ \text{TOTAL } S \mid E \ \text{TOTAL OFF} \\
E & ::= E_1 + E_2 \mid E_1 \ast E_2 \mid N \mid (E) \mid \text{IF } E_1, E_2, E_3 \mid \text{LASTANSWER} \\
N & ::= N \ D \mid D \\
D & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0
\end{align*}
\]
Clearly in this grammar the terminal symbols are given by what is on the button (ON, LASTANSWER, etc.); P, S, E and D are the main non-terminals.

A typical ‘program’ would be: ON 1 + 2 TOTAL 3 * LASTANSWER TOTAL OFF, which would show 3 and then 9 in the display. A program can thus contain many calculations; something we will have to show in the semantics.

**SEMANTIC ALGEBRAS**

This calculator only deals with natural numbers, so clearly the domain $\text{Nat}$ used in the binary numeral example is also used here. In addition to arithmetic, we will also need an equality operation for the IF test. The domain is then:

Domain: $\text{Nat} = \mathbb{N}$

Operations:
- $\text{zero}, \text{one}, \text{two}, \ldots : \text{Nat}$
- $\text{plus} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$
- $\text{times} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$
- $\text{equals} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Tr}$

We will need the Boolean domain for the last operation:

Domain: $\text{Tr} = \mathbb{B}$

Operations:
- $\text{true}, \text{false} : \text{Tr}$

This last operation is the conditional form that we used to extend the lambda calculus. Notice that we have written all the functions out in curried form, without the cross product.

**VALUATION FUNCTIONS**

Following the method used for the binary numerals, each non-terminal in the grammar has a corresponding function, overloaded with each of the alternatives that define the non-terminal. We need to define the functionality of each before we start, so that each alternative equation is correctly defined. Starting with the syntactic category, Program, we need to decide what its function will return. We have three choices. We could have it return nothing, since the calculator has done its job when the OFF button is pushed, or we could have it return the last value calculated. Neither of these choices capture the whole story, so we will choose to have it return a list of all the values calculated. The denotation of our program will then be $\text{Nat}^*$, or a list of all the values calculated. So, we have:

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

The list is actually a domain with the following definition:

Domain: $\text{List} = A^*$, where $A$ is any domain

Operations:
- $\text{nil} : \text{List}$
- $\text{hd} : \text{List} \rightarrow A$
  \[ \text{hd}(a_1 \text{cons} a_2 \cdots \text{nil}) = a_1 \]
- $\text{tl} : \text{List} \rightarrow \text{List}$
  \[ \text{tl}(a_1 \text{cons} a_2 \cdots \text{nil}) = a_2 \cdots \text{nil} \]
- $\text{cons} : A \rightarrow \text{List} \rightarrow \text{List}$
  \[ \text{cons}(a_0, a_1 \text{cons} a_2 \cdots \text{nil}) = a_0 \text{cons} a_1 \cdots \text{nil} \]
This is reminiscent of the language Lisp, or Scheme, which, of course, have lambda calculus as their origin. We can define these operations in pure lambda calculus, but the expressions then become overly complicated, so we will omit them, just as we omit the definitions of arithmetic. \(hd\) returns the first item of the list, \(tl\) returns the list without the first item, and \(cons\) adds an item to the front of an existing list. \(nil\) is the empty list. We will only need \(cons\) and \(nil\) for our example, in order to form the list of values to be returned.

The expression sequence, \(S\), actually produces this list of values, but it has a hidden parameter, since any one of the expression in the sequence can use the value from LASTANSWER. Thus we have:

\[
S : \text{Expression-sequence} \times \text{Nat} \rightarrow \text{Nat}^*
\]

or, \(S : \text{Expression-sequence} \rightarrow \text{Nat} \rightarrow \text{Nat}^*\) in curried form. The parentheses surround the ‘output’, while the rest are ‘inputs’.

The expression function is easy now, since it only produces one value:

\[
E : \text{Expression} \rightarrow \text{Nat} \rightarrow \text{Nat}
\]

Again, the extra parameters are for LASTANSWER.

Numerals are easy, and the function for digits is trivial:

\[
N : \text{Numeral} \rightarrow \text{Nat}
\]

\[
D : \text{Digit} \rightarrow \text{Nat}
\]

Let us finish up the equations for the constants first:

\[
D[0] =\text{zero}
\]

\[
\quad \ldots
\]

\[
D[9] =\text{nine}
\]

\[
N[N D] = (\text{ten times } N[N]) \text{ plus } D[D]
\]

\[
N[D] = D[D]
\]

Expressions can be simple:

\[
E[N](n) = N[N]
\]

\[
E[(E)](n) = E[E](n)
\]

The argument \(n\) is the value of the calculator’s memory that can be used by pressing LASTANSWER:

\[
E[LASTANSWER](n) = n
\]

Arithmetic is now easy:

\[
E[E_1 + E_2](n) = E[E_1](n) \text{ plus } E[E_2](n)
\]

\[
E[E_1 \times E_2](n) = E[E_1](n) \text{ times } E[E_2](n)
\]

The conditional form \(\text{IF } E_1, E_2, E_3\) will be given the semantics that if the first expression evaluates to zero, then the value of the expression will be the second expression, otherwise the value will be the third one. This is where the \(\text{equals} \) operations comes in:

\[
E[\text{IF } E_1, E_2, E_3](n) = (E[E_1](n) \text{ equals zero}) \rightarrow E[E_2](n) \text{ } E[E_3](n)
\]

We finish up with the expression sequence:

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

\[
S[E \text{ TOTAL OFF}](n) = E[E](n) \text{ cons nil}
\]
Notice how the result of calculating the first expression in the sequence is passed to the rest of the sequence, but is remembered as the head of the list. The final one in the sequence need not be passed on since the OFF button has been pushed.

Finally the program itself:

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

where the memory of the calculator is initialized to zero for the first expression in the sequence.

**SUMMARY OF SEMANTIC DEFINITION**

It is instructive to collect all the definitions together in one place to see the whole picture. We do this below:

1. **Syntax**
   
   \( P \in \text{Program} \)
   
   \( S \in \text{Expression-sequence} \)
   
   \( E \in \text{Expression} \)
   
   \( N \in \text{Numeral} \)
   
   \( D \in \text{Digit} \)
   
   \( P ::= \text{ON } S \)
   
   \( S ::= E \ \text{TOTAL} \ S | E \ \text{TOTAL} \ \text{OFF} \)
   
   \( E ::= E_1 + E_2 | E_1 \ * \ E_2 | N | ( E ) | \text{IF } E_1 , E_2 , E_3 | \text{LASTANSWER} \)
   
   \( N ::= N \ D \ | D \)
   
   \( D ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 \)

2. **Semantic algebras**
   
   **Domain:** \( \mathbb{N} = \mathbb{N} \)
   
   **Operations:**
   
   \( \text{zero}, \text{one}, \text{two}, \ldots : \mathbb{N} \)
   
   \( \text{plus} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \)
   
   \( \text{times} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \)
   
   \( \text{equals} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{B} \)
   
   **Domain:** \( \mathbb{B} = \mathbb{B} \)
   
   **Operations:**
   
   \( \text{true}, \text{false} : \mathbb{B} \)
   
   \( \_ \rightarrow \_ : \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} \), for any domain \( A \).
   
   **Domain:** \( \text{List} = A^* \), where \( A \) is any domain
   
   **Operations:**
   
   \( \text{nil} : \text{List} \)
   
   \( \text{hd} : \text{List} \rightarrow A \)
   
   \( \text{tl} : \text{List} \rightarrow \text{List} \)
   
   \( \text{cons} : A \rightarrow \text{List} \rightarrow \text{List} \)

3. **Valuation functions**
P: Program $\rightarrow$ Nat$^*$

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

S: Expression-sequence $\rightarrow$ Nat $\rightarrow$ Nat$^*$

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

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

E: Expression $\rightarrow$ Nat $\rightarrow$ Nat

$$E[E_1 + E_2](n) = E[E_1](n) \text{ plus } E[E_2](n)$$

$$E[E_1 \times E_2](n) = E[E_1](n) \text{ times } E[E_2](n)$$

$$E[N](n) = N[N]$$

$$E[(E)](n) = E[E](n)$$

$$E[\text{IF } E_1, E_2, E_3](n) = (E[E_1](n) \text{ equals zero} ) \rightarrow E[E_2](n) \text{ nil } E[E_3](n)$$

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

N: Numeral $\rightarrow$ Nat

$$N[N D] = (\text{ten times } N[N]) \text{ plus } D[D]$$

$$N[D] = D[D]$$

D: Digit $\rightarrow$ Nat

$$D[0] = \text{zero}$$

$$\ldots$$

$$D[9] = \text{nine}$$

EXAMPLE DERIVATION

In order to show how the functions work, let us take the example program ON 1 + 2 TOTAL 3 * LASTANSWER TOTAL OFF and derive its denotation:

$$P[\text{ON } 1 + 2 \text{ TOTAL } 3 \times \text{ LASTANSWER } \text{ TOTAL } \text{ OFF}]$$

$$= S[1 + 2 \text{ TOTAL } 3 \times \text{ LASTANSWER } \text{ TOTAL } \text{ OFF}](\text{zero})$$

$$= \text{let } n' = E[1 + 2](\text{zero}) \text{ in } n' \text{ cons } S[3 \times \text{ LASTANSWER } \text{ TOTAL } \text{ OFF}](n')$$

At this point we can work on the first expression to find what $n'$ is:

$$E[1 + 2](\text{zero})$$

$$= E[1](\text{zero}) \text{ plus } E[2](\text{zero})$$

$$= N[1] \text{ plus } N[1]$$

$$= D[1] \text{ plus } D[1]$$

$$= \text{one plus two}$$

We could reduce this to three at this point or leave it unevaluated. This is the difference between the normal order of reduction and the applicative order. Here we will use the applicative order, and pass three to the rest of the sequence:
three cons \( S \left[ 3 \ast \text{LASTANSWER} \right] \) \( (three) \)

\[
= \text{three cons} \left( E \left[ 3 \ast \text{LASTANSWER} \right] \right) \ast \text{cons nil}
\]

Again, we work on the expression separately:

\[
E \left[ 3 \ast \text{LASTANSWER} \right] \ (three)
\]

\[
= E \left[ 3 \right] \ (three) \ast \text{times} \ E \left[ \text{LASTANSWER} \right] \ (three)
\]

\[
= N \left[ 3 \right] \ast \text{times} \ three
\]

\[
= D \left[ 3 \right] \ast \text{times} \ three
\]

\[
= \text{three times} \ three
\]

\[
= \text{nine}
\]

The whole derivation thus reduces to:

\[
\text{three cons nine cons nil}
\]

which is the program’s static denotation. If we had left the domain algebra expression unevaluated, we would have had:

\[
\left( \text{one plus two} \right) \ast \text{cons} \left( \text{three times} \left( \text{one plus two} \right) \right) \ast \text{cons nil}
\]

In a sense this is like the compiled version of the program. The syntax has disappeared and the result is a more primitive program running on the calculator’s ‘virtual machine’, which is purely abstract in this case. Notice that, denotationally speaking, \( \text{three cons nine cons nil} \) and \( \left( \text{one plus two} \right) \ast \text{cons} \left( \text{three times} \left( \text{one plus two} \right) \right) \ast \text{cons nil} \) are identical, since they both denote the same list. The difference is purely in the level of representation.

**LAMBDA FORMS FOR VALUATION FUNCTIONS**

Every valuation function has an alternative expansion of a syntactic category as its first argument. Where there are other arguments necessary, as in the functions for \( S \) and \( E \), then we can write these using the lambda notation:

\[
S \left[ E \ \text{TOTAL} \ S \right] = \lambda n. \ \text{let} \ n' = E \left[ E \right] \ (n) \ \text{in} \ n' \ \ast \text{cons} \ S \left[ S \right] \ (n')
\]

\[
S \left[ E \ \text{TOTAL} \ \text{OFF} \right] = \lambda n. E \left[ E \right] \ (n) \ \ast \text{cons nil}
\]

and

\[
E \left[ E_1 + E_2 \right] = \lambda n. E \left[ E_1 \right] \ (n) \ \ast \text{plus} \ E \left[ E_2 \right] \ (n)
\]

\[
E \left[ E_1 \ast E_2 \right] = \lambda n. E \left[ E_1 \right] \ (n) \ \ast \text{times} \ E \left[ E_2 \right] \ (n)
\]

Sometimes the lambda version is easier to read than the other, but it is a choice we can make if you we want to. Where the argument is present on both sides, it is technically possible to omit it. e.g.

\[
E \left[ \left( E \right) \right] = E \left[ E \right]
\]

This is possible because of the extensionality of functions. \( E \) still has functionality \( \text{Expression} \rightarrow \text{Nat} \rightarrow \text{Nat} \), so both side of the equation have functionality \( \text{Nat} \rightarrow \text{Nat} \). Thus they both need \( \text{Nat} \) to produce their final result.