


7. Introduction to Denotational Semantics

Oscar Nierstrasz

[[SEMANTICS]]

of a structure

**[[

[[**

Roadmap

- > Syntax and Semantics
- > Semantics of Expressions
- > Semantics of Assignment
- > Other Issues



References

- > D. A. Schmidt, *Denotational Semantics*, Wm. C. Brown Publ., 1986
- > D. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990

Roadmap

- > **Syntax and Semantics**
- > Semantics of Expressions
- > Semantics of Assignment
- > Other Issues



Defining Programming Languages

There are three main characteristics of programming languages:

1. **Syntax:** What is the *appearance and structure* of its programs?
2. **Semantics:** What is the *meaning* of programs?
The static semantics tells us *which (syntactically valid) programs are semantically valid* (i.e., which are type correct) and the dynamic semantics tells us *how to interpret the meaning of valid programs*.
3. **Pragmatics:** What is the *usability* of the language? How *easy is it to implement*? What *kinds of applications* does it suit?

Uses of Semantic Specifications

Semantic specifications are useful for language designers to communicate with implementors as well as with programmers.

A precise standard for a computer implementation:

How should the language be *implemented* on different machines?

User documentation:

What is the *meaning* of a program, given a particular combination of language features?

A tool for design and analysis:

How can the language definition be tuned so that it can be *implemented efficiently*?

Input to a compiler generator:

How can a *reference implementation* be obtained from the specification?

Methods for Specifying Semantics

Operational Semantics:

- > $\llbracket \text{program} \rrbracket = \textit{abstract machine program}$
- > can be simple to implement
- > hard to reason about

Denotational Semantics:

- > $\llbracket \text{program} \rrbracket = \textit{mathematical denotation}$
- > (typically, a function)
- > facilitates reasoning
- > not always easy to find suitable semantic domains

Axiomatic Semantics:

- > $\llbracket \text{program} \rrbracket = \textit{set of properties}$
- > good for proving theorems about programs
- > somewhat distant from implementation

Structured Operational Semantics:

- > $\llbracket \text{program} \rrbracket = \textit{transition system}$
- > (defined using inference rules)
- > good for concurrency and non-determinism
- > hard to reason about equivalence

The “semantic brackets” notation is commonly used as follows:

$$\llbracket \langle \textit{program fragment} \rangle \rrbracket = \langle \textit{mathematical object} \rangle$$

Sometimes the semantic brackets have additional subscripts, superscripts or arguments to indicate additional information that is needed to determine the meaning of the code fragment (for example, a table of the names and types of declared variables).

As we shall see, there will typically be different semantic brackets for different kinds of program elements (e.g, programs, statements, expressions, modules, classes, etc.).

The first attempts to specify the *operational semantics* of programming languages in the 1960s did so by translating programs to abstract machines. Such specifications are useful as a reference for implementing a compiler or interpreter, but can be very difficult to use as a basis for proving properties of programs. The semantics of the lambda calculus is specified *operationally* in terms of reductions (i.e., *transitions*) over syntactic lambda expressions. Note that this does *not* yield a mathematical object (i.e., a denotation) that represents the meaning of a lambda expression. This is a problem because we have no way of saying whether two lambdas have the same semantics. Dana Scott and Christopher Strachey laid the foundations of *denotational semantics* in the 1970s to address this problem.

Axiomatic semantics attempts to understand the meaning of programs in terms of *algebras* of mathematical objects with operators over them. Considerable work on modeling concurrency using “process algebras” took place in the 1980s. Programs are modeled as communicating “processes” that can be composed using various algebraic operators. Various theorems allow process expressions to be rewritten to other, equivalent forms.

In recent years, most work on semantics has adopted the *structural operational style* that we saw briefly in the slide on Featherweight Java. This has a strong affinity to the lambda calculus in that semantics of programs is described operationally in terms of transitions between program states. The states may be syntactic expressions (as in the lambda calculus) or states of some kind of abstract machine (or even a combination of the two).

In this lecture we will focus on denotational semantics. Programs and their components will be given meaning in terms of mathematical objects, i.e., numbers, sets, functions etc.

As we shall see, this approach is especially nice for specifying the semantics of plain, imperative languages, and offers a nice basis for translating the semantics to a straightforward language implementation.

Things get complicated, however, when we start to consider issues like objects, exceptions, concurrency, distribution, and so on. For this reason, denotational semantics is preferred only for relatively simple languages.

Roadmap

- > Syntax and Semantics
- > **Semantics of Expressions**
- > Semantics of Assignment
- > Other Issues



Concrete and Abstract Syntax

How to parse “4 * 2 + 1”?

Abstract Syntax is *compact but ambiguous*:

Expr ::= Num | Expr Op Expr
Op ::= + | - | * | /

Concrete Syntax is *unambiguous but verbose*:

Expr ::= Expr LowOp Term | Term
Term ::= Term HighOp Factor | Factor
Factor ::= Num | (Expr)
LowOp ::= + | -
HighOp ::= * | /

Concrete syntax is needed for parsing; abstract syntax suffices for semantic specifications.

Concrete syntax refers to the detailed, unambiguous grammar needed to parse (i.e., recognize) program source into a structured form. The parsed structure may be a concrete syntax tree representing the fine details of the concrete grammar, but more usually it is expressed as an *abstract syntax tree* (AST) that eliminates details that are not interesting after the code has already been parsed.

In the example, once we know the structure of the parsed expression, it is no longer of interest to us that multiplication and division have higher precedence than addition and subtraction. We can group them all together as *operators*.

In short: concrete syntax is for parsers, abstract syntax is for semantics.

A Calculator Language

Abstract Syntax:

```
Prog ::= 'ON' Stmt
Stmt ::= Expr 'TOTAL' Stmt
      | Expr 'TOTAL' 'OFF'
Expr  ::= Expr1 '+' Expr2
      | Expr1 '*' Expr2
      | 'IF' Expr1 ', ' Expr2 ', ' Expr3
      | 'LASTANSWER'
      | '(' Expr ')'
      | Num
```

The program “ON 4 * (3 + 2) TOTAL OFF” should print out 20 and stop.

This language represents a programmatic interface to a calculator. Programs start by turning the calculator on (keyword ON), proceed by evaluation a number of expressions, and conclude by turning the calculator off (keyword OFF).

Each expression is computed and printed when the keyword TOTAL is encountered. The value of the last computed expression is available in the special register LASTANSWER.

The *meaning of a program P* is therefore the *list of numbers* printed out.

Calculator Semantics

We need three semantic functions: one for programs, one for statements (expression sequences) and one for expressions.

The meaning of a program is the list of integers printed:

Programs:

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

$$P \llbracket \text{ON } S \rrbracket = S \llbracket S \rrbracket (0)$$

A statement may use and update LASTANSWER:

Statements:

$$S : \text{ExprSequence} \rightarrow \text{Int} \rightarrow \text{Int}^*$$

$$S \llbracket \text{E TOTAL } S \rrbracket (n) = \text{let } n' = E \llbracket E \rrbracket (n) \text{ in } \text{cons}(n', S \llbracket S \rrbracket (n'))$$

$$S \llbracket \text{E TOTAL OFF} \rrbracket (n) = [E \llbracket E \rrbracket (n)]$$

Calculator Semantics...

Expressions:

$E : \text{Expression} \rightarrow \text{Int} \rightarrow \text{Int}$

$$E \llbracket E1 + E2 \rrbracket (n) = E \llbracket E1 \rrbracket (n) + E \llbracket E2 \rrbracket (n)$$

$$E \llbracket E1 * E2 \rrbracket (n) = E \llbracket E1 \rrbracket (n) \times E \llbracket E2 \rrbracket (n)$$

$$E \llbracket \text{IF } E1, E2, E3 \rrbracket (n) = \begin{array}{l} \text{if } E \llbracket E1 \rrbracket (n) = 0 \\ \text{then } E \llbracket E2 \rrbracket (n) \\ \text{else } E \llbracket E3 \rrbracket (n) \end{array}$$

$$E \llbracket \text{LASTANSWER} \rrbracket (n) = n$$

$$E \llbracket (E) \rrbracket (n) = E \llbracket E \rrbracket (n)$$

$$E \llbracket N \rrbracket (n) = N$$

We have three different sets of semantic brackets:

- $\mathbf{P}[\langle program \rangle]$ interprets a program as a list of numbers
- $\mathbf{S}[\langle statement \rangle]$ interprets a statements as a lists of numbers (given a LASTANSWER)
- $\mathbf{E}[\langle expression \rangle]$ interprets an expression as a number (given a LASTANSWER)

Each semantic function is defined in terms of simpler ones, by decomposing the structure of the syntactic argument (program is broken into statements, and so on).

The semantic functions for statements and expressions need extra arguments to represent the state of the computation, in this case n , the value of the LASTANSWER register.

Note the distinction between the *syntactic operators* (+ and *) appearing as arguments to $\mathbf{E}[\]$ and the *semantic* ones appearing in the interpretation (+ and \times).

Semantic Domains

In order to define semantic mappings of programs and their features to their mathematical denotations, the semantic domains must be precisely defined:

```
data Bool = True | False
(&&), (||) :: Bool -> Bool -> Bool
False && x = False
True && x = x
False || x = x
True || x = True

not :: Bool -> Bool
not True = False
not False = True
```

In denotational semantics, we map programs to *semantic domains*, i.e., sets of mathematical objects whose behavior is precisely defined. For our language, we are mapping programs to the domain of functions, but these in turn need Booleans, so we should be precise about how they are defined.

Actually we will cheat here and just use the Haskell Booleans, but we could just as well define our own Booleans, as shown here.

Data Structures for Abstract Syntax

We can represent programs in our calculator language as syntax trees:

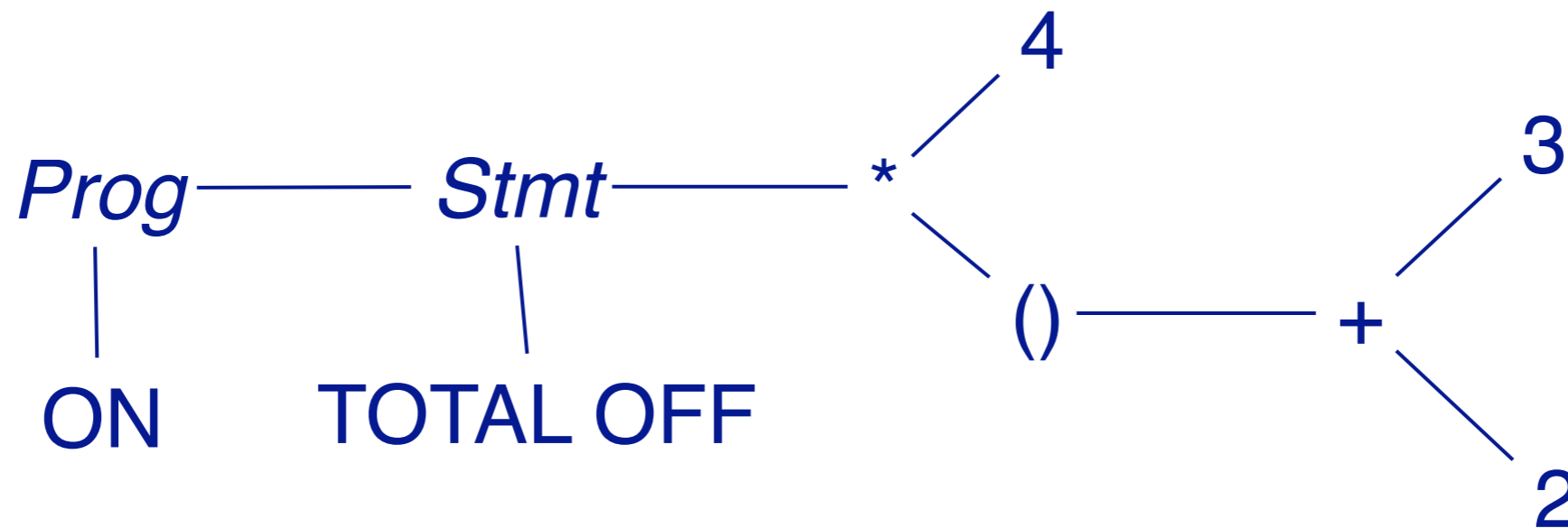
```
data Program      = On ExprSequence
data ExprSequence = Total Expression ExprSequence
                  | TotalOff Expression
data Expression   = Plus Expression Expression
                  | Times Expression Expression
                  | If Expression Expression Expression
                  | LastAnswer
                  | Braced Expression
                  | N Int
```

These data types simply express the content of an AST for the calculator language. The constructors are chosen so that they resemble somewhat the syntax of the source language.

A program consists of a statement (expression sequence). An expression can take one of six different forms, each possibly containing further subexpressions.

Representing Syntax

The test program “ON 4 * (3 + 2) TOTAL OFF” can be *parsed* as:



And *represented* as:

```
test = On (TotalOff (Times (N 4)
                          (Braced (Plus (N 3)
                                         (N 2))))))
```


Note that we are *not* implementing a parser for the calculator language. We assume that we have a parser that will parse the string:

```
"ON 4 * ( 3 + 2 ) TOTAL OFF"
```

and produce an AST represented in Haskell as:

```
On (TotalOff (Times (N 4) (Braced (Plus (N 3) (N 2))))))
```

Now, given this AST, we would like to interpret this program using our denotational semantics for the calculator language.

Implementing the Calculator

We can implement our denotational semantics directly in a functional language like Haskell:

```
pp :: Program -> [Int]
pp (On s)                = ss s 0

ss :: ExprSequence -> Int -> [Int]
ss (Total e s) n        = let n' = (ee e n) in n' : (ss s n')
ss (TotalOff e) n       = (ee e n) : [ ]

ee :: Expression -> Int -> Int
ee (Plus e1 e2) n       = (ee e1 n) + (ee e2 n)
ee (Times e1 e2) n      = (ee e1 n) * (ee e2 n)
ee (If e1 e2 e3) n      =
  | (ee e1 n) == 0       = (ee e2 n)
  | otherwise            = (ee e3 n)
ee (LastAnswer) n       = n
ee (Braced e) n         = (ee e n)
ee (N num) n            = num
```

Note how the semantic functions defined earlier can be directly implemented in Haskell, with little modification.

Note in particular that the arguments to the semantic functions are *not strings, but syntactic structures* (ASTs). This may not be obvious from the denotational semantics we saw earlier, but it is very clearly the case here.

This is true of all semantic specifications (including Featherweight Java): *semantic functions operate over syntactic structures, not strings*. We always assume that parsing is a separate issue.

See Calc.hs in the lectures-pl-examples git repo for the Haskell implementation.

$P [ON S] = S [S] (0)$
 $S [E TOTAL S] (n) = \text{let } n' = E [E] (n) \text{ in } \text{cons}(n', S [S] (n'))$
 $S [E TOTAL OFF] (n) = [E [E] (n)]$
 $E [E1 + E2] (n) = E [E1] (n) + E [E2] (n)$
 $E [E1 * E2] (n) = E [E1] (n) \times E [E2] (n)$
 $E [IF E1 , E2 , E3] (n) = \text{if } E [E1] (n) = 0$
 $\quad \text{then } E [E2] (n)$
 $\quad \text{else } E [E3] (n)$
 $E [LASTANSWER] (n) = n$
 $E [(E)] (n) = E [E] (n)$
 $E [N] (n) = N$

Here you can easily compare the two versions.

```

pp (On s)                = ss s 0

ss (Total e s) n         = let n' = (ee e n) in n' : (ss s n')
ss (TotalOff e) n       = (ee e n) : [ ]

ee (Plus e1 e2) n        = (ee e1 n) + (ee e2 n)
ee (Times e1 e2) n       = (ee e1 n) * (ee e2 n)
ee (If e1 e2 e3) n      =
  | (ee e1 n) == 0      = (ee e2 n)
  | otherwise           = (ee e3 n)

ee (LastAnswer) n       = n
ee (Braced e) n         = (ee e n)
ee (N num) n            = num

```

Roadmap



- > Syntax and Semantics
- > Semantics of Expressions
- > **Semantics of Assignment**
- > Other Issues

A Language with Assignment

```
Prog ::= Cmd '.'
Cmd  ::= Cmd1 ';' Cmd2
      | 'if' Bool 'then' Cmd1 'else' Cmd2
      | Id ' := ' Exp
Exp  ::= Exp1 '+' Exp2
      | Id
      | Num
Bool ::= Exp1 '=' Exp2
      | 'not' Bool
```

Example:

```
z := 1 ; if a = 0 then z := 3 else z := z + a .
```

Input number initializes a; output is final value of z.

This example is a simple imperative language with assignments to variables.

Variables are named **a** to **z**. The input to a program is stored as the initial value of **a**. The output is the value of **z**, i.e., the program should assign a value to **z** to produce an output.

There are four syntactic (and semantic) categories: programs, commands, (numeric) expressions and Booleans.

Representing abstract syntax trees

Data Structures:

```
data Program          =      Dot Command
data Command         =      CSeq Command Command
                       |      Assign Identifier Expression
                       |      If BooleanExpr Command Command
data Expression      =      Plus Expression Expression
                       |      Id Identifier
                       |      Num Int
data BooleanExpr     =      Equal Expression Expression
                       |      Not BooleanExpr
type Identifier      =      Char
```


As before, we define data structures to represent the syntactic elements of our language

An abstract syntax tree

Example:

```
z := 1 ; if a = 0 then z := 3 else z := z + a .
```

Is represented as:

```
Dot  (CSeq  (Assign 'z' (Num 1))
        (If (Equal (Id 'a') (Num 0))
            (Assign 'z' (Num 3))
            (Assign 'z' (Plus (Id 'z') (Id 'a'))))
      )
    )
```

Modelling Environments

A store is a mapping from identifiers to values:

```
type Store = Identifier -> Int
newstore :: Store
newstore id = 0

update :: Identifier -> Int -> Store -> Store
update id val store = store'
    where store' id'
        | id' == id = val
        | otherwise = store id'
```

Pay particular attention to the types of `newstore` and `update`:

A `newstore` maps every identifier to the initial value 0.

The `update` function takes an identifier, a new value and an old store, and produces a new store. The new store is just like the old store, except it maps the identifier to the new value.

Note how assignment and update are modeled in a pure language (Haskell) without assignment or update: state is represented as an environment (a store), and an update is modeled as the creation of *a new environment*.

Functional updates

Example:

```
env1 = update 'a' 1 (update 'b' 2 (newstore))  
env2 = update 'b' 3 env1
```

```
env1 'b'  
⇒ 2  
env2 'b'  
⇒ 3  
env2 'z'  
⇒ 0
```

Semantics of assignments in Haskell

```
pp :: Program -> Int -> Int
pp (Dot c) n = (cc c (update 'a' n newstore)) 'z'

cc :: Command -> Store -> Store
cc (CSeq c1 c2) s = cc c2 (cc c1 s)
cc (Assign id e) s = update id (ee e s) s
cc (If b c1 c2) s = ifelse (bb b s) (cc c1 s) (cc c2 s)

ee :: Expression -> Store -> Int
ee (Plus e1 e2) s = (ee e2 s) + (ee e1 s)
ee (Id id) s = s id
ee (Num n) s = n

bb :: BooleanExpr -> Store -> Bool
bb (Equal e1 e2) s = (ee e1 s) == (ee e2 s)
bb (Not b) s = not (bb b s)

ifelse :: Bool -> a -> a -> a
ifelse True x y = x
ifelse False x y = y
```

We model the semantics of our language in much the way as we did before — we have a separate semantic function for each of the four syntactic (and semantic categories). Look carefully at the type declarations for insight into what they do:

- A *program* takes an integer as input (**a**) and produces an integer output (**z**).
- A *command* updates the store, a mapping from identifiers to integers; that is, it takes a store and produces an updated store.
- An *expression* accesses the store and produces an integer value.
- A *Boolean* accesses the store and produces a Boolean value.

Finally `ifelse` is a helper function that takes a Boolean and selects between two alternative values.

Running the interpreter

```
src1 = "z := 1 ; if a = 0 then z := 3 else z := z + a ."  
ast1 = Dot (CSeq  
    (Assign 'z' (Num 1))  
    (If (Equal (Id 'a') (Num 0))  
        (Assign 'z' (Num 3))  
        (Assign 'z' (Plus (Id 'z') (Id 'a')))))
```

```
pp ast1 10
```

```
⇒ 11
```


Roadmap

- > Syntax and Semantics
- > Semantics of Expressions
- > Semantics of Assignment
- > **Other Issues**



Practical Issues

Modelling:

- > Errors and non-termination:
 - need a special “error” value in semantic domains
- > Branching:
 - semantic domains in which “continuations” model “the rest of the program” make it easy to transfer control
- > Interactive input
- > Dynamic typing
- > ...

Theoretical Issues

What are the denotations of lambda abstractions?

- > need Scott's theory of semantic domains






What are the semantics of recursive functions?

- > need least fixed point theory





How to model concurrency and non-determinism?

- > abandon standard semantic domains
- > use “interleaving semantics”
- > “true concurrency” requires other models ...

What you should know!

-  *What is the difference between syntax and semantics?*
-  *What is the difference between abstract and concrete syntax?*
-  *What is a semantic domain?*
-  *How can you specify semantics as mappings from syntax to behaviour?*
-  *How can assignments and updates be modelled with (pure) functions?*

Can you answer these questions?

-  *Why are semantic functions typically higher-order?*
-  *Does the calculator semantics specify strict or lazy evaluation?*
-  *Does the implementation of the calculator semantics use strict or lazy evaluation?*
-  *Why do commands and expressions have different semantic domains?*



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<http://creativecommons.org/licenses/by-sa/4.0/>