Semantics

Semantics refers to the behavior of a program, while ***syntax*** refers to its structure. At this time there is no generally agreed upon representation for semantics. Theories of program semantics are also hotly debated. (In my opinion this is because the semantics of imperative languages is tied to machine architecture, and thus does not even address the relevant issue of program meaning.)

Static Semantics

Static semantics refers to programming language characteristics which cannot be expressed in BNF form, but can be verified by a compiler. This does not refer to semantics at all, thus it is a misnomer. It does make obvious that even syntax is inadequately defined, and the entire enterprise of assuring the meaning or behavior of a program is suspect, at least for imperative languages.

Examples:

Type compatibility rules:

Consider adding an integer to a real:

```
3 + 4.1 = ?
```

The types of these objects do not match. In a strongly typed language, such an addition would be a typing error. More conveniently, languages have ***coercion rules*** which permit some types to be dynamically converted to other types. In the above example, `Int -> Real`.

Variable declaration:

How can you assure that all variables are declared before they are used? This structural requirement cannot be stated in BNF, so it requires a meta-language stronger than BNF to talk about how the program behaves.

Closing brackets:

In some block structured languages, the beginning and end of each block is labeled:

```
begin do <body> end do
```

There is no way to specify that the name of the `end` statement (`do` here) matches the name of the `begin` statement.

Attribute Grammars

Attribute grammars extend the expressability of context-free grammars like BNF, to include the checks for static semantics. An attribute grammar is a context-free parse tree with each token augmented with a set of attributes (such as type and initialization information).

Intrinsic attributes are those properties of leaf nodes (i.e. names) which are not contained in the parse tree itself. The type of a variable is an example. It is usually included in the symbol table, but may not show up as part of the parse tree. These are also called *synthesized attributes*, since they pass information up the parse tree starting at the variable names.

In contrast, *inherited attributes* are those that are passed down the parse tree, properties that depend on the operator structure of the parse tree.

Example: Simple Assignment Statements for adding two numbers

Here is the attribute grammar for typing for an assignment statement, `A := B + C`

Syntax BNF:

```
<assign> ::=  <var> := <expression>

<expression> ::=  <var> + <var>  |  <var>

<var> ::=  A  |  B  |  C
```

Parse tree:

```
              <assign>
            /    |    \
       <var>1    |     <expression>
         |       |       /    |    \
         |       |  <var>2    |   <var>3
         |       |     |      |     |
         A      :=     B      +     C
```

Static semantics:

We will assume that `{A,B,C}` are either integers or reals.

actual-type: a synthesized attribute which stores the actual type of a `<var>` or an `<expression>`. In the case of an `<expression>`, the type is computed given the types of the component `<var>`s.

expected-type: an inherited attribute which stores the expected type of `<expression>`. It is determined by the type of the `<var>` on the left-hand-side of the expression statement.

Attribute grammar:

Syntax:	`<assign> ::=  <var> := <expression>`
Semantics:	`<var>.actual-type implies <expression>.expected-type`
Syntax:	`<expression> ::=  <var>1 + <var>2`
Semantics:	`if (<var>1.actual-type = int) and (<var>2.actual-type = int) then int else real` `<expression>.actual-type = <expression>.expected-type`
Syntax:	`<expression> ::=  <var>2`
Semantics:	`<var>.actual-type implies <expression>.actual-type` `<expression>.actual-type = <expression>.expected-type`
Syntax:	`<var> ::=  A  |  B  |  C`
Semantics:	`lookup-type[<var>.string] implies <var>.actual-type`

One of the difficulties of attribute grammars is that specifying the semantic rules for an actual programming language is very difficult due to language size and complexity.

Types of Dynamic Semantics

The semantics of non-imperative languages is usually clear and straight-forward, because these languages were designed with their meaning in mind.

Functional languages:	substitution semantics defined by *lambda calculus*
Logical languages:	implication semantics defined by *predicate calculus*
Object-oriented languages:	mathematical semantics defined by *domain theories*

There have been several approaches to specifying semantics for ***imperative languages***, none are entirely satisfactory.

Dynamic Semantics for Imperative Languages

A clear definition of semantics is necessary for

1. knowing how the language actually works
2. compiler design
3. proof of correctness

For program understanding, the semantic specification must also be small and intelligible.

Operational Semantics

Operational semantics defines the meaning of a program by executing its statements, either in hardware, or simulated in software. Meaning is the changes which occur in a machine's state, or memory.

Naturally the specifics of a machine architecture, its hardware implementation, and its operating system interact strongly with operational semantics. This makes operational semantics difficult to understand and very machine specific (i.e. non-portable).

The concept of a ***virtual machine*** was introduced to buffer operational semantics from machine specific details. The changes of state of the virtual machine (i.e. the software simulation) define program meaning; that is, a program is defined in terms of another program. For this idea to work, we must first translate a program into appropriate low-level statements in the assembly language of the virtual machine (i.e. statements about virtual registers, memory, and data movement). Then we must cast the changes in machine state in terms of changes that align with the programmer's intentions.

Since operational semantics, even on a virtual machine, is defined in terms of algorithms rather than mathematics, knowledge of machine changes does not help with program understanding or verification. As yet, there is no principled theory of algorithms.

Denotational Semantics

Denotational semantics is a rigorous attempt to define program behavior in terms of ***recursive function theory***. Each language object is defined both as a mathematical object and as a function which maps instances of the language object (as they occur in a program) onto the appropriate mathematical object. The problem with this approach is that it is not at all clear what the appropriate mathematical objects are for programming constructs.

Example: ***Binary Numbers***

Domain: N the set of all non-negative integers

We will associate each decimal number with some non-negative integer. The functional mapping M follows :

```
M['0'] = 0

M['1'] = 1

M[<binary-number> '0'] = 2 * M[<binary-number>]

M[<binary-number> '1'] = 2 * M[<binary-number>] + 1
```

Note that this is implemented by a recursive function:

```
M[bin] =def=
  if bin='0' then 0
    elseif bin='1' then 1
      elseif last[bin]='0' then 2* M[but-last[bin]]
        elseif last[bin]='1' then 2* M[but-last[bin]] +1
          else ERROR
```

The functions `last` and `butlast` are accessors for the binary number, decomposing by separating the last digit from all digits butlast. If we reversed the binary number before decomposition, then the accessors would be `first` and `rest`. That is:

```
last[bin] =def= first[reverse[bin]]

butlast[bin] =def= reverse[rest[reverse[bin]]]
```

Since recursive functions are a model for computation, there is a close relationship between operational semantics and denotational semantics: both require a virtual machine to identify the state changes which define the meaning of a computation. Denotational semantics is an improvement, since it relies on mathematical functions for definition rather than on algorithms.

Example: *Assignment Statement*

Let the environment `E` (i.e. the state of the computation) be represented by pairs `(name1, value1)` of variable names and their current values. The mathematical function M defines the meaning of the assignment `x := <expression>`

```
M[x := expression, E] =
  if M[expression, E] = ERROR then ERROR         ;expression is not valid
    else E' = {...(namei', valuei')...}          ;new environment
      where for j=1..n
        if namej=x then                          ;compare names
          M[expression,E]
          else valuej' = get-value-from-environment[namej, E]
```

Axiomatic Semantics

This method evolved out of proving program correctness. The idea is that each program statement is surrounded by *constraints* (preconditions and postconditions) which specify the behavior of program variables. The language of these constraints is Predicate calculus. The constraints are called ***assertions***. Assertions are specified in a program by enclosing them in curly brackets.

Example:

```
{x isa integer, x>0}
  sum := 2*x + 1
{sum > 2}
```

The precondition specifies constraints on the input `x`. The postcondition specifies constraints on the output `sum`.

In designing preconditions, the ***weakest precondition*** is the least restrictive assertion which still guarantees the validity of the postcondition. If you can compute the weakest precondition given the postcondition, then a correctness proof for the statement can be developed. Continuing the example:

Given `((sum > 2) and (sum = 2x+1)),`

```
2 < 2x+1
1 < 2x
x >= 1/2
```

We derive that `x` is greater than `1/2`. The assertion that `x` is an integer cannot be proved, but it is know from the static semantics analysis. Thus

```
x>0   is true
```

It is also true that `x>5` (or any other integer) when `sum>10`. Since there are cases where this precondition is not true (i.e. when `sum<11`), `x>5` is not the weakest precondition.

The usual abstract syntax for assertions is:

```
{P} S {Q}
```

where `P` is the precondition, `Q` is the postcondition, and `S` is the statement.

Axioms cannot cover sequences of statements, since that would require a separate axiom for each different sequence of program statement types. Rather sequences are analyzed using rules of inference of the form

Preconditions imply `S1`; `S1` implies `S2`; ...; `Sn` implies postconditions

Thus to use axiomatics, the entire mechanism of proofs in predicate calculus is needed. This is both hard to understand and difficult to use. Like denotational semantics, although axiomatic semantics is a tractable idea, it becomes very complex for normal programming languages, and is thus of very limited utility.