Informal Semantics

- Grammars can be used to define the legal programs of a language, but not what they do! (Actually, most languages place further, non-grammatical restrictions on legal programs, e.g., type-correctness.)
- Language behavior is usually described, documented, and implemented on the basis of natural-language (e.g., English) descriptions.
- Descriptions are usually structured around the language’s grammar, e.g., they describe what each nonterminal does.
- Natural-language descriptions tend to be imprecise, incomplete, and inconsistent.

Example: FORTRAN DO-loops.

“DO n i = m1, m2, m3
Repeat execution through statement n, beginning with \(i = m_1\), incrementing by \(m_3\), while \(i\) is less than or equal to \(m_2\). If \(m_3\) is omitted, it is assumed to be 1. \(m\)’s and \(i\)’s cannot be subscripted. \(m\)’s can be either integer numbers or integer variables; \(i\) is an integer variable.”


Consider:

```
DO 100 I = 10, 9, 1
...
100 CONTINUE
```

How many times is the body executed?

Experimental Semantics

Try it and see!

Implementation becomes standard of correctness.

This is certainly precise: compiler source code becomes specification.

But it is:

- difficult to understand;
- awkward to use;
- subject to accidental change;
- wholly non-portable.
Formal Semantics

Aims:
- **Rigorous** and unambiguous definition in terms of a well-understood formalism, e.g., logic, naive set theory, etc.
- Independence from implementation. Definition should describe how the language behaves as abstractly as possible.

Uses:
- Provably-correct implementations.
- Provably-correct programs.
- Basis for language comparison.
- Basis for language design.

(But usually not basis for learning a language.)

Main varieties:
- Operational
- Denotational
- Axiomatic

Each has different purposes and strengths. In this course, we’ll mostly focus on operational semantics, with brief looks at the others.

Operational Semantics

Define behavior of language on an abstract machine.

Abstract machine should be much simpler than real machines, since otherwise a compiler for a real machine would be just as good!

Typical mechanisms:
- Characterize the state of the abstract machine (typically as an environment mapping variables to values) and give a set of evaluation rules describing how each syntactic construct affects the state.
- Define a simple Von Neumann-style stack machine and describe how each syntactic construct can be compiled into stack machine instructions.

Some useful things to do with an operational semantics:
- Build an implementation for a real machine by interpreting or compiling the abstract machine code.
- Explicate the meaning of a language feature by proving that it has the same behavior as a combination of simpler features.
- Prove that correctly typed programs cannot “dump core” at runtime.

Semantics from Interpreters

In the homework, we’ll be building **definitional interpreters** for small languages that display key programming language constructs.

Our goal is to study the interpreter code in order to understand implementation issues associated with each language.

In addition, the interpreter serves as a form of semantic definition for each language construct. In effect, it defines the meaning of the language in terms of the semantics of Java or ML.

(Of course, you’ll also be learning more about the semantics of Java and ML as we go!)

Semantics and Erroneous Programs

An important part of a language specification is distinguishing valid from invalid programs.

It is useful to define four classes of errors that make programs “bad.”

**Static errors** are violations of the language specification that can be detected at compilation time (or, in an interpreter, before interpretation begins)
- Includes: lexical errors, syntactic errors (caught during parsing), type errors, etc.
- Compiler or interpreter issues an error pinpointing erroneous location in source program.
- Language semantics are usually defined only for programs that have no static errors.

**Checked runtime errors** are violations that the language implementation is required to detect and report at runtime, in a clean way.
- Examples in Java or ML: division by zero, array bounds violations, dereferencing a null pointer.
- Depending on language, implementation may issue an error message and die, or raise an exception (which can be caught by the program).
- Language semantics must specify behavior precisely.
Erroneous Programs (cont.)

Unchecked runtime errors are violations that the implementation need not detect.

- Subsequent behavior of the computation is arbitrary. (Error is often not manifested until much later in execution.)
- Examples in C: division by zero, dereferencing a null pointer, array bounds violations.
- Language semantics probably don’t specify behavior.
- Java and ML have no such errors!

Logic errors are mistakes in the program that don’t violate language rules but cause program to behave incorrectly.

Stateful Programming

Imperative programming is a good match to underlying Von Neumann machine programs, which are sequences of instructions that modify the contents of registers and memory locations.

- User-program variables are mapped to machine locations.
- User-program operations correspond to primitive machine instructions.

Imperative languages are also suitable for writing reactive programs that interact with the state of the “real world.” Examples:

- Reading mouse clicks and modifying the contents of a display.
- Controlling a set of relays in an external device.

Imperative programming is the dominant paradigm, but there are alternative “declarative” paradigms too...

Imperative Languages

Most commonly-used programming languages are imperative: they consist of a sequence of operations that alter the state of the program.

- The state includes the values of variables, which can change by means of assignment operations.
- The state also includes the input/output history of the program, e.g., the contents of files (or virtual files) read or written by the program’s I/O operations.

Many languages put have a separate syntactic category of statements (or commands) that includes stateful operations which don’t produce a result value. But expressions can also affect the state (in which case they are said to have side-effects) in addition to returning a result.

Most languages also support user-defined procedures (which are called in a statement context, and don’t return a value), and/or functions (called in an expression context, and do return a value). Usually, both of these can also have side-effects.

- Order of evaluation often matters, and is not always obvious when expressions are involved.

Assignment

The basic primitive stateful operation is typically assignment, which alters a value stored in a location.

Depending on language, assignments are statements (with no result value), or expressions (maybe with result value).

In the simplest form, the location is associated with a simple variable, e.g.,

\[ a := a + 2 \]

(Will use := for assignment, = for equality relational operator. C/C++/Java use =, == respectively: a bad idea, because both form expressions.)

In most languages, the variable name \( a \) means different things on the left-hand and right-hand sides.

On the LHS, \( a \) denotes the location of the variable \( a \), into which the value of the RHS expression is to be stored.

On the RHS, \( a \) denotes the value currently contained in \( a \), i.e., it indicates an implicit dereference operation.
ML References

In ML, ordinary “variables” are immutable, i.e., they are really just names for values (computed at runtime), rather than for locations. Updatable variables, called references, must be explicitly created as such, and always serve as l-values. The contents of the variable must be explicitly dereferenced:

```ml
let val x = ref 2
in x := !x + 2
end
```

```ml
let val y = ref 0
  fun setto10 (x: int ref) = x := 10
in setto10 y
end
```

This is somewhat more verbose, but removes any confusion between l-value and r-value.

Initialization Values

Most languages require variables (and other sources of l-values) to be declared before they are used: gives them a type and scope, and optionally, an initializing expression.

In fact, it is surely a bug to use any variable as an r-value unless it has previously assigned a value. But many languages permit this, resulting in runtime errors.

The simplest fix is to require an initial value to be given for every declared variable. ML requires this for mutable ref variables (and also of course for ordinary immutable variables).

Java takes a slightly more sophisticated approach:

- variables do not need to be initialized at the point of declaration; but
- they must be initialized before they are actually used.

But in any reasonably powerful language, checking initialization before use is an uncomputable problem.

Definite Assignment

So the Java language reference manual carefully details a conservative, computable, set of conditions, which every program must meet, that guarantee there will be no uses before definition.

This is called the definite assignment property; just defining it takes 16 pages of the reference manual.

Some programs that do in fact initialize before use will be rejected because they violate the conditions.

Legal example:

```ml
int a;
if (b) // b is some boolean variable
  a = 3;
else
  a = 4;
a = a + 1;
```

Illegal example:

```ml
int a;
if (b)
  a = 3;
if (!b)
  a = 4;
a = a + 1;
```

Order of Evaluation

Order of stateful operations affects program semantics.

Statements are always explicitly ordered, making these differences obvious.

Expressions can also have side-effects, but order of evaluation is often under-specified (precedence and associativity don’t always fix order).

ANSI C example:

```c
a = 0;
b = (a = a + 1) - (a = a + 2);
```

Result (1-3 = -2 or 3-2 = 1 ?) depends on compiler whim.

Side-effects are not always obvious:

```ml
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
int h(a,f(2)); // = 0 or 2 ??
```

Keeping expression evaluation order or argument evaluation order undefined sometimes lets compiler generate more efficient code.

But modern languages (e.g., Java, ML) have moved towards precise definition of evaluation order within expressions (e.g., left-to-right).
Structured Control Flow

All modern higher-level imperative languages are designed to support structured programming. Loosely, a structured program is one in which the syntactic structure of the program text corresponds to the flow of control through the dynamically executing program.

Originally proposed (most famously by Dijkstra) as an improvement on the incomprehensible “spaghetti code” that is easy to produce using the labels and jumps supported directly by hardware.

More specifically, structured programs use a very small collection of (recursively defined) compound statements to describe their control flow.

Compounds are of three kinds:

- Sequential composition: form a statement from a sequence of statements, e.g.
  
  (Java) \{ x = 2; y = x + 4; \}
  
  (Pascal) begin x := 2; y := x + 4; end

- Selection: execute one of several statements, e.g.,
  
  (Java) if (x < 0) y = x + 1; else z = y + 2;

- Iteration: repeatedly execute a statement, e.g.,
  
  (Java) while (x > 10) output(x--);
  
  (Pascal) for x := 1 to 12 do output(x*2);

Selection

The basic selection statement is based on boolean values

\[ \text{if } e \text{ then } s_1 \text{ else } s_2 \]

which translates to

\[
\text{evaluate } e \text{ into } t \\
\text{cmp } t, \text{true} \\
\text{brneq } l_1 \\
\text{s}_1 \\
\text{br } l_2 \\
\text{l}_1: \text{s}_2 \\
\text{l}_2: \\
\]

To test types with more than two values, multi-way selections against constants are appropriate:

\[
\text{case } e \text{ of} \\
\text{c}_1 : s_1 \\
\text{c}_2 : s_2 \\
\ldots \\
\text{c}_n : s_n \\
\text{default : } s_d \\
\]

The most efficient translation of case statements depends on density of the value \(c_1, c_2, \ldots, c_n\) within the range of possible values for \(e\).

Sparse Cases

For sparse distributions, it’s best to translate the case just as if it were:

\[
t := e; \\
\text{if } t = c_1 \text{ then } s_1 \\
\text{else if } t = c_2 \text{ then } s_2 \\
\text{else} \\
\ldots \\
\text{else if } t = c_n \text{ then } s_n \\
\text{else } s_d
\]

Dense Cases

For a dense set of labels in the range \([c_1, c_n]\), it’s better to use a jump table:

\[
\text{evaluate } e \text{ into } t \\
\text{cmp } t, c_1 \\
\text{brlt } l_d \\
\text{cmp } t, c_n \\
\text{brgt } l_d \\
\text{sub } t, c_1, t \\
\text{add table}, t, t \\
\text{br } \ast t \\
\text{table}: l_1 \\
\ldots \\
\text{l}_n \\
\text{l}_1: \text{s}_1 \\
\text{l}_2: \text{s}_2 \\
\text{br done} \\
\ldots \\
\text{l}_n: \text{s}_n \\
\text{br done} \\
\text{l}_d: \text{s}_d \\
\text{done:}
\]

The best approach for a given case may involve a combination of these two techniques. Compilers differ widely in the quality of the code generated for case.
**Iteration**

The basic loop construct is

```
while e do s
```

corresponding to:

```
top: evaluate e into t
cmp t, true
brneq done
s
br top
done:
```

A commonly-supported variant is to move the test to the bottom:

```
repeat s until e
```

which is equivalent to:

```
s;
while not e do s
```

---

**Loop exits**

It is sometimes desirable to exit from the middle of a loop:

```
loop
s1;
exitif e;
s2
end
```

is equivalent to:

```
top: s1
   evaluate e into t
cmp t, true
brneq done
s2
br top
done:
```

C/C++/Java have an unconditional form of `exit`, called `break`. They also have a `continue` statement that jumps back to the top of the loop.

---

**Uses for goto?**

An efficient program with `goto`:

```
int i;
for (i = 0; i < n; i++)
   if (a[i] == k)
      goto found;
n++; a[i] = k;
b[i] = 0;
found:
b[i]++;
```

In most languages (e.g., Modula, C/C++) there is no equivalently efficient solution without `goto`.

But we can do as well in Java, using a named, multi-level `break`:

```
int i;
search:
   { for (i = 0; i < n; i++)
      if (a[i] == k)
         break search;
      n++; a[i] = k;
b[i] = 0;
   }
b[i]++;
```

---

**Counted loops**

Since iterating a definite number of times is very common, languages often offer a dedicated statement, with basic form:

```
for i := e1 to e2 do s
```

Here `s` is executed repeatedly with `i` taking on the values `e1, e1 + 1, ..., e2` in each successive iteration.

The detailed semantics of this statement vary, and can be tricky. Often, `s` is prohibited from modifying `i`, which (under certain other conditions) guarantees that the loop will be executed exactly `e2 - e1 + 1` times.

C/C++/Java have a much more general version of `for`, which guarantees much less about the behavior of the loop:

```
for (e1; e2; e3) s;
```

is exactly equivalent to:

```
e1;
while (e2) {
   s;
e3
}
```
The COME FROM statement

```
10 J = 1
11 COME FROM 20
12 PRINT J
  STOP
13 COME FROM 10
20 J = J + 2
```


But is this really a joke?

Even with a GO TO, we must examine both the branch and the target label to understand the programmer’s intent.

Triples involving Assertions

We write a triple

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

to mean that if the precondition \( P \) is true before the execution of \( S \) then the postcondition \( Q \) will be true after the execution of \( S \).

Note that the triple says nothing about what happens if \( S \) doesn’t terminate. So we are only characterizing statements that terminate.

Examples of triples (not all stating true things!)

```
{ y ≥ 3 } x := y + 1 { x ≥ 4 }

{x + y = c} while x > 0 do
  y := y + 1;
  x := x - 1
end { x + y = c }

{ y = 2 } x := y + 1 { x = 4 }

{ y = 2 } x := y + z { x = 4 }
```

How do we distinguish true triples from false?

Who’s to say which ones are true?

It all depends on the semantics of statements!

Axiomatic Semantics

So far, we’ve given an (informal) operational semantics for imperative statements:

- we translate them into instructions for a simple abstract machine instructions.
- we rely on our intuitions about how this machine works.

(Homework will also illustrate an another, higher-level operational semantics.)

An important alternative approach is to give a logical interpretation to statements.

- The state of an imperative program is defined by the values of the all its variables.
- We can characterize a state by giving a logical predicate (or assertion), mentioning the variables, which is satisfied by the values of the variables in that state.
- We can define the semantics of statements by saying how they affect (arbitrary) predicates.

Axioms and Rules of Inference

If we work in a suitably structured language, we can give a fixed set of axioms and rules of inference, one for each kind of statement. We then take as true the set of triples that can be logically deduced from these axioms and rules.

Of course, we want to choose axioms and rules that capture our intuitive understanding of what the statements do, and they need to be as strong as possible.

**Assignment Axiom**

```
{ P[E/x] } x := E { P }
```

where \( P[E/x] \) means \( P \) with all instances of \( x \) replaced by \( E \).

This axiom may seem backwards at first, but it makes sense if we start from the postcondition. For example, if we want to show \( x ≥ 4 \) after the execution of

```
x := y + 1
```

then the necessary precondition is \( y + 1 ≥ 4 \), i.e., \( y ≥ 3 \).

**Conditional Rule**

```
{ P ∧ E } S₁ { Q }, { P ∧ ¬ E } S₂ { Q }
------------------------------
{ P } if E then S₁ else S₂ endif { Q }
```
More Proof Rules

Composition Rule

{ \( P \) } S_1 { \{ Q \} } , \{ Q \} S_2 \{ R \} 
\-------------------------------
\{ \( P \) \} S_1 ; S_2 \{ R \} 

While Rule

( \( P \land E \) ) S \{ \( P \) \}  
\-------------------------------
\{ \( P \) \} while E do S \{ \( P \land \neg E \) \}  

Consequence Rule

\( P \Rightarrow P' \), \{ \( P' \) \} S \{ \( Q' \) \}, \( Q' \Rightarrow Q \)  
\-------------------------------
\{ \( P \) \} S \{ \( Q \) \} 

Here \( P \Rightarrow Q \) means that “\( P \) implies \( Q \),” i.e., “\( Q \) is true whenever \( P \) is true,” i.e. “\( P \) is false or \( Q \) is true.” Hence we always have \( False \Rightarrow Q \) for any \( Q \)!

Annotated Program Example

Proof trees can be unwieldy. Because the structure of the tree corresponds directly to the structure of the program code, it is common to use an alternative representation of proofs in which we annotate programs with assertions.

\{ \( x + y = c \) \} 
while \( x > 0 \) do  
{ \( x + y = c \land x > 0 \) } 
{ \( x + y = c \) } 
y := y + 1;  
{ \( x + y = c + 1 \) } 
x := x - 1  
{ \( x + y = c \) } 
end  
{ \( x + y = c \land x \leq 0 \) } 
{ \( x + y = c \) } 

To verify that this is a valid proof, we have to check that the annotations are consistent with each other and with the rules and axioms.

Proof Tree Example

---(ASSIGN)---  
\{ \( x + y = c \) \} 
y := y + 1  
\{ \( x + y = c + 1 \) \}  
---(CONSEQ)---  
\{ \( x + y = c \land x > 0 \) \} 
y := y + 1  
\{ \( x + y = c + 1 \) \} 
x := x - 1  
\{ \( x + y = c \) \}  
---(COMP)---  
\{ \( x + y = c \land x > 0 \) \} 
y := y + 1; x := x - 1  
\{ \( x + y = c \) \}  
---(WHILE)---  
while \( x > 0 \) do y := y + 1; x := x - 1 end  
\{ \( x + y = c \land x \leq 0 \) \}  
---(CONSEQ)---  
\{ \( x + y = c \) \} 
while \( x > 0 \) do y := y + 1; x := x - 1 end  
\{ \( x + y = c \) \} 

---(CONSEQ)---  
\{ \( x + y = c \) \} 
while \( x > 0 \) do y := y + 1; x := x - 1 end  
\{ \( x + y = c \) \}  

Merits and Problems of Axiomatic Semantics

Gives a very clean semantics for structured statements.

But things get more complicated if we add features like:

- expressions with side-effects
- statements that break out of loops
- procedures
- non-trivial data structures and aliases

Useful for formal proofs of program properties (though these are seldom done).

Thinking in terms of assertions is good for informal reasoning about programs. (And there are beginning to be useful automated theorem proving support tools too.)

Other forms of semantic definition, e.g., natural semantics, also use similar logical structures.