CS558 Programming Languages
Winter 2006
Lecture 2a
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 = m_1, m_2, m_3

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

It is useful to define three classes of errors that make programs invalid. (Of course, even valid programs may behave differently than the programmer intended!)

**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.

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!
Most commonly-used programming languages are **imperative**: they consist of a sequence of actions that alter the **state** of the world.

State includes the values of program variables and also the program’s external environment (e.g. files the program reads or writes).

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...
Many languages put have a separate syntactic category of statements (or commands) that includes stateful operations which don’t produce a result value.

But in some languages, certain expressions can also affect the state (in which case they are said to have side-effects) in addition to returning a result.

Also, most languages support user-defined functions, which contain statements but return a value and are invoked in an expression context; this is another way expressions can have side-effects.
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.
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

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

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

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

Legal example:

\begin{verbatim}
    int a;
    if (b)  /* b is boolean */
        a = 3;
    else
        a = 4;
    a = a + 1;
\end{verbatim}

Illegal example:

\begin{verbatim}
    int a;
    if (b)
        a = 3;
    if (!b)
        a = 4;
    a = a + 1;
\end{verbatim}
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:

```java
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
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).
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.
Kinds of Compound Statements

- 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);
The basic selection statement is based on boolean values

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

which translates to

\[
\text{evaluate } e \text{ into } t \\
\text{cmp } t, \text{true} \\
\text{brneq } l_1 \\
s_1 \\
br \ l_2 \\
l_1: \ s_2 \\
l_2:
\]
To test types with more than two values, multi-way selections against constants are appropriate:

```plaintext
case e of
c1 : s1
c2 : s2
...
cn : sn
default : sd
```

The most efficient translation of `case` statements depends on **density** of the value `c1, c2, ..., cn` within the range of possible values for `e`. 
For sparse distributions, it’s best to translate the case just as if it were:

\[
\begin{align*}
  t &:= e; \\
  \text{if } t &= c_1 \text{ then} \\
  &\quad s_1 \\
  \text{else if } t &= c_2 \text{ then} \\
  &\quad s_2 \\
  \text{else} \\
  &\quad \ldots \\
  \text{else if } t &= c_n \text{ then} \\
  &\quad s_n \\
  \text{else} \\
  &\quad s_d
\end{align*}
\]
For a **dense** set of labels in the range \([c_1, c_n]\), it’s better to use a **jump table**:

\[
\begin{align*}
e & \text{ evaluate } e \text{ into } t \\
cmp & \quad l_1: \quad s_1 \\
brlt & \quad l_2: \quad s_2 \\
cmp & \quad \text{br done} \\
brgt & \quad \text{br done} \\
sub & \quad \ldots \\
add \ 	ext{table}, t, t & \quad l_n: \quad s_n \\
br \ *t & \quad \text{br done} \\
table: & \quad l_d: \quad s_d \\
\end{align*}
\]

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**.
The basic loop construct is

\[
\text{while } e \text{ do } s
\]

corresponding to:

\[
\text{top: evaluate } e \text{ into } t \\
\text{cmp } t, \text{true} \\
\text{brneq done} \\
\text{s} \\
\text{br top} \\
\text{done:}
\]

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

\[
\text{repeat } s \text{ until } e
\]

which is equivalent to:

\[
s; \\
\text{while not } e \text{ do } s
\]
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
      breq 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.
An efficient program with goto:

```c
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`:

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

(This construct was invented by Knuth in the 1960’s, but not adopted into a mainstream language for about 30 years!)
Since iterating a definite number of times is very common, languages often offer a dedicated statement, with basic form:

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

Here $s$ is executed repeatedly with $i$ taking on the values $e_1, e_1 + 1, \ldots, e_2$ 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 $e_2 - e_1 + 1$ times.

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

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

is exactly equivalent to:

```plaintext
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.