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?
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 a short look at axiomatic semantics.
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.
Syntax-directed Language Processing

Use grammatical structure of language to guide checking, interpretation, translation into lower-level form, etc.

One way to use this structure: traverse parse tree or abstract syntax tree, evaluating semantic rules.

Semantic rules (“attribute equations”):

- Assign values to attributes attached to nodes of tree.

Examples: type or value of expression; code for statement block.

- Perform side-effects on global state (DEPRECATED!).

Examples: make entries in symbol table; issue errors; generate code to output file.

Attributes are pieces of information (any kind!) attached to nodes of a grammar-induced tree.

Semantic rules are associated with grammar productions, because each tree node is “built” by a production. (Terminal nodes are assumed to have their attributes “at the beginning.”)

Collectively, semantic rules make up an attribute grammar.
Attribute Evaluation

Attribute grammars can be used with parse tree or abstract syntax tree; we will concentrate on ASTs.

**Evaluation** order of semantic rules may or may not follow reduction order during parser: depends on form of rules.

Computing attribute values is called **annotating** or **decorating** the tree.

If used with parse tree, often try to compute attribute values **while parsing**.

Sometimes, attributes are more important than parse tree itself, so we may try to calculate the attributes while parsing, **instead** of building a parse tree.

Example: can use attribute grammar on a **parse** tree to compute the **AST** as an attribute!

More complicated attribute equations may require whole tree to exist first, before attribute evaluation begins.

An attribute is:

- **synthesized** if its value at a node depends only on values of attributes of **descendants** of that node; or

- **inherited** if its value at a node depends only on the values of attributes of **ancestors** and/or **siblings** of that node.
Synthesized attributes on AST’s

Attribute values at non-terminal node depend only on values at node’s children. Values at terminal nodes are provided by lexical analyzer.

Example: interpreting “calculator” expression language

\[
\begin{align*}
Add &: \quad E \rightarrow E_1 \ E_2 \quad & E.\text{val} &= \ E_1.\text{val} + \ E_2.\text{val} \\
Mul &: \quad E \rightarrow E_1 \ E_2 \quad & E.\text{val} &= \ E_1.\text{val} \ast \ E_2.\text{val} \\
Exp &: \quad E \rightarrow E_1 \ E_2 \quad & E.\text{val} &= \ \text{exp}(E_2.\text{val} \ast \ \text{ln}(E_1.\text{val}))
\end{align*}
\]

... \\
Num (val) &: \quad E \rightarrow \ \epsilon

Here Num:E.val is a terminal attribute assumed to be defined before attribute evaluation begins.

“Decorated” AST for \((3 \ast 5) + 4\):

```
Add:E.val = 19
    \
|       |       |
Mul:E.val = 15    Num:E.val=4
    |       |       |
Num:E.val=3    Num:E.val=5
```

Note: Same parse tree and attribute evaluation pattern would hold for static attributes, such as expression type, code sequence, etc.
Inherited Attributes

Sometimes convenient to make node’s attributes dependent on siblings or ancestors in tree.

Useful for expressing dependence on context, e.g., relating identifier uses to declarations. (Especially important because CFG cannot capture such dependencies.)

Example: Consider evaluating expressions in a language with local let-bindings as in ML, e.g., “let id = E₁ in E₂ end”.

Use an attribute grammar that maintains an inherited attribute containing an environment.

```
Let (id) : E → E₁ E₂
    E₁.env := E.env;
    E₂.env := extend(E.env, E.id, E₁.val);
    E.val := E₂.val

Add : E → E₁ E₂
    E₁.env := E.env; E₂.env := E.env;
    E.val := E₁.val + E₂.val

...  
Id (id) : E → ε    E.val := lookup(E.env, E.id)
```

Here Id:E.id and Let:E.id are terminal attributes assumed to be defined before attribute evaluation begins.

Inherited attributes need to be given an initial value at the root node of the AST. In this example, an empty environment is suitable.
Dependency Graphs

Example of decorated AST for `let x = 3*5 in x + 4`:

```
+---+     +---+     +---+     +---+     +---+     +---+
| Let: E.val = 19; .env = {} |
+---+     +---+     +---+     +---+     +---+     +---+
| Mul: E.val = 15; .env = {} |
+---+     +---+     +---+     +---+     +---+     +---+
| Add: E.val = 19; .env = {x=15} |
+---+     +---+     +---+     +---+     +---+     +---+
| Num: E.val=3; env={} |
+---+     +---+     +---+     +---+     +---+     +---+
| Num: E.val=5; env={} |
+---+     +---+     +---+     +---+     +---+     +---+
| Id: E.val=15; .env={x=15} |
+---+     +---+     +---+     +---+     +---+     +---+
| Num: E.val=4; env={x=15} |
```

Arrows show essential dependency relation among attributes. Taken together, arrow describe dependency graph.

- Solid lines represent synthesized attribute dependencies.
- Dotted lines represent inherited attribute dependencies.

Must evaluate attributes in topological order of dependency graph.
Evaluating Attribute Grammars

Attributes on ASTs can be computed by a recursive traversal of the tree structure.

- Define one evaluation function for each possible node label (similar to recursive descent parsing).
- Inherited attributes are passed as function arguments.
- Synthesized attributes are returned as function results.
- Each function invokes the evaluation functions for its children (after preparing their inherited attributes) and processes the results of these calls to produce and return its own synthesized attributes.
- Works unless there are cycles in the attribute dependency graph.

Implementing the evaluation function:

- In ML (or conventional imperative language), each node label is a variant record; write a single evaluation function and use case dispatch over the variant label.
- In Java (or other OO language), each node label is a separate subclass; overload the evaluation function for each subclass.
Why attribute grammars?

- **Compact**, convenient formalism.
- **Local** rules describe entire computation.
- Separate **traversal** from **computation**.
- (Purely **functional** rules can be evaluated in any order.)
Names and Binding

One essential part of being a “high-level” language is having convenient names for things: operators, variables, constants, types, procedures, classes, etc.

A binding is an association between a name and the thing it names.

Bindings can occur at many different times in the life-cycle of a language/program.

Key characteristics of any binding:

- **scope**: in what textual region of the program is the binding active?

- **lifetime**: when is the binding created and destroyed?

Scope and lifetime can each have “holes” in which binding is inaccessible, possibly because of hiding by another binding with the same name.
Examples (in C)

```c
static int x = 101;
bar (double x) {
    x += 1.0;
}
main () {
    int y;
    bar (3.14);
    { char *y; /* inner block */
        y = 'a';
    }
    y = y + x;
}
```

- **int x**
  Lifetime: whole execution; hole while `bar` executes.
  Scope: whole program except body of `bar`.

- **double x**
  Lifetime: between entry and exit of `bar`.
  Scope: body of `bar`.

- **int y**
  Lifetime: between entry and exit of `main`; holes while `bar` and inner block execute.
  Scope: body of `main`, except for inner block.

- **bar and main**
  Lifetime: whole execution.
  Scope: From point of definition to end of program.
Binding Lifetime vs. Object Lifetime

Many bindings are to memory-allocated “objects” (variables, records, subroutine activation records, etc.).

The lifetime of a memory object is the period from when it is allocated to when it is deallocated for possible re-use. Object lifetimes don’t have holes.

The lifetime of a binding to an object is not necessarily the same as the object’s lifetime.

- Objects often remain alive even when there is (temporarily) no live binding to them.

Example: The memory allocated for x remains alive from entry to exit of foo, but there is a hole in the lifetime of the binding of x to that memory during the execution of bar.

```c
foo () {
    int x[100];
    bar ();
    x[0] = x[99] + 1;
}
```
Object Lifetime (cont.)

- Objects sometimes die even when there is still a live binding for them. This is usually a bug (a dangling pointer) and many languages try to prevent it:

Example: The binding of p to the allocated storage remains alive to the exit of foo, even though that storage may now be reallocated for something else!

```c
char *foo() {
    char *p = malloc(100);
    free(p);
    return p;
}
```
Storage Classes for Objects

Static Data: Permanent Lifetimes

- Global variables and constants.
- Allows fixed address to be compiled into code.
- No runtime management costs.
- Original FORTRAN (no recursion) used static activation records.

Stack Data: Nested Lifetimes

- Allocation/deallocation is cheap (just adjust stack pointer).
- Most architectures support cheap sp-based addressing.
- Good locality for VM systems, caches.
- C, Algol/Pascal family, Java use stack for activation records.

Heap Data: Arbitrary Lifetimes

- Requires explicit allocation and (dangerous) explicit deallocation or garbage collection.
- Lisp, ML, many interpreted languages need heap for activation records, which have non-nested lifetimes.
Storage Organization

- Subdivide machine address space by function, access, allocation.

- Typical organization (Unix)

```
∞ -------------------------------
| | STACK
| ↓
```

```
| “top” of stack               (but stacks nearly always grow DOWN!)
| ↓
```

```
| (in VM systems, these pages don’t actually exist)
```

```
| (controlled by sbrk system call)
```

```
| HEAP Managed by allocator/collector.
```

```
| STATIC DATA Uninitialized (“b[lock] s[tatic] s[torage]’’)
| ↓ Initialized (in object file)
```

```
| CODE Read-only (in object-file)
```

```
Lexical Scope

VAR a : INT = 0
PROCEDURE f(b:INT) IS
    RETURN a+b
END
PROCEDURE g(c:INT) IS
    VAR a : INT = 100
    a := a + 100;
    RETURN f(c)
END
a := -100;
PRINT(g(10))

When \( f \) is executed, it needs to look up a value for \( a \), which is a **free variable** of \( f \), i.e., not a local variable or parameter. Which \( a \) is **visible** (or **in scope**)?

Under **lexical scope** rules, variables are identified by looking backwards through the program **text** to find the nearest enclosing declaration.

- In this case, \( f \) would use the global declaration of \( a \).
- At the time \( f \) executes, this has the value \(-100\), so \(-90\) is printed.
- Nearly all languages use lexical scope.
Dynamic Scope

VAR a : INT = 0
PROCEDURE f(b:INT) IS
    RETURN a+b
END
PROCEDURE g(c:INT) IS
    VAR a : INT = 100
    a := a + 100;
    RETURN f(c)
END
a := -100;
PRINT(g(10))

Under **dynamic scope** rules, variables are identified by looking backwards through the program execution to find the most recently executed declaration that is still active.

- In this case, \( f \) would use the local declaration of \( a \) within \( g \).
- At the time \( f \) executes, this has the value 200, so 210 is printed.
- Early versions of LISP and some scripting languages use dynamic scope.
- Dynamic scope makes procedures act like macros.
Nested procedure declarations

```ml
fun addtoall (a: int, u: int list) : int list =
  let fun f (v : int list) : int list =
    case v of
      nil => nil
    | (h::t) => (h+a)::(f t)
  in f u
end

val w = addtoall (3, [1,2,3]) (* yields [4,5,6] *)
```

- In Pascal, Ada, ML, etc., we can nest procedure declarations inside other procedure declarations. (Cannot do this in C,C++,Java!)

- Parameters and local variables of outer procedures are visible within inner procedures (using lexical scoping rules).

- Purpose: localize scope of nested procedures, and avoid the need to pass auxiliary parameters defined in outer scopes.

- In most languages, if procedure \( f \) is declared inside \( g \), then \( f \) can only appear as a descendent of \( g \) in the procedure activation tree. This allows us to stack-allocate activation records, and still guarantee that non-local variables will still exist when they are needed.
Implementing Nested Procedures

Suppose procedure $f$ uses non-local variables? How are they found?

In an interpreter, it suffices to know the environment that was in place when the declaration of $f$ was encountered.

- The interpreter can simply attach the current variable environment to its description of $f$ when it encounters $f$’s declaration and records it in the function environment.

- When the interpreter applies $f$, it evaluates its body in an initial environment taken from the recorded description, which is then extended with $f$’s parameters and locals.

- When the interpreter looks up a variable while executing $f$, it looks first among $f$’s locals and parameters, and then in the lexically-enclosing environment.

A compiler can precompute the positions of variables relative to the start of each procedure activation record, supporting fast runtime access to variables. It is still necessary to pass to each nested function (at runtime) a pointer to its lexically enclosing function’s activation record, or equivalent information. Thus, nested functions carry some (modest) runtime cost.