Topics

- Syntax and Semantics
- Static Semantics
  - Attribute grammars
- Dynamic Semantics
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics

Static Semantics

- Nothing to do with meaning
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., type compatibility rules)
  - Non-context-free (e.g., variables must be declared before they are used)

Is it legal in C or Java?

- int A, B;
- double C;
- A=B+C;

Attribute Grammars : Definition

- Def: An attribute grammar is a context-free grammar $G = (S, N, T, P)$ with the following additions:
  - For each grammar symbol *x* there is a set *A(x)* of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots , A(X_n))$ define *synthesized attributes*
- Functions of the form $I(X_j) = f(A(X_0), \ldots , A(X_{j-1}))$ define *inherited attributes*
- Initially, there are *intrinsic attributes* on the leaves

Attribute Grammars: An Example

- Syntax

```
<assign> -> <var> = <expr>
<expr> -> <var> + <var> | <var>
<var> -> A | B | C
```

- `actual_type`: synthesized for `<var>` and `<expr>`
- `expected_type`: inherited for `<expr>`

Attribute Grammar (continued)

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

Attribute Grammars (continued)

- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

(Dynamic) Semantics

- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies
- Dynamic Semantics
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics

Operational Semantics

- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine is needed

Operational Semantics: an example

C Statement	*Meaning*

```
C Statement                          Meaning
for (expr1; expr2; expr3) {                  expr1;
  ...                                loop:   if expr2 == 0 goto out
}                                            ...
                                             expr3;
                                             goto loop
                                     out:    ...
```

Operational Semantics (continued)

- Uses of operational semantics:
  - Language manuals and textbooks
  - Teaching programming languages

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

Denotational Semantics – continued

- The process of building a denotational specification for a language:
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- Named *denotational* because the mathematical objects denote the meaning of their corresponding syntactic entities.

Denotational Semantics: program state

- The state of a program is the values of all its current variables

$$s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$$

- Let **VARMAP** be a function that, when given a variable name and a state, returns the current value of the variable

$$\text{VARMAP}(i_j, s) = v_j$$

Decimal Numbers

<dec_num> → '0' | '1' | '2' | '3' | '4' | '5' |
'6' | '7' | '8' | '9' |
<dec_num> ('0' | '1' | '2' | '3' |
'4' | '5' | '6' | '7' |
'8' | '9')

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)

M_{dec} (<dec_num> '1’) = 10 * M_{dec} (<dec_num>) + 1

…

M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Expressions

- Map expressions onto Z ∪ {error}

<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *

Expressions

```
Me(<expr>, s) Δ=
    case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
           if VARMAP(<var>, s) == undef
                then error
                else VARMAP(<var>, s)
     <binary_expr> =>
          if (Me(<binary_expr>.<left_expr>, s) == undef
                OR Me(<binary_expr>.<right_expr>, s) =
                              undef)
               then error
          else
          if (<binary_expr>.<operator> == '+' then
             Me(<binary_expr>.<left_expr>, s) +
                    Me(<binary_expr>.<right_expr>, s)
          else Me(<binary_expr>.<left_expr>, s) *
               Me(<binary_expr>.<right_expr>, s)
       ...
```

Assignment Statements

- Maps state sets to state sets U {`error`}

$$
\begin{aligned}
M_a(x = E, s) \;\Delta= \;& \text{if } M_e(E, s) == \textbf{error} \\
& \text{then } \textbf{error} \\
& \text{else } s' = \{<i_1, v_1'>, <i_2, v_2'>, \ldots, <i_n, v_n'>\}, \text{ where} \\
& \qquad \text{for } j = 1, 2, \ldots, n \\
& \qquad\quad \text{if } i_j == x \\
& \qquad\qquad \text{then } v_j' = M_e(E, s) \\
& \qquad\qquad \text{else } v_j' = \text{VARMAP}(i_j, s)
\end{aligned}
$$

Logical Pretest Loops

- Maps state sets to state sets U {`error`}

$$
\begin{aligned}
M_l(\texttt{while}\ B\ \texttt{do}\ L,\ s)\ \Delta{=}\ &\text{if } M_b(B, s) == \textbf{undef} \\
&\quad\text{then } \textbf{error} \\
&\quad\text{else if } M_b(B, s) == \text{false} \\
&\qquad\text{then } s \\
&\qquad\text{else if } M_{sl}(L, s) == \textbf{error} \\
&\qquad\quad\text{then } \textbf{error} \\
&\qquad\quad\text{else } M_l(\texttt{while}\ B\ \texttt{do}\ L,\ M_{sl}(L, s))
\end{aligned}
$$

Axiomatic Semantics

- Based on mathematical logic
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language
- The logic expressions are called *assertions*

Axiomatic Semantics (continued)

- An assertion before a statement (a *precondition*) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a *postcondition*
- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition

Axiomatic Semantics Form

- Pre-, post form: `{P} statement {Q}`
- An example
  - `a = b + 1  {a > 1}`
  - One possible precondition: `{b > 10}`
  - Weakest precondition: `{b > 0}`

Axiomatic Semantics: Assignment

- An axiom for assignment statements (x = E): $\{Q_{x->E}\}$ x = E {Q}
- `a = b + 1 {a > 1}`
- Q: a>1
- P or $Q_{x->E}$: a>1 or b+1>0 or b>0

Axiomatic Semantics: Sequences

- An inference rule for sequences of the form S1; S2

{P1} S1 {P2}

{P2} S2 {P3}

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

Axiomatic Semantics: Selection

- An inference rules for selection
  - **`if`** B **`then`** S1 **`else`** S2

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$

- If $x>0$ then $y=y-1$ else $y=y+1$ $\{y>0\}$
- $y=y-1$ $\{y>0\}$, P: $\{y>1\}$
- $y=y+1$ $\{y>0)$, P: $\{y>-1\}$
- P: $\{y>1\}$ for the if statement

Axiomatic Semantics: Loops

- **`while`** B **`do`** S **`end`**

  {P} **`while`** B **`do`** S **`end`** {Q}

- The key is to find the loop invariant I

Axiomatic Semantics: Loop Invariant

- Characteristics of the loop invariant: I must meet the following conditions:
  - P => I -- the loop invariant must be true initially
  - {I and B} S {I} -- I is not changed by executing the body of the loop
  - (I and (not B)) => Q -- if I is true and B is false, Q is implied
  - The loop terminates -- can be difficult to prove
- while y != x do y=y+1 {y==x}
- I: y<=x
- P: y<=x

Reading Assignment

- Read Sections 3.4 and 3.5