Semantics

• “Semantics” has to do with the meaning of a program.

• We will consider two types of semantics:
  – Static semantics: semantics which can be enforced at compile-time.
  – Dynamic semantics: semantics which express the run-time meaning of programs.
Static semantics

• Semantic checking which can be done at compile-time

• Type-compatibility
  – int can be assigned to double (type coercion)
  – double cannot be assigned to int without explicit type cast

• Can be captured in grammar, but only at expense of larger, more complex grammar
Type rules in grammar

- Must introduce new non-terminals which encode types:
- Instead of a generic grammar rule for assignment:
  - `<stmt> → <var> '=' <expr> ';'
- we need multiple rules:
  - `<stmt> → <doubleVar> '=' <intExpr> | <doubleExpr> ';'
  - `<stmt> → <intVar> '=' <intExpr> ';'
- Of course, such rules need to handle all the relevant type possibilities (e.g. `byte`, `char`, `short`, `int`, `long`, `float` and `double`).
Attribute grammars

- Attribute grammars provide a neater way of encoding such information.
- Each syntactic rule of the grammar can be decorated with:
  - a set of semantic rules/functions
  - a set of semantic predicates
Attributes

• We can associate with each symbol $X$ of the grammar a set of attributes $A(X)$. Attributes are partitioned into:

  synthesized attributes $S(X) –$ pass info up parse tree

  inherited attributes $I(X) –$ pass info down parse tree
Semantic rules/functions

- We can associate with each rule $R$ of the grammar a set of semantic functions.

- For rule $X_0 \rightarrow X_1 \ X_2 \ \ldots \ X_n$
  - synthesized attribute of LHS:
    $$S(X_0) = f(A(X_1), A(X_2), \ldots, A(X_n))$$
  
  - inherited attribute of RHS member:
    For $1 \leq j \leq n$, $I(X_j) = f(A(X_0), \ldots, A(X_{j-1}))$
    (note that dependence is on siblings to left only)
Predicates

- We can associate with each rule $R$ of the grammar a set of semantic predicates.

- Boolean expression involving the attributes and a set of attribute values

- If $true$, node is ok

- If $false$, node violates a semantic rule
Example

\[
\begin{align*}
\langle \text{assign} \rangle & \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \\
\text{<expr>.expType} & \leftarrow \text{<var>.actType} \\
\langle \text{expr} \rangle & \rightarrow \langle \text{var} \rangle[2] + \langle \text{var} \rangle[3] \\
\text{<expr>.actType} & \leftarrow \text{if (var[2].actType = int) and (var[3].actType = int) then int else real} \\
\text{<expr>.actType} & == \langle \text{expr} \rangle .\text{expType} \\
\langle \text{expr} \rangle & \rightarrow \langle \text{var} \rangle \\
\text{<expr>.actType} & \leftarrow \text{<var>.actType} \\
\text{<expr>.actType} & == \langle \text{expr} \rangle .\text{expType} \\
\langle \text{var} \rangle & \rightarrow A | B | C \\
\text{<var>.actType} & \leftarrow \text{lookUp(<var>.string)}
\end{align*}
\]
Suppose:

- A is int
- B is int
Suppose:
A is int
B is int

actual type = int

<assign>

A = A + B

<expr>

expected type = int

<var>[2]

actual type = int

<var>[3]

actual type = int

actual type = int

A

A

B
Suppose:

A is real
B is int

actual type = real
actual type = int
expected type = real

actual type = real
int → real

type coercion during ‘+’:
Suppose:
A is int
B is real

expected type = int
actual type = int

actual type = real

Houston, we have a problem! Semantic predicate is false.
Dynamic semantics

- Dynamic semantics precisely specify the meanings of programs.
- Why is this important?
  - Programmers need to understand the meanings of programs they read
  - Programmers need to understand how to express themselves in a given language
  - Compiler writers need to understand how to translate a high-level program into a semantically equivalent low-level one
- Formal semantics are not always used – no single formalism exists. We will look at:
  - Operational semantics
  - Axiomatic semantics
- There is also denotational semantics. Look in text if you’re interested.
Operational semantics

• Basic idea: we describe the meanings of program elements by describing how a machine executes them.
• Basic approach: express semantics of a high-level construct in terms of its translation into a low-level construct
• Example: Java described in terms of JVM
• Example: Pascal described in terms of p-code
• Example: C described in terms of PDP-11 assembly
Example

**high-level program**

```
for (e1; e2; e3) { 
    .
    .
    .
}
<rest of program>
```

**operational semantics**

```
e1;
L1: if not(e2) goto L2
    .
    .
    .
e3;
goto L1
L2: <rest of program>
```
Axiomatic semantics

• Basic idea: you make assertions about statements in a program.
• A *precondition* is an assertion about what is true prior to execution of a statement.
• A *postcondition* is an assertion about what is true after execution of a statement. For example:

\[
\text{sum} = 2*\text{sum} + 1 \{ \text{sum} > 1 \}
\]
Weakest precondition

“"The **weakest precondition** is the least restrictive precondition that will guarantee the validity of the associated postcondition."” [p. 151]

In example above \( \{ \text{sum} > 10 \} \) is a valid precondition but \( \{ \text{sum} > 0 \} \) is the **weakest** precondition:

\[
\{ \text{sum} > 0 \} \; \text{sum} = 2*\text{sum} + 1 \; \{ \text{sum} > 1 \}
\]
Correctness proofs

• “If the weakest precondition can be computed from the given postconditions for each statement of a language, then correctness proofs can be constructed for programs in that language.” [p. 151]

• To do this, start with output specifications for the program as the postcondition for the program as a whole.
Next, work backwards, computing weakest preconditions one statement at a time, to derive the weakest precondition for the program as a whole:

\[
\text{stmt1} \ {c0} \ \text{stmt1} \ {c1} \\
\text{stmt2} \ \{c1\} \ \text{stmt2} \ {c2} \\
\dotsc \\
\text{stmtN} \ {cN} \ \{cN-1\} \ \text{stmtN} \ {cN}
\]
• If the input specification for the program satisfies the program’s weakest precondition, then the program will (provably) produce the correct result.
Assignment statements

• The axiomatic semantics of an assignment statement \( x = E \) is written as \( P = Q_{x \to E} \).

• This means that the precondition is the postcondition with all instances of \( x \) replaced by \( E \).

• Example:

  \[
  a = \frac{b}{2} - 1 \quad \{a < 10\}
  \]

To compute precondition, replace all instances of \( a \) in postcondition \( a < 10 \) by \( \frac{b}{2} - 1 \):

\[
\frac{b}{2} - 1 < 10, \text{ or }
\]

\[
b < 22
\]

• Semantics: \( \{b < 22\} \; a = \frac{b}{2} - 1 \; \{a < 10\} \)

• In general: \( \{Q_{x \to E}\} \; x = E \; \{Q\} \)
Inference

• Suppose we wanted to prove the following:
  \{\text{sum} > 5\} \text{ sum} = 2\times\text{sum} + 1 \ {\text{sum} > 3}\}

• Starting with the postcondition, we derive as the weakest precondition something different:
  \begin{align*}
  2\times\text{sum} + 1 &> 3 \\
  2\times\text{sum} &> 2 \\
  \text{sum} &> 1 \quad \text{(weakest precondition)}
  \end{align*}

• Clearly this is OK, because the actual precondition \{\text{sum}>5\} implies the weakest precondition \{\text{sum}>1\}:
  \text{sum}>5 \Rightarrow \text{sum}>1

• We use an inference rule, the rule of consequence, to prove this.
Rule of consequence

• General form:
  \[ S_1, S_2, \ldots, S_n \]
  \[ S \]

• Precondition strengthening:
  \[ P \Rightarrow P', \quad \{P'\} \subseteq \{Q\} \]
  \[ \{P\} \subseteq \{Q\} \]

• Postcondition weakening:
  \[ Q' \Rightarrow Q, \quad \{P\} \subseteq \{Q'\} \]
  \[ \{P\} \subseteq \{Q\} \]
Precondition strengthening applied

• Recall our example. We want to prove the following:
  \[{\text{sum} > 5} \implies \text{sum} = 2\times\text{sum} + 1 \implies \text{sum} > 3\]

• The weakest precondition we derive is:
  \[\text{sum} > 1\]

• Apply precondition strengthening to finish proof:

\[
\begin{align*}
P &\implies P', \quad \{P'\} \subseteq \{Q\} \\
\{P\} &\subseteq \{Q\}\end{align*}
\]

\[
\begin{align*}
\text{sum} > 5 &\implies \text{sum} > 1, \quad \{\text{sum} > 1\} \implies \text{sum} = 2\times\text{sum} + 1 \implies \text{sum} > 3 \\
\{\text{sum} > 5\} &\implies \text{sum} = 2\times\text{sum} + 1 \implies \text{sum} > 3
\end{align*}
\]
Hang on!

- Why get so formal?
- Because this way correctness proofs can be (partially) automated.
Sequences:
S1  S2

Start with a sequence of statements:
y = 3*x+1; \quad x = y+3; \quad \{x<10\}

Compute weakest precondition for 2\textsuperscript{nd} stmt:
\{y+3<10\} or \{y<7\}, use as postcondition
for 1\textsuperscript{st} stmt:
y = 3*x+1; \quad \{y<7\} \quad \{y<7\} x = y+3; \quad \{x<10\}

Compute weakest precondition for 1\textsuperscript{st} stmt:
\{3x+1<7\} or \{x<2\}
\{x<2\} y = 3*x+1; \quad \{y<7\} \quad \{y<7\} x = y+3; \quad \{x<10\}

Conclude, applying sequence rule:
\{x<2\} y = 3*x+1; x = y+3; \quad \{x<10\}
Start with a conditional statement:
if (x>0) then y=y-1 else y=y+1 {y>0}
Deal with arms of the conditional one at a time, first the then-arm:
y = y-1 {y>0} \Rightarrow \{y>1\} y = y-1; \{y>0\}
Now the else-arm:
y = y+1 {y>0} \Rightarrow \{y>-1\} y = y+1; \{y>0\}
Use rule of precondition strengthening on the else-arm result (to make both arms uniform):
\{y>1\} y = y+1; \{y>0\}
Now strengthen both arms’ preconditions by imposing a constraint on x:
\{ x>0 \ & \ y>1 \} y = y-1; \{y>0\}
\{!(x>0) \ & \ y>1 \} y = y+1; \{y>0\}
Conclude, applying selection rule:
\{y>1\} if (x>0) then y=y-1 else y=y+1 \{y>0\}
Let’s prove the following:

\{\text{true}\}

\[
r=x; \quad q=0; \text{while } y\leq r \text{ do } r=r-y; \quad q=q+1; \text{ end}
\]

\{y>r \land x=r+y*q\}

Start by proving loop body:

\{y\leq r \land x=r+y*q\} \quad r=r-y; \quad q=q+1; \{x=r+y*q\}

Start with last statement:

\[
q=q+1 \quad \{x=r+y*q\} \Rightarrow \{x=r+y*(q+1)\} \quad q=q+1 \quad \{x=r+y*q\}
\]

Continue with second-to-last statement:

\[
r=r-y \quad \{x=r+y+y*q\} \Rightarrow \{x=r-y+y+y*q\} \quad r=r-y \quad \{x=r+y+y*q\}
\]

Use rule for sequence to get:

\[
\{x=r+y*q\} \quad r=r-y; \quad q=q+1; \quad \{x=r+y*q\}
\]

Now strengthen precondition to conclude proof of loop body:

\{y\leq r \land x=r+y*q\} \quad r=r-y; \quad q=q+1; \quad \{x=r+y*q\}

This lets us derive a weakest precondition for the while loop:

\[
\{x=r+y*q\} \quad \text{while } y\leq r \text{ do } r=r-y; \quad q=q+1; \text{ end} \quad \{x=r+y*q \land !(y\leq r)\}
\]
The next step is to prove the sequence
\{true\} \ r=x; \ q=0; \ while \ y\leq r \ do \ r=r-y; \ q=q+1; \ end \ \{x=r+y*q & y>r\}\}
Start by moving backwards from the while loop (since we derived a weakest precondition from its postcondition already):
\{true\} \ r=x; \ q=0; \ {x=r+y*q}
Start with last statement:
q=0; \ {x=r+y*q} \ \Rightarrow \ \{x=r+y*0\} \ q=0 \ {x=r+y*q}
\{x=r\} \ q=0 \ {x=r+y*q}
Continue with second-to-last statement:
\ r=x \ \{x=r\} \ \Rightarrow \ \{x=x\} \ r=x \ \{x=r\}
Precondition strengthening:
\{true\} \ r=x \ \{x=r\}
Sequence rule (applied in general form):
\{true\} \ r=x; \ q=0; \ while \ y\leq r \ do \ r=r-y; \ q=q+1; \ end \ \{x=r+y*q & !(y\leq r)\}\}
Finally, postcondition weakening because !(y\leq r) \implies \ y>r : \ \{true\} \ r=x; \ q=0; \ while \ y\leq r \ do \ r=r-y; \ q=q+1; \ end \ \{x=r+y*q & y>r\}\}
We're done!
Loop invariant

• Recall the While rule:

\[
\{B \& I\} \ S \ \{I\} \\
\{I\} \ \text{while } B \ \text{do } S \ \text{end} \ \{I \text{ and } !B\}
\]

• \(I\) is called the *loop invariant*:

  – "…if executing [the body] once preserves the truth of [the invariant], then executing [the body] any number of times also preserves the truth of [the invariant]." [Gordon, *Programming Language Theory and its Implementation*, paraphrased from page 24]
Importance of loop invariants

- Developing loop invariants are a powerful way to design and understand algorithms.
- Consider selection sort:

```java
selectionSort(int[] array) {
    int min, temp, bar=0;
    while (bar < array.length - 1) {
        min = indexOfSmallest(array, bar);  // find min
        temp = array[bar];                  // swap
        array[bar] = array[min];
        array[min] = temp;
        bar = bar + 1;
        // Loop invariant: region before bar is sorted
        // for all i,j<=bar, if i<j, array[i] <= array[j]
    }
}
```
Example

[ 7 4 6 8 3 2 ]
[ 2 4 6 8 3 7 ]
[ 2 3 6 8 4 7 ]
[ 2 3 4 8 6 7 ]
[ 2 3 4 6 8 7 ]
[ 2 3 4 6 7 8 ]

region not known to be sorted
region known to be sorted