Outline

• Semantics
• Attribute Grammars
• Categories of Semantics
  • Operational
  • Denotational
  • Axiomatic
Semantics

• the *meaning* of the expressions, statements, and program units

• Why care? So we...
  • Know how a language works
  • Understand what various statements mean
  • Improve our ability to learn a new language quickly
Attribute Grammars
Attribute Grammars

• Addition to the **syntactic** grammar of a language
• Describes a small subset of **semantic** behavior

• Why care?
  • Static semantics specification
  • Compiler design (static semantics checking)
Attribute Grammars

• Each grammar symbol has:
  • A set of attribute values $A$

• Each grammar rule has:
  • a set of functions $F$ that define certain attributes of the nonterminals in the rule
  • a (possibly empty) set of predicates $P$ to check for attribute consistency

Remember, a Grammar already has:

Start, Nonterminals, Terminals, Rules
Attribute Grammars

**Rules** have the form:

\[ X_0 \rightarrow X_1 \ldots X_n \]

We also have:

- **Synthesized Attributes** – a.k.a. information which is realized during the parsing (bottom – up)
- **Inherited Attributes** – a.k.a. information which is defined based on the structure (top – down)
- **Intrinsic Attributes** – a.k.a. static information affixed to Leaves/Terminals
Example Attribute Grammar

• Syntax

<assign>  →  <var>  =  <expr>
<expr>    →  <var> + <var> | <var>
<var>     →  A | B | C

• Attributes:
  • actual_type: synthesized for <var> and <expr>
  • expected_type: inherited for <expr>
Example Attribute Grammar

- **Syntax Rule:**
  \[ <\text{expr}> \rightarrow <\text{var}>[1] + <\text{var}>[2] \]

- **Semantic Rules:**
  \[ <\text{expr}>.\text{actual\_type} \leftarrow <\text{var}>[1].\text{actual\_type} \]

- **Predicates:**
  \[ <\text{var}>[1].\text{actual\_type} == <\text{var}>[2].\text{actual\_type} \]
  \[ <\text{expr}>.\text{expected\_type} == <\text{expr}>.\text{actual\_type} \]
Example Attribute Grammar

• Syntax Rule:
  `<var>  →  id`

• Semantic Rules:
  `<var>.actual_type  ←  lookup(id.string)`

• Predicates:
  `None`
Example Attribute Grammar

• Syntax Rule:
  \(<assign> \rightarrow <var> = <expr>\)

• Semantic Rules:
  \(<expr>.expected_type \leftarrow <var>.actual_type\)

• Predicates:
  \(<var>.actual_type == <expr>.actual_type\)
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 *most cases*, both kinds of attributes are used, so we use some combination of top-down and bottom-up
Example

Suppose we have the following code:

\[ z = x + y \]

Type Information:

• \( x \) is an \textit{int}
• \( y \) is an \textit{int}
• \( z \) is a \textit{string}
Example

\[ z = x + y \]

Predicates:

- `<var>[1].actual_type == <var>[2].actual_type`
- `<expr>.expected_type == <expr>.actual_type`

Predicates:

- `<var> = id`
- `<var].actual_type <= lookup(id.string)`
Example

\[ x = x + y \]
Categories of Semantics
Operational Semantics

• Describe the meaning of a program by executing its statements on a machine
  • The machine can be 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

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

• Evaluation
  • Good if used informally (language manuals, etc.)
  • Extremely complex if used formally
Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- The process of building a denotational specification for a language:
  1. Define a mathematical object for each language entity
  2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables
Denotational Semantics

\[ \langle \text{digit} \rangle \rightarrow \ '0' \mid '1' \mid '2' \mid '3' \mid '4' \]
\[ \quad \mid '5' \mid '6' \mid '7' \mid '8' \mid '9' \]

\[ \langle \text{dec_num} \rangle \rightarrow \ \langle \text{digit} \rangle \mid \langle \text{dec_num} \rangle \ \langle \text{digit} \rangle \]

\[ M_{\text{dec}}('0') = 0 \]
\[ M_{\text{dec}}('1') = 1 \]
\[ \ldots \]
\[ M_{\text{dec}}('9') = 9 \]

\[ M_{\text{dec}}(\langle \text{dec_num} \rangle '0') = 10 \times M_{\text{dec}}(\langle \text{dec_num} \rangle) \]
\[ M_{\text{dec}}(\langle \text{dec_num} \rangle '1') = 10 \times M_{\text{dec}}(\langle \text{dec_num} \rangle) + 1 \]
\[ \ldots \]
\[ M_{\text{dec}}(\langle \text{dec_num} \rangle '9') = 10 \times M_{\text{dec}}(\langle \text{dec_num} \rangle) + 9 \]
Operational vs. Denotational

• In **operational semantics**, the state changes are defined by coded algorithms

• In **denotational semantics**, the state changes are defined by rigorous mathematical functions
  • We only looked at defining a number
  • Imagine an entire program/loop!
Axiomatic Semantics

• Based on **formal logic** (predicate calculus)

• Original purpose
  • Formal Program Verification

• Axioms are defined for each statement type in the language
  • to allow transformations of logic expressions into more formal logic expressions
  • Also known as inference rules

• The logic expressions are called **assertions**