Describing Semantics

Programming Languages

William Killian

Millersville University

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, ***N***onterminals, ***T***erminals, ***R***ules

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:

```
<expr> → <var>[1] + <var>[2]
```

- Semantic Rules:

```
<expr>.actual_type ← <var>[1].actual_type
```

- Predicates:

```
<var>[1].actual_type == <var>[2].actual_type

<expr>.expected_type == <expr>.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> → <var> = <expr>

- Semantic Rules:

  <expr>.expected_type ← <var>.actual_type

- Predicates:

  <var>.actual_type == <expr>.actual_type

How are Attribute Values Computed?

- If all attributes where **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 *<u>most cases</u>*, 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 `int`
- y is an `int`
- z is a `string`

Example

```
<expr> → <var>[1] + <var>[2]
<expr>.actual_type ← <var>[1].actual_type
Predicates:
<var>[1].actual_type == <var>[2].actual_type
<expr>.expected_type == <expr>.actual_type
```

```
<var> → id
<var>.actual_type ← lookup(id.string)
Predicates:
None
```

```
<assign> → <var> = <expr>
<expr>.expected_type ← <var>.actual_type
Predicates:
<var>.actual_type == <expr>.actual_type
```

z = x + y

<assign>
actual=
expected=

<var>
actual=

=

<expr>
actual=
expected=

id
string=**z**

<var>[1]
actual=

+

<var>[2]
actual=

id
string=**x**

id
string=**y**

Example

```
<expr> → <var>[1] + <var>[2]
<expr>.actual_type ← <var>[1].actual_type
```
Predicates:
```
<var>[1].actual_type == <var>[2].actual_type
<expr>.expected_type == <expr>.actual_type
```

```
<var> → id
<var>.actual_type ← lookup(id.string)
```
Predicates:

None

```
<assign> → <var> = <expr>
<expr>.expected_type ← <var>.actual_type
```
Predicates:
```
<var>.actual_type == <expr>.actual_type
```

```
x = x + y
```

```
<assign>
actual=
expected=

<var>                 =                <expr>
actual=                                actual=
                                       expected=

id               <var>[1]      +      <var>[2]
string=x         actual=              actual=

                 id                   id
                 string=x             string=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

```
<digit> -> '0' | '1' | '2' | '3' | '4'
         | '5' | '6' | '7' | '8' | ’9’

<dec_num> →  <digit> | <dec_num> <digit>
```

$M_{dec}(\text{'0'}) = 0$

$M_{dec}(\text{'1'}) = 1$

...

$M_{dec}(\text{'9'}) = 9$

$M_{dec}(\text{<dec\_num> '0'}) = 10 * M_{dec}(\text{<dec\_num>})$

$M_{dec}(\text{<dec\_num> '1’}) = 10 * M_{dec}(\text{<dec\_num>}) + 1$

...

$M_{dec}(\text{<dec\_num> '9'}) = 10 * M_{dec}(\text{<dec\_num>}) + 9$

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**