Describing Semantics of Programming Languages

Two Semantics of Languages

- **Static semantics**
  - Meanings that can be determined statically (at compile time)
- **Dynamic semantics**
  - Meanings that can be determined dynamically

Example: Static Semantics

- Context-free but cumbersome
  - Type checking
- Noncontext-free
  - Variables must be declared before they are used.

How to Describe Static Semantics Formally?

- CFGs **cannot** describe all of the static semantics of programming languages.
- Need additions to CFGs to carry some semantic info. along through parse trees.
  - **Attribute Grammars (AG)**

Attribute Grammars (AG)

- Attribute Grammars (AGs)
  = **CFG + Additional features**
- Primary value of AGs:
  - **Static semantics specification**
  - **Static semantics checking**

Attribute Grammar

- An attribute grammar (AG) is a CFG G = (T, N, S, P) with the following additions:
  - Each grammar symbol X has
    - A set A(X) of **attributes**
  - Each rule has
    - A set of **semantic functions** that define certain attributes of the non-terminals in the rule.
    - A (possibly empty) set of **predicates** to check for attribute consistency.

Attributes

- Each grammar symbol X has a set A(X) of attributes.
- Two kinds of attributes:
  - S(X): **Synthesized attributes**
    - To pass semantic info up a parse tree.
  - I(X): **Inherited attributes**
    - To pass semantic info down a parse tree.

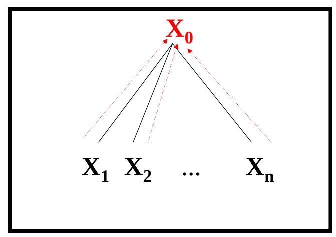
Attributes

- **Intrinsic attributes**
  - Synthesized attributes of leaf nodes in a parse tree
  - Whose values are determined outside the parse tree and given.
  - Initially, there are *intrinsic attributes* on the leaves

Semantic Functions

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule & $S(X_0)$ = The synthesized attributes of X_0.
- The synthesized attributes of X_0 are computed by a semantic function of the form:
  - $S(X_0) = f(A(X_1), \ldots A(X_n))$
  - Depends only on the attributes of the node's children nodes!

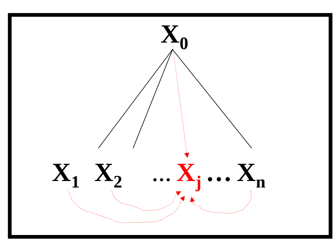
Synthesized Attributes



Semantic Functions

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule & $I(X_j)$ = The inherited attributes of X_j where 1<=j<=n.
- The inherited attributes of X_j are computed by a semantic function of the form:
  - $I(X_j) = f(A(X_0), \ldots, A(X_n))$
  - Depends on the attributes of the node's parent and sibling nodes!

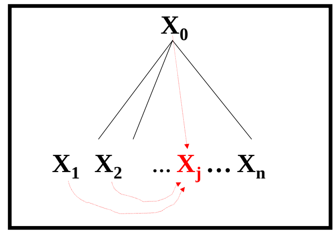
Inherited Attributes



L-Inherited Attributes

- The inherited attributes of X_j are computed by a semantic function of the form:
  - $I(X_j) = f(A(X_0), \ldots, A(X_{j-1}))$
  - Depends on the attributes of the node's parent and left sibling nodes!
  - **L-attributed attribute**

L-Inherited Attributes



Predicates

- A **predicate** has the form of a Boolean expression on the attribute set $\{A(X_0), \ldots, A(X_n)\}$.
- Derivations are allowed:
  - Every predicate associated with every non-terminal is true.

Example: CFG of Ada Procedures

- The name on the end of an Ada procedure must match the procedure's name.

CFG:
<proc_def> → **procedure** <proc_name> <proc_body>
end <proc_name>

Example: Attribute Grammar of Ada Procedures

- The name on the end of an Ada procedure must match the procedure's name.

AG:
<proc_def> → **procedure** <proc_name> <proc_body>
end <proc_name>
Attribute:
string

Semantic function:
<proc_name>.string = <proc_name>.string

Example: Attribute Grammar of Ada Procedures

- The name on the end of an Ada procedure must match the procedure's name.

AG:
<proc_def> → **procedure** <proc_name>[1] <proc_body>
end <proc_name>[2]
Attribute:
string

Semantic function:
<proc_name>[1].string = <proc_name>[2].string

Example: Assignment Statements

- A simple assignment statement.

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

Example: Type Rules

- The type rules of a simple assignment statement:
  - The variables can be one of two types: int or real.
  - The type of the expression is that of its operands if the same. Otherwise real.
  - The type of LHS of an assignment must match the type of RHS.

A := A + B

A: real
B: int

Example: Attributes

- **Attributes:**
  - actual_type = A synthesized attribute for <var> and <expr>
  - expected_type = An inherited attribute for <expr>

Example: Attribute Grammar of Assignment Statements Types

AG:
Syntax rule:
<assign> → <var> := <expr>

Semantic rule:

<expr>.expected_type ← <var>.actual_type

Example: Attribute Grammar Assignment Statements Types

AG:
Syntax rule:
<expr> → <var>[1] + <var>[2]

Semantic rule:

<expr>.actual_type ← if (<var>[1].actual_type = int) and (<var>[2].actual_type = int) then int else real

Predicate:
<expr>.actual_type = <expr>.expected_type

Example: Attribute Grammar Assignment Statements Types

AG:
Syntax rule:
<expr> → <var>

Semantic rule:

<expr>.actual_type ← <var>.actual_type

Predicate:

<expr>.actual_type = <expr>.expected_type

Example: Attribute Grammar Assignment Statements Types

AG:

Syntax rule:

<var> → A | B | C

Semantic rule:

<var>.actual_type ← look-up(<var>.string)

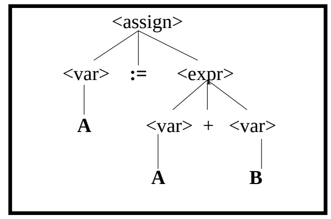
Example: Attribute Grammar Assignment Statements Types

CFG:

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

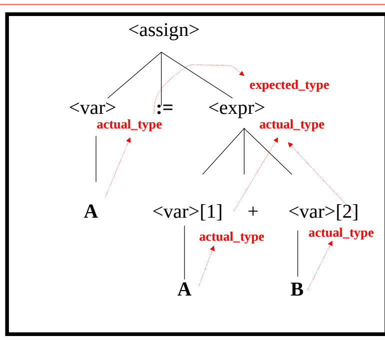
A := A + B

A Parse Tree



Example: Flow of Attributes

A := A + B

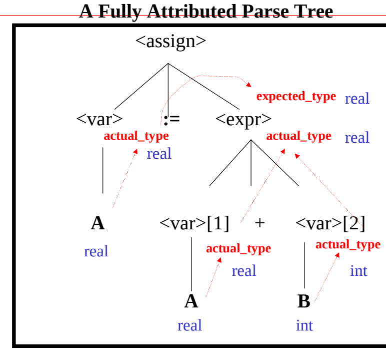


Example: Computing Attributes

A Fully Attributed Parse Tree

A := A + B

A: real
B: int



Computing Attribute Values

- 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 of Languages

- Dynamic Semantics
  - cannot be determined statically (at compile time)
  - can only be determined by executing dynamically

How to Describe Dynamic Semantics?

- Three methods to describe semantics formally:
  - **Operational Semantics**
  - **Axiomatic Semantics**
  - **Denotational Semantics**
- No single widely acceptable notation or formalism for describing semantics

CS216 31

Three Formal Methods

- **Operational Semantics**
  - By using operations of an actual or hypothetical machine.
- **Axiomatic Semantics**
  - By using mathematical logic.
- **Denotational Semantics**
  - By using mathematical functions.

CS216 32

Three Formal Methods

- All these methods are **syntax-directed**.
  - The semantic definitions are based on a CFG or BNF rule.

CS216 33

1. Operational Semantics

- Based on machines.
- Describe the meaning of a program by specifying how the program is to be executed on a machine whose operations are completely known.

CS216 34

Operational Semantics

- To use operational semantics for a high-level language, a defining machine in needed.
- Focuses on the individual steps by which each program is executed.
- The change in the state of the machine (memory, registers & etc.) defines the meaning of the program.

CS216 35

Operational Semantics: Evaluation

- Give useful insight into the way the program is implemented.
- Too much details – hard to understand the net effect of executing a program.
- Good if used informally
  - Extremely complex if used formally.

CS216 36

2. Axiomatic Semantics

- Based on formal logic (first order predicate calculus).
- Describe the meaning of a program by describing the effect its execution has on assertions about the data manipulated by the program.

CS216 37

Axiomatic Semantics

- Precondition:
  - An assertion before a statement (the relationships and constraints among variables that are true at that point in execution).
- Postcondition:
  - An assertion following a statement.
- Pre-post form: **{P} statement {Q}**

CS216 38

Example: Axiomatic Semantics

{P} **a = b + 1** {a > 1}

- One possible precondition: {b > 10}
- Weakest precondition:
  - The least restrictive precondition that will guarantee the postcondition.
- Weakest precondition: {b > 0}

CS216 39

Proof of Program Correctness

- Using Axiomatic Semantics:
  - The postcondition for the whole program is the desired results.
  - Work back through the program to the first statement and find the weakest preconditions.
  - If the precondition on the first statement is the same as the program spec, then the program is correct.

CS216 40

Axiomatic Semantics: Evaluation

- Developing axioms or inference rules for all of the statements in a language is difficult.
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs.
- It is not as useful for language users and compiler writers.

CS216 41

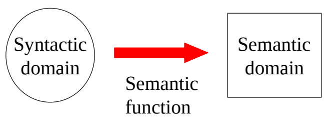
3. Denotational Semantics

- Based on mathematics (recursive function theory).
- Describe the meaning of a program by using mathematical functions.
- The most abstract semantics description method.

CS216 42

Denotational Semantics

- Define syntactic domains.
- Define semantic domains.
- Define semantic functions from a syntactic domain to a semantic domain.



Example: Binary numbers

- The syntax of binary numbers:

```
<bin_num> → 0
          |  1
          | <bin_num> 0
          | <bin_num> 1
```


Example: Binary numbers

- The semantics of binary numbers:
  - The domain of syntactic values = The syntax
  - The domain of semantic values = The set of nonnegative decimal integer values.
  - The semantic function = maps the syntactic objects to the objects in the semantic domain.

Example: Denotational Semantics of Binary numbers

```
<bin_num> → 0
          |  1
          | <bin_num> 0
          | <bin_num> 1
```

0, 1, 2, 3, 4, ………
(Non-negative integer values)

M_b(‘0’) = 0
M_b (‘1’) = 1
M_b (<bin_num> ‘0’) = 2 * M_b (<bin_num>)
M_b (<bin_num> ‘1’) = 2 * M_b (<bin_num>) + 1

Example: Denotational Semantics of 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)
```

0, 1, 2, 3, 4, ………
(Non-negative integer values)

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

Denotational vs. Operational Semantics

- The difference between denotational and operational semantics:
  - In operational semantics, the state changes are defined by coded algorithms.
  - in denotational semantics, they are defined by rigorous mathematical functions.

The State of a Program

- S = The *state* of a program, i.e. the values of all its current variables:

  $s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$
- VARMAP = a function that, when given a variable name and a state, returns the current value of the variable:

  $VARMAP(i_j, s) = v_j$

Denotational Semantics of Expressions

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

M_e(<expr>, s) =
  case <expr> of
  <dec_num> => M_{dec}(<dec_num>, s)

Denotational Semantics of Expressions

M_e(<expr>, s) =
  case <expr> of
   <var> =>
      if VARMAP(<var>, s) = undef
      then error
      else VARMAP(<var>, s)

Denotational Semantics of Expressions

M_e(<expr>, s) =
  case <expr> of
  <binary_expr> =>
      if (M_e(<binary_expr>.<left_expr>, s) = undef
          OR M_e(<binary_expr>.<right_expr>, s) = undef)
      then error
      else if (<binary_expr>.<operator> = '+'
          then M_e(<binary_expr>.<left_expr>, s) **+** M_e(<binary_expr>.<right_expr>, s)
          else M_e(<binary_expr>.<left_expr>, s) ***** M_e(<binary_expr>.<right_expr>, s)

Denotational Semantics of Assignment Statements

M_a(x := E, s) =
  if M_e(E, s) = error
  then error
  else
    $s' = \{<i_1,v_1'>,<i_2,v_2'>,\ldots,<i_n,v_n'>\}$,
    where for j = 1, 2, ..., n,
        v_j' = VARMAP(i_j, s) if i_j <> x
          = M_e(E, s) if i_j = x

Denotational Semantics of Loops

M_b(B, s): maps boolean exp to boolean values.
M_{sl}(L, s): maps statement lists to states.

M_l(while B do L, s) =
  if M_b(B, s) = undef
  then error
  else if M_b(B, s) = false
    then s
    else if M_{sl}(L, s) = error
      then error
      else M_l(while B do L, M_{sl}(L, s))

Denotational Semantics: Evaluation

- Can be used to prove the correctness of programs.
- Provides a rigorous way to think about programs.
- Can be an aid to language design.
- Has been used in compiler generation systems.