Enforce context-dependent language rules that are not reflected in the grammar, e.g., a function must have a return statement.

The evaluation of the attributes can require an arbitrary number of traversals of the AST: arbitrary context dependence (e.g., the value of a declared constant is found in the constant declaration).

The rule is local: it only refers to other symbols in the same production.

For every symbol in the grammar we define some computable properties (e.g., the value of a constant expression).

For every symbol in the grammar we define some computable properties.

For every symbol in the grammar we define some computable properties associated with the non-terminals.

A systematic process of assigning meanings to programs can be viewed as computation of attributes.

A syntax-directed framework.

Attributes and Attribute Grammars

The concept of a declared constant is found in the constant declaration.

For the properties of all symbols on both sides of the production.

For every production in the grammar we give computation rules for the properties of all symbols on both sides of the production.

For every production in the grammar, we give computation rules for the properties of all symbols on both sides of the production.

A syntax-directed approach to the specification of syntax-directed translations.

Syntax-Directed Translation

A systematic process of assigning meanings to programs can be viewed as computation of attributes.

We consider two general approaches to the specification of syntax-directed translations: syntax-directed rules and translation schemes.

We consider two general approaches to the specification of syntax-directed translations:

The concept of a declared constant is found in the constant declaration.

The evaluation of the attributes can require an arbitrary number of traversals of the AST: arbitrary context dependence (e.g., the value of a declared constant is found in the constant declaration).

The rule is local: it only refers to other symbols in the same production.

For every symbol in the grammar we define some computable properties.

For every symbol in the grammar we define some computable properties associated with the non-terminals.

A systematic process of assigning meanings to programs can be viewed as computation of attributes.

We consider two general approaches to the specification of syntax-directed translations:

The concept of a declared constant is found in the constant declaration.

The rule is local: it only refers to other symbols in the same production.

For every symbol in the grammar we define some computable properties associated with the non-terminals.

A systematic process of assigning meanings to programs can be viewed as computation of attributes.

We consider two general approaches to the specification of syntax-directed translations:

The concept of a declared constant is found in the constant declaration.

The rule is local: it only refers to other symbols in the same production.

For every symbol in the grammar we define some computable properties.

For every symbol in the grammar we define some computable properties associated with the non-terminals.

A systematic process of assigning meanings to programs can be viewed as computation of attributes.

We consider two general approaches to the specification of syntax-directed translations:
### Syntax-Directed Semantics

#### Definition

- **Production**
  - Each production $A \rightarrow \gamma$ is associated with a set of semantic rules of the form $b := f(c_1; \ldots; c_k)$, where $f$ is a function and either
  - $b$ is a synthesized attribute of $A$, and $c_1; \ldots; c_k$ are attributes of the symbols in $\{A:b\}$,
  - or $b$ is an inherited attribute of one of the symbols and $c_1; \ldots; c_k$ are attributes of the symbols in $\{A:b\}$.

#### Example of Synthesized Attributes

Following is an example of the semantic actions of a simple desk calculator:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E$</td>
<td>$\text{print}(E:val)$</td>
</tr>
<tr>
<td>$E \rightarrow E + T$</td>
<td>$E:val := E:val + T:val$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T:val := F:val$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$F:val := E:val$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$F:val := \text{digit}:\text{lexval}$</td>
</tr>
</tbody>
</table>

These can be computed during bottom-up parsing.

#### Example Parse Tree

The parse tree for the declaration `real id1; id2; id3` is given by:

```
addtype(id, entry; L:in)
```

We can use this grammar in order to parse the declaration `real id1; id2; id3` with a type declaration of a list of variables:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow TL$</td>
<td>$\text{addtype}(\text{id}, entry; L:in)$</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>$T:type := \text{integer}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{real}$</td>
<td>$T:type := \text{real}$</td>
</tr>
<tr>
<td>$L \rightarrow L_1; id$</td>
<td>$L:in := \text{addtype}(\text{id}, entry; L:in)$</td>
</tr>
</tbody>
</table>

### Example of Inherited Attributes

Following is an example of inherited attributes associated with a type declaration of a list of variables:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow TL$</td>
<td>$\text{addtype}(\text{id}, entry; L:in)$</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>$T:type := \text{integer}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{real}$</td>
<td>$T:type := \text{real}$</td>
</tr>
<tr>
<td>$L \rightarrow L_1; id$</td>
<td>$L:in := \text{addtype}(\text{id}, entry; L:in)$</td>
</tr>
</tbody>
</table>

---

**Honors Compilers**, NYU, Fall, 2009
Lecture 5: Syntax-Directed Semantics

A.Pnueli

Dependency Graph

Given syntax-directed attribution rules, we can draw the dependency graph, indicating the dependence between attributes in the parse tree, which YACC/BISON is used in order to construct the parse tree.

Syntax Trees as a Synthesized Attribute

General Properties of Attribute Grammars

Attributes are computed by repeated passes over the AST.

In practice, many inherited attributes are handled by means of global data structures (symbol table). The finite-domain restriction is not more expressive than CFG's. The finite-domain restriction is not potentially expressive. An attribute grammar has cycles is decidable but attribute definitions may be cyclic: checking whether attributes are computed by repeated passes over the parse tree.

Useful subsets: S-attribute and L-attribute grammars.

Operations on Computation of Synthesized Attributes

The operations of push and pop are then performed in conjunction for both stacks.

Semantic Rules

Production Rules

Honors Compilers, NYU, Fall, 2009

Honors Compilers, NYU, Fall, 2009
Lecture 5: Syntax-Directed Semantics

A.Pnueli

S-Attributed Grammars

These are grammars which can compute attributes in a general depth-first traversal of the parse tree, according to the following scheme:

- They are usable with bottom-up parsers, which can compute the attributes together with the parsing process.
- These are grammars that have only synthesized attributes.

Translation Schemes

These are grammars rules in which semantic actions are embedded inside the body of a production. Following is an example of a translation scheme which transforms infix expressions into postfix expressions (under LL(1) parsing):

```
E -> TR
R -> addop T [print(advoc(lexeme))] R1 | ε
T -> num [print(lexeme)]
```

The parse tree for the expression 9 − 5 + 2 is given by:

```
  E
  / \  ___
R | T   5
 | |   /   \\
| num | +   2
|     |   /   \       \\
R | T   addop | print(−)   \\
T | print | R1
```

Computing L-Attributes

Typically, L-Grammars are appropriate for LL(1) parsers. LALR parsers as an independent pass, after the syntax tree has been constructed.

In some special cases (as shown below), it is possible to compute L-attributes also while performing LALR parsing. There is a general transformation which converts an arbitrary L-Grammar into an S-Grammar, but it is often clumsy.
Type Inheritance

Generalized code can be treated as synthesized attributes.

Inherited and synthesized: two passes required over

In the presence of overloading, type is both

Synthesized attributes on terminals: names, literal

as one global data structure. Symbol table carries inherited information

Inherited attributes computed during declaration

with localized multiple traversals.

Some systems employ left-to-right, top-down traversals

Attribute Computation and Tree Traversals

Some Important Attributes

For expressions:

.type.

For overloaded calls:

candidate interpretations.

For identifiers:

.entity (dening occurrence).

For definitions:

.scope.

For data/function members:

.visibility (public, protected, private).

For function:

.virtual functions (primitive operations).

Etc, etc.

Honors Compilers, NYU, Fall, 2009

161
Data Structures for Name Resolution

Name Resolution does not require any hashing.

Semantic Actions for Visibility Processing

In a scope:

- On scope entry:
  - Place new scope on stack, initialize list of local entities.
  - For every declared name:
    - Chain name entry to local entity.
    - Set homonym of local entity to outer entity with same name.
    - Update name table.
- On scope exit:
  - Chain name entry to homonym of local entity.
  - Local entity becomes invisible.
  - Update name table.

Table entity (handled by scanner)

- All entities with given name point to same name.
- Entity in names table (dynamic cell) points to innermost occurrence of entity with given name.
- A names label
- A list of local entities declared in each scope.
- A tree of scopes: defining occurrences of functions, packages, blocks, loops, records.
- A set of variables: defining occurrences of variables.

Name Resolution and Block Structure

Name Resolution and Block Structure

Full information remains in the tree for subsequent passes.

Context and Import Rules:

- Block structure and hiding rules.
- Contextual rules for defining (use, occurrence) with the corresponding defining occurrence.

If entity is overloaded, associate entity with set of candidate entities, to be resolved by types and context.

Candidate entities: individual, type, context.

Complications:

- Context and import rules.
- Block structure and hiding rules.
- Names and labels.
To resolve A.B, first resolve A (direct name).

If A is enclosing scope, follow homonym chain for B

If A is a variable, find its type:
- if record or "struct", find component of type named B.
- if pointer, apply rule to designated type (implicit dereferencing).
- if task, find entry named B.

To resolve A.B.C, recurse: resolve prefix A.B, then apply previous rules.

To resolve A.B (C++) type of A must be of the form

*T. Proceed as above.