Syntax-Directed Semantics
Here we focus on syntax-directed approaches which can be based on computation of attributes.

- Enforce context-dependent language rules that are not reflected in the grammar, e.g., a function must have a return statement.
- Expand constructs in preparation for code generation, e.g., identify end of loops and procedures.
- Decorate AST with semantic information for subsequent code generation, e.g., determine type of expressions.
- Computation of semantics.
For every symbol in the grammar we define some computable properties (e.g., the value of a constant). The value 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).

For every production in the grammar we give computation rules (e.g., the value of a sum is the sum of the values of the operands). The rule is local: it only refers to other symbols in the same production. The syntax-directed framework.

Attributes and Attribute Grammars.
A systematic process of assigning meanings to programs can be viewed as computation of attributes associated with the non-terminals. Syntax-directed translations can be presented as syntax-directed rules of syntax-directed translations: syntax-directed translation can be presented as a conceptual view of syntax-directed semantics.

We consider two general approaches to the specification of syntax-direct translation: syntax-directed rules for semantic rules evaluation order graph dependency parse tree input stringing.
With each production \( A \rightarrow a \) we associate a set of semantic rules of the form \( f \) where

\[
(q^c, c_1, \ldots, c_k, c_{-1}, q, q^A) \ f = q^c \ A \ \{c_1, \ldots, c_k\} \cap a \ \{A\} \ 
\]

\( q \) is a synthesized attribute of one of the symbols and

\( q \) is an inherited attribute of one of the symbols or

\( q \) is a synthesized attribute of \( A \) and \( c_1, \ldots, c_k \) are attributes of the symbols in \( (q^c, c_1, \ldots, c_k) \ f = q^A \ A \ \{c_1, \ldots, c_k\}, c_{-1} \) where

\( f \) is a function and either

Syntax-Directed Definitions

A. Pnueli
These can be computed during bottom-up parsing.

<table>
<thead>
<tr>
<th>Production Rules</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>E ::= digit.lexval</td>
<td>E ::= digit</td>
</tr>
<tr>
<td></td>
<td>E ::= E E</td>
</tr>
<tr>
<td></td>
<td>E ::= T E</td>
</tr>
<tr>
<td></td>
<td>E ::= E [E]</td>
</tr>
<tr>
<td></td>
<td>E ::= (E)</td>
</tr>
<tr>
<td></td>
<td>E ::= E + T</td>
</tr>
<tr>
<td></td>
<td>E ::= E $</td>
</tr>
</tbody>
</table>

Simple desk calculator:

Following is an example of the semantic actions of a

Example of Synthesized Attributes
We can use this grammar in order to parse the following declaration of a list of variables:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addtypede(id, entry, T);</code></td>
<td><code>id L</code> ← <code>T</code></td>
</tr>
<tr>
<td><code>addtypede(id, entry, T);</code></td>
<td><code>T L</code> ← <code>id</code></td>
</tr>
<tr>
<td><code>L L1 = T</code></td>
<td><code>T L</code> ← <code>id</code></td>
</tr>
<tr>
<td><code>L type = real</code></td>
<td><code>L L</code> ← <code>L</code></td>
</tr>
<tr>
<td><code>L type = integer</code></td>
<td><code>L L</code> ← <code>L</code></td>
</tr>
<tr>
<td><code>L L = L type</code></td>
<td><code>D L L</code> ← <code>D</code></td>
</tr>
</tbody>
</table>

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

Examples of Inherited Attributes

```
real id1, id2, id3
```
Example Parse Tree

The parse tree for the declaration

is given by

real id_1, id_2, id_3
Given syntax-directed attribution rules, we can draw the dependency graph, indicating the dependency between attributes in the parse tree.

Dependency Graph

Lecture 5: Syntax-Directed Semantics
Syntax Trees as a Synthesized Attribute

The parse tree corresponding to a non-terminal can be viewed as a synthesized attribute. This is the way by which YACC/BISON is used in order to construct the parse tree.

### Production

- $E \rightarrow E_1 + T$
- $E \rightarrow E_1 - T$
- $E \rightarrow T$
- $E \rightarrow (E)$
- $T \rightarrow \text{id}$
- $T \rightarrow \text{NUM}$

### Semantic Rules

- $E.nptr := \text{mknode}('+', E_1.nptr, T.nptr)$
- $E.nptr := \text{mknode}('-', E_1.nptr, E.nptr)$
- $T.nptr := E.nptr$
- $T.nptr := \text{mkleaf}('id', \text{id}.entry)$
- $T.nptr := \text{mkleaf}('NUM', \text{NUM}.value)$
Bottom-Up Computation of Synthesized Attributes

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

<table>
<thead>
<tr>
<th>Production</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[1 - \text{dot}] \text{val} \equiv: \text{[dot]}\text{[val]} \text{val}$</td>
<td>\text{digit} \leftarrow \text{digit} \leftarrow \text{digit}</td>
</tr>
<tr>
<td>$\text{[dot]}\text{[val]} \times [2 - \text{dot}]\text{val} \equiv: \text{[dot]}\text{[val]} \text{val}$</td>
<td>$\emptyset \leftarrow \text{digit}$</td>
</tr>
<tr>
<td>$\text{[dot]}\text{[val]} + [2 - \text{dot}]\text{val} \equiv: \text{[dot]}\text{[val]} \text{[val]}$</td>
<td>$\text{digit} \times \text{digit} \leftarrow \text{digit} \times \text{digit}$</td>
</tr>
</tbody>
</table>

In parallel with the stack that contains the partially parsed sentential form (possibly represented by automaton states), we keep a stack of attribute values.

A. Pnueli
General Properties of Attribute Grammars

• Useful subclasses: S-attribute and L-attribute grammars.

  (symbol table)

• Attributes are computed by repeated passes over the
  AST.

  Attributes are more expressive than CFGs.

  Turing Machines. The finite-domain restriction is not
  powerful enough.

• In practice, many inherited attributes are handled by
  means of global data structures (symbol table).

• Attributes may be cyclic: checking whether
  an attribute grammar has cycles is decidable but
  potentially expensive.

• Attribute definitions may be cyclic: checking whether
  a useful subset: S-attribute and L-attribute grammars.

  Turing Machines. The finite-domain restriction is not
  powerful enough.
These are grammars that have only synthesized attributes. They are usable with bottom-up parsers, which can compute the attributes together with the parsing process.
Lecture 5: Syntax-Directed Semantics

A. Pnueli

L-attributed Grammars

Contains definitions which can compute attributes in a general depth-first traversal of the parse tree, according to the following schemes.

for each child m of n from left to right do

A syntax-directed definition is L-attributed if each inherited attribute depends only on

A equivalent definition is:

\[ [ \text{dfvisit}(n) ] \left[ \begin{array}{c}
\text{evaluate synthesized attributes of } \text{node}(n) \\
\text{evaluate inherited attributes of } m' \\
\text{foreach child } m \text{ of } n \text{ from left to right do}
\end{array} \right] \]

Each inherited attribute in a production of the form

\[ uX \cdots \bar{Z}X X \rightarrow A \]

depends only on the right-hand side of \( X \), and on the right-hand side of \( \bar{A} \) such that \( \bar{A} \rightarrow \bar{X} \) the attributes of the symbols \( X, X_1, X_2, \ldots, X_n \) to the left of \( X \) in \( \bar{A} \) can compute attributes of \( A \).
Typically, L-Grammars are appropriate for LL(1) parsers. They can be applied with 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.
Translation Schemes

These are grammar 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):

\[
\begin{align*}
\text{num} & \leftarrow T \\
\text{addop} & \leftarrow R \\
E & \leftarrow E \, \text{addop} \, T \\
T & \leftarrow T \, \text{num} \, \text{val}
\end{align*}
\]

The parse/action tree for the expression \( 9 - 5 + 2 \) is given by:

\[
\begin{align*}
E & \leftarrow E \, \text{print} \\
T & \leftarrow T \, \text{print} \\
R & \leftarrow R \, \text{print} \\
\end{align*}
\]
The following grammar uses a translation scheme to propagate type information along a declaration:

\[
\begin{align*}
D & \rightarrow T \{ \text{L.in} := T.\text{type} \} \\
T & \rightarrow \text{int} \{ T.\text{type} := \text{integer} \} \mid \text{real} \{ T.\text{type} := \text{real} \} \\
L & \rightarrow \{ \text{L.in} := \text{L.in} \} \text{L}, \text{id} \{ \text{addtype} (\text{id} . \text{entry}, \text{L.in}) \} \\
\end{align*}
\]

In spite of being an inherited scheme, this can still be applied within bottom-up parsing as follows:

This is based on the fact that, in all the reductions involving \( L \), we know that there exists a \( T \) below the handle in the stack.

\[
\begin{align*}
D & \rightarrow TL; \\
T & \rightarrow \text{int} \mid \text{real} \\
L & \rightarrow \text{id} \\
\end{align*}
\]
Replacing Inherited by Synthesized Attributes

Sometimes, it is possible to modify the grammar in a way that will transform an inherited attribute into a synthesized one. Consider the following grammar for Pascal type declaration:

\[
\begin{align*}
\{ \text{real} \} & \rightarrow \text{real} \\
\{ \text{integer} \} & \rightarrow \text{integer} \\
\{ \text{addtype}(\text{id}, \text{L}, s) \} & \rightarrow \text{id} \text{L} \\
(\{ \text{addtype}(\text{id}, \text{L}, s) \}) & \rightarrow \text{id} \text{L} \\
\end{align*}
\]

Propagating the type across the variable list is not even L-attributed.

Consider the following grammar for Pascal type declaration:

\[
\begin{align*}
\text{id} & \rightarrow \text{id} \\
\text{real} & \rightarrow \text{real} \\
\text{integer} & \rightarrow \text{integer} \\
\text{L} : \text{L} & \rightarrow \text{L} \\
\end{align*}
\]

Sometimes, it is possible to modify the grammar in a way that will transform an inherited attribute into a synthesized one. Consider the following grammar for Pascal type declaration:

\[
\begin{align*}
\{ \text{real} = : \text{real} \} & \rightarrow \text{real} \\
\{ \text{integer} = : \text{integer} \} & \rightarrow \text{integer} \\
\{ \text{addtype}(\text{id}, \text{L}, s) \} & \leftarrow \text{id} \text{L} \\
(\{ \text{addtype}(\text{id}, \text{L}, s) \}) & \leftarrow \text{id} \text{L} \\
\end{align*}
\]

Honors Compilers, NYU, Fall, 2009
Some Important Attributes

- For function: virtual functions (primitive operations).
  - private.

- For data/function members: visibility (public, protected).
  - scope.
  - entity (defining occurrence).

- For identifiers: candidate interpretations.
  - overloaded calls.

- For expressions: type.
  - etc, etc.


### Attribute Computation and Tree Traversals

- Inherited and synthesized values: Names, literals.
- Inherited attributes computed during declaration.
- Synthesized attributes on terminals.
- Generated code can be treated as synthesized expressions.
- Expression two passes required over.

Some systems employ left-to-right, top-down traversals, and Tree Traversals.

In the presence of overloading, type is both.

Localized multiple traversals.

Symbol table carries inherited information.

Honors Compilers, NYU, Fall, 2009

A. Pnueli

Lecture 5: Syntax-Directed Semantics
Name Resolution

- Context and import rules.
- Block structure and hiding rules.
- Complications:

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

Compute attribute entity: associate every identifier occurrence with the corresponding defining entity:

Compute entity: associate every identifier occurrence with the corresponding defining entity:

Honors Compilers, NYU, Fall, 2009
Name Resolution does not require any hashing

table entry (handled by scanner)

All identifiers with given name point to same names

Entry in names table (dynamically) points to innermost

A names table

A list of local entities declared in each scope.

A list of scopes: defining occurrences of functions,

A tree of scopes: reflecting scope nesting

Data structures reflect scope nesting

same name. Consistent with systematic renaming.

Basic Rule: Inner definition hides outer one with

Name Resolution and Block Structure

A Pnueli
Data Structures for Name Resolution

Entity chain, homonym chain, chars

 unbeameableamja

Name table
Semantic Actions for Visibility Processing

A. Pnueli

• Function, procedure, record, block.

Full information remains in the tree for subsequent passes.

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

On scope exit:
- Chain name entry to homonym of local entity.
- Local entity becomes invisible.
- Update names table.
- Full information remains in the tree for subsequent passes.

Honors Compilers, NYU, Fall, 2009
Resolving Qualified Names

To resolve A.B, first resolve A (direct name).

- If A is enclosing scope, follow homonym chain for B until we find a variable whose scope is A.
  - 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.
Top-Down Processing: All but Expressions

- Semantic analysis of expression.
- Analyze and resolve expression.
- Resolve type definition.
- Enter new entry into current scope.

Semantic analysis of object declaration:

- Enter new scope.
- Process visible declarations.
- Process private declarations.
- Process list of statements.

Semantic analysis of while statement:

- Exit scope.