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 complex constructs in preparation for code generation.
- Decorate AST with semantic information for subsequent code generation, e.g., determine type of expressions.
- Identity end of loops and procedures.

Computation of Semantics

A. Pnueli
Attributes and Attribute Grammars

Syntax-directed framework.

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

The evaluation of the attributes can require an arbitrary number of traversals of the AST: arbitrary context dependence.

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

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

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

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

The value of a sum is the sum of the values of the operands.

The evaluation of 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).
A systematic process of assigning meanings to programs can be viewed as computation of attributes associated with the non-terminals. We consider two general approaches to the specification of syntax-directed translations: syntax-directed rules and translation schemas. The conceptual view of syntax-directed translation can be presented as:

Input \rightarrow \text{Parse} \rightarrow \text{Parse Tree} \rightarrow \text{Dependency Graph} \rightarrow \text{Evaluation Order for Semantic Rules} \rightarrow \text{Syntax-Directed Translation}

A Syntax-Directed Translation

A Pnueli
A.Pnueli

Syntax-Directed Definitions

With each production \( A \rightarrow \alpha \) we associate a set of semantic rules of the form 

\[
\sigma^\alpha(q. c_1, \ldots, c_k) f = q\]  

where \( f \) is a function and either

- \( q \) is a synthesized attribute of one of the \( \alpha \) symbols and \( c_i \) are attributes of the symbols in \( \alpha \) and \( \sigma^\alpha \) is an inherited attribute of one of the symbols, or
- \( q \) is a synthesized attribute of \( A \) and \( c_i \), \( \ldots \), \( c_k \) are attributes of the symbols in \( \sigma^\alpha(q. c_1, c_{i+1}, \ldots, c_k, c_{i-1}) \), and \( q \) is an inherited attribute of one of the symbols.

Hence, \( \sigma^\alpha \) is a function.
These can be computed during bottom-up parsing:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E ::=$ digit. lexval</td>
<td>$\text{digit} \leftarrow E$</td>
</tr>
<tr>
<td>$E ::= E \cdot E$</td>
<td>$(E) \leftarrow E$</td>
</tr>
<tr>
<td>$E ::= \overline{E}$</td>
<td>$\overline{E} \leftarrow E$</td>
</tr>
<tr>
<td>$E ::= \overline{E} \cdot \overline{E}$</td>
<td>$\overline{E} \leftarrow \overline{E}$</td>
</tr>
<tr>
<td>$E ::= \overline{E}$</td>
<td>$\overline{E} \leftarrow E$</td>
</tr>
<tr>
<td>$E ::= E + E$</td>
<td>$E \leftarrow E + E$</td>
</tr>
<tr>
<td>$E ::= E + E$</td>
<td>$E \leftarrow E$</td>
</tr>
<tr>
<td>print(\text{val})</td>
<td>$\text{print(\text{val})} \leftarrow E$</td>
</tr>
<tr>
<td>$E ::= 1$</td>
<td>$1 \leftarrow 1$</td>
</tr>
<tr>
<td>$E ::= E + E \cdot E$</td>
<td>$E \leftarrow E + E \cdot E$</td>
</tr>
</tbody>
</table>

Example of synthesized attributes

Following is an example of the semantic actions of a simple desk calculator:
Examples 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>$L \rightarrow TL$</td>
<td>$L.tType := T.tType$</td>
</tr>
<tr>
<td>$T \rightarrow int$</td>
<td>$T.tType := integer$</td>
</tr>
<tr>
<td>$T \rightarrow real$</td>
<td>$T.tType := real$</td>
</tr>
<tr>
<td>$L \rightarrow L_1, id$</td>
<td>$L.ENTRY := L.ENTRY, L.tType$</td>
</tr>
<tr>
<td>$L \rightarrow id$</td>
<td>$addType(id.ENTRY, L.ENTRY)$</td>
</tr>
</tbody>
</table>

We can use this grammar in order to parse the declaration:

```
real id_1, id_2, id_3;
```
The parse tree for the declaration "real id1, id2, id3" is given by:

```
  (real)
  /     |
 (id2)   (id3)
   |       |
  (real)
  /     |
 (id1)   (id2)
   |       |
  (real)
```

Example Parse Tree
Given syntax-directed attribution rules, we can draw the dependency graph, indicating the dependency between attributes in the parse tree.
Syntax-Directed Semantics

A Pnueli

Syntax Trees as a Synthesized Attribute

The parse tree corresponding to a non-terminal can be viewed as a synthesized attribute. This is done by

The parse tree constructed by YACC/BISON is used in order to construct the parse tree. This is the way by

<table>
<thead>
<tr>
<th>Semantic Rules</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T \cdot \text{value} = \text{mkleaf}(\text{id}, \text{id}, \text{entry}) )</td>
<td>( \text{num} \leftarrow \text{T} )</td>
</tr>
<tr>
<td>( T \cdot \text{value} = \text{mkleaf}(\text{id}, \text{id}, \text{entry}) )</td>
<td>( \text{id} \leftarrow \text{T} )</td>
</tr>
<tr>
<td>( \text{E} \cdot \text{value} = \text{mknode}(\text{T}, \text{T}, \text{T}, \text{entry}) )</td>
<td>( (\text{E}) \leftarrow \text{T} )</td>
</tr>
<tr>
<td>( \text{E} \cdot \text{value} = \text{mknode}(\text{T}, \text{T}, \text{T}, \text{entry}) )</td>
<td>( \text{E} \leftarrow \text{T} )</td>
</tr>
<tr>
<td>( \text{T} \cdot \text{value} = \text{mknode}(\text{E} \cdot \text{value}, \text{E} \cdot \text{value}, \text{E} \cdot \text{value}) )</td>
<td>( \text{E} \leftarrow \text{E} \cdot \text{value} )</td>
</tr>
<tr>
<td>( \text{T} \cdot \text{value} = \text{mknode}(\text{E} \cdot \text{value}, \text{E} \cdot \text{value}, \text{E} \cdot \text{value}) )</td>
<td>( \text{E} \leftarrow \text{E} \cdot \text{value} )</td>
</tr>
<tr>
<td>( \text{T} \cdot \text{value} = \text{mknode}(\text{E} \cdot \text{value}, \text{E} \cdot \text{value}, \text{E} \cdot \text{value}) )</td>
<td>( \text{E} \leftarrow \text{E} \cdot \text{value} )</td>
</tr>
</tbody>
</table>

Lecture 5: Syntax-Directed Semantics
Bottom-Up Computation of Synthesized Attributes

A. Pnueli

### Bottom-Up Computation of Synthesized Attributes

In parallel with the stack that contains the partially parsed sentential form (interleaved with automaton states), we keep a stack of attribute values:

<table>
<thead>
<tr>
<th>push</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$Z$</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>$\lambda$</td>
</tr>
<tr>
<td>$X$</td>
<td>$X$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>

### Production Operations

- **digit**
  - $digit \leftarrow E$
  - $(E) \leftarrow P$
  - $P \leftarrow L$
  - $L \leftarrow T$
  - $T \leftarrow E$
  - $E \leftarrow \{E, \lambda\}$
  - $E \leftarrow \{\lambda\}$
  - $P \leftarrow \{E\}$

The operations of push and pop are then performed in conjunction for both stacks.
Useful subsets: $S$-attributed and $L$-attributed grammars.

- In practice, many inherited attributes are handled by means of global data structures (symbol table).
- Attributes are computed by repeated passes over the AST.
- Attributes are more expressive than CFGs.
- The finite-domain restriction is not more powerful than Turing Machines. The finite-domain restriction have the power of Turing Machines.

General Properties of Attribute Grammars

- Attributes are computed by repeated passes over the AST.
- Attributes are more expressive than CFGs.
- The finite-domain restriction is not more powerful than Turing Machines. The finite-domain restriction have the power of Turing Machines.

- In practice, many inherited attributes are handled by means of global data structures (symbol table).
- Attributes are computed by repeated passes over the AST.
- Attributes are more expressive than CFGs.
- The finite-domain restriction is not more powerful than Turing Machines. The finite-domain restriction have the power of Turing Machines.
S-Attributed Grammars

- These are grammars that have only synthesized attributed.
- They are usable with bottom-up parser, which can compute the attributes together with the parsing process.
A.L-attributed Grammars

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

```
procedure dfvisit(node : n);
foreach child m of n from left to right do
  evaluate inherited attributes of m;
  dfvisit(m);
```

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

\[ uX \cdots \alphaX_{-1}X \alphaX_1 \cdots \alphaX_n \leftarrow A \]

depends only on

\[ u, \alphaX_1 \cdots \alphaX_n \]

A syntactically-directed definition is L-attributed if each inherited attribute

\[ uX \cdots \alphaX_{-1}X \alphaX_1 \cdots \alphaX_n \leftarrow A \]

evaluates synthesized attributes of \( A \).

The inherited attributes of \( A \) depend only on

\[ \alphaX_1 \cdots \alphaX_n \]

in the right-hand side of \( A \), \( \alphaX_1 \) to the left of \( \alphaX_n \) in

\[ uX \cdots \alphaX_1X \alphaX_2 \cdots \alphaX_n \leftarrow A \]

An equivalent definition is:

```plaintext
procedure dfvisit(node : n);
for each child m of n from left to right do
  evaluate inherited attributes of m;
  dfvisit(m);
```

\[ uX \cdots \alphaX_{-1}X \alphaX_1 \cdots \alphaX_n \leftarrow A \]
Typically, L-grammars are appropriate for LL(1) parsers. The 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 while performing LALR parsing. There is a general transformation which converts an arbitrary L-grammar into an S-grammar, but it is often clumsy.

Typically, L-grammars are appropriate for LL(1)
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):

```
num <- T
addop <- R
E <- E
R <- T
+ <- print(num.val)
- <- print(num.val)
*
/ <- print(num.val)
```
This is based on the fact that, in all the reductions involving `T`, we know that there exists a `T` below the handle in the stack.

The following grammar uses a translation scheme to propagate type information along a declaration:

<table>
<thead>
<tr>
<th>Production</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>D</code></td>
<td><code>addtype(id:entry;L)</code></td>
</tr>
<tr>
<td><code>T</code></td>
<td><code>int val[ntop]</code></td>
</tr>
<tr>
<td><code>T</code></td>
<td><code>real val[ntop]</code></td>
</tr>
<tr>
<td><code>L</code></td>
<td><code>id addtype(val[top];val[top+1])</code></td>
</tr>
</tbody>
</table>

This is based on the fact that, in all the reduction involving `L`, we know that there exists a `T` below the handle in the stack.
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 declarations:

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

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

Similarly, we can make C-type declarations amenable for bottom-up attribute computation, we can make C-type declarations amenable for bottom-up attribute computation.

Consider the following grammar for Pascal type declarations:

\[
\begin{align*}
L &::= \text{id} \mid \text{real} \mid \text{integer} \\
L &::= \text{L} \cdot T \mid \text{id} \cdot L \mid \text{addtype}(\text{id}, \text{L} \cdot T) \cdot T \\
\end{align*}
\]

However, we can modify the grammar to make the type a synthesized attribute.

Sometimes, it is possible to modify the grammar in a way that will transform an inherited attribute into a synthesized one.

Replacing Inherited by Synthesized Attributes
Some Important Attributes

- For function: virtual functions (primitive operations).
- For data/variable members: visibility (public, protected, private).
- For definitions: scope.
- For identifiers: entity (defining-occurrence).
- For overloaded calls: candidate interpretations.
- For expressions: type.

• Etc., etc.
Some systems employ left-to-right, top-down traversals. Inherited attributes and synthesized: two passes required over.

Inherited and inherited attributes, type is both.

In the presence of overloading, type is both.

Synthesized attributes on terminals: names, literals, values.

Inherited attributes computed during declaration.

As one global data structure, symbol table carries inherited information.

Generated code can be treated as synthesized. 

Attribute Computation and Tree Traversals. 

A. Pnueli

Honors Compilers, NYU, Spring, 2007.
Name Resolution

- Context and Import rules.
- Block structure and hiding rules.
- Complications:
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 set of scopes: defining occurrences of functions,

• Data structures reflect scope nesting

• Same name. Consistent with systematic renaming.

Basic Rule: Inner definition hides outer one with
Data Structures for Name Resolution

Entity chain, homonym chain, chars

A. Pnueli

Honors Compilers, NYU, Spring 2007
A scope is any entity that may have local declarations.

Semantic Actions for Visibility Processing

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

On scope exit:
- Chain name entry to homonym of local.
- Local entity becomes invisible.
- Update names table.

For every entity in the list of local entities:
- Update name. Local entity becomes invisible.
- Update names table.

Full information remains in the tree for subsequent passes.
To resolve A.B, first resolve A (direct name).

- If A is a variable, find its type:
  - If A is a variable whose scope is B, follow homonym chain for B.
  - If A is enclosing scope, follow homonym chain for B.
  - If A is a record or “struct”, find component of type named B.
  - If A is a pointer, apply rule to designated type (implicit dereferencing).
  - If A is a variable, find its type:
    - If task, find entry named B.
    - If record or “struct”, find component of type named B.
    - If record or “struct”, find component of type named B.

- To resolve A.B.C, recurse: resolve prefix A.B, then homonym chain for B.

- To resolve A.B.C’, recurse: resolve prefix A.B, then homonym chain for B.

- To resolve A.B (C++) typeof A must be of the form *T. Proceed as above.

* To resolve A.B (C++) type of A must be of the form *T. Proceed as above.

A. Pnueli
Top-Down Processing: All but Expressions

- Semantic analysis of package declaration:
  - Enter new scope
- Resolution of visible declarations:
  - Process visible declarations
- Process list of statements:
  - Process (analyze and resolve) condition
- Process while statement:
  - Process list of statements
  - Process (analyze and resolve) condition
- Semantic analysis of object declaration:
  - Enter new entry into current scope
  - Resolve type definition
  - Analyze and resolve expression

- Semantic analysis of package declaration:
  - Enter new scope
- Resolution of private declarations:
  - Process private declarations
- Resolution of visible declarations:
  - Process visible declarations
- Exit scope
- Resolve type definition
- Analyze and resolve expression

- Semantic analysis of package declaration:
  - Enter new scope
- Resolution of visible declarations:
  - Process visible declarations
- Process list of statements:
  - Process (analyze and resolve) condition
- Process while statement:
  - Process list of statements
  - Process (analyze and resolve) condition
- Semantic analysis of object declaration:
  - Enter new entry into current scope
  - Resolve type definition
  - Analyze and resolve expression