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

## Syntax-Directed Translation

A systematic process of assigning meanings to programs can be viewed as computation of attributes associated with the non-terminals. A systematic framework for semantic rules can be based on computation of attributes.

### Syntax-Directed Translation

- 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, e.g., determine type of expressions.
- Generate code in preparation for code generation, e.g., identity end of loops and procedures.
- For every symbol in the grammar, we define some computable properties (e.g., the value of a constant expression found in the constant declaration).
- The value of a declared constant is found in the constant declaration of the AST, arbitrary context dependence.
- Syntax-directed framework.

### Attributes and Attribute Grammars

- For every production in the grammar, we give computation rules for the properties of symbols on both sides of the production.
- The evaluation of the attributes can require an arbitrary number of traversals of the AST.
- The rule is local: it only refers to other symbols in the same production.
- The value of a sum is the sum of the values of the operands.
- The value of a constant expression is found in the constant declaration.
- The value of a function is found in the function declaration.

### Computation of Semantics

- Input parse dependency graph.
- Initial parse.
- String free evaluation order.
- Parse dependency graph.
- Syntax-directed translation can be presented as syntax and translation schemes.
- The conceptual view of syntax-directed translation: syntax-directed rules are associated with the non-terminals.
- Syntax-directed approaches to the specification of syntax-directed translations.
- Syntax-directed rules and translation schemes.
- The conceptual view of syntax-directed translation.
We can use this grammar in order to parse the type declaration of a list of variables:

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$ ! $TL$</td>
<td>$L:in = T:type$</td>
</tr>
<tr>
<td>$T$ ! $int$</td>
<td>$T:type = integer$</td>
</tr>
<tr>
<td>$T$ ! $real$</td>
<td>$T:type = real$</td>
</tr>
<tr>
<td>$L$ ! $1 \cdot id \cdot 1$</td>
<td>$L:in = \text{addtype}(id:entry;L:in)$</td>
</tr>
</tbody>
</table>

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>$D$ ! $1\cdot id \cdot 1$</td>
<td>$D:T:type = {A \cap (q_1, \ldots, q_n) \cdot f = q_1, \ldots, q_n \cdot f}$</td>
</tr>
</tbody>
</table>

$q$ is an inherited attribute of one of the attributes of the symbols and

$q$ is a synthesized attribute of $A$ and $c_1, \ldots, c_n$ are attributes of the symbols in $A$.

$q$ is a function and either

$f$ is a function defined by a set of semantic rules of the form $\forall c_1, \ldots, c_n. q \cdot f = q_1, \ldots, q_n \cdot f$.
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.

### 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.

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

### General Properties of Attribute Grammars

- Attribute grammars have the power of Turing Machines. The finite-domain restriction is not more expressive than CFG’s.
- Attributes are computed by repeated passes over the AST.
- Attribute definitions may be cyclic: checking whether an attribute grammar has cycles is decidable but potentially expensive.
- In practice, many inherited attributes are handled by means of global data structures (symbol table).
- Useful subsets: S-attributed and L-attributed grammars.

### Syntax-Directed Semantics

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

- Dependency Graph
- Syntax-Directed AttributionRules
- Semantic Rules
- Production
- Bottom-Up Computation of Synthesized Attributes
- General Properties of Attribute Grammars
A. Pnueli

L-Attributed Grammars

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.

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

```
\[ E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\}) \]
```

Following is an example of a translation scheme which transforms infix expressions into Polish expressions (under LL(1) parsing):

```
E \rightarrow TRR
T \rightarrow numf \rightarrow print(\{num:val\})
```

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

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) parsers. They can be applied with LALR parsers as in some special cases (as shown below), it is possible to compute L-attributes while performing LALR parsing.

In independent pass, after the syntax tree has been constructed.

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.

L-Attributed Grammars

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

```
E \rightarrow TRR \rightarrow addop Tf \rightarrow print(\{addop:lexeme\})
  \[ \rightarrow T \rightarrow numf \rightarrow print(\{num:val\}) \]
  \[ \rightarrow + \rightarrow print(\{0\}) \]
  \[ \rightarrow 2 \rightarrow print(\{0\}) \]
```

Computation of L-Attributes

The inherited attributes of a production, and the attributes of the symbols $X_1, X_2, \ldots, X_n$ to the left of $X_j$ depend only on $X_1, X_2, \ldots, X_{j-1}$ on the right-hand side of $A$.

A syntax-directed definition is an L-attribute if each inherited attribute depends only on the leftmost child of a node.

A general transformation scheme involves evaluating synthesized attributes of $n$ for each child of $n$ from left to right.
The following grammar uses a translation scheme to propagate type information:

```plaintext
D ! T f L : in T := T : type
L ! T ! int f T : type := integer
L ! T ! real f T : type := real
L ! f L 1 : in L : in T := L : in addtype (id : entry; L : in)
```

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

```plaintext
Production Operations
D ! T L ; T ! int val [ntop] := integer
T ! real val [ntop] := real
L ! L ; id addtype (val [top]; val [top + 3])
L ! id addtype (val [top]; val [top + 1])
```

Some systems employ left-to-right, top-down traversals, while others use left-to-right, top-down and bottom-up traversals. Attributes can be computed as follows:

- **Inherited** attributes computed during declaration processing, symbol table carries inherited information as one global data structure.
- **Synthesized** attributes on terminals: names, literal values.
- **Attributes** on non-terminals: type.
- **Expressions**:
  - For function: virtual functions (primitive operations).
  - For overloading: candidate interpretations.
  - For definitions: visibility (public, protected, private).
  - For data/function members: visibility (public, protected, private).
  - For identifiers: entity (defining occurrence).
  - For expressions: type.
  - Etc.

Replacing Inherited by Synthesized Attributes

Consider the following grammar for Pascal type declarations:

```plaintext
D ! L : T
L ! L : in L : in T := L : in addtype (id : entry; L : in)
```

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.

```
{ {id} } { {id} } { {id} } { {id} } { {id} } { {id} } { {id} } { {id} } { {id} } { {id} } { {id} } { {id} }
```

Some Important Attributes

- For expressions:
  - type.
  - values.
  - names, literal values.
  - candidate interpretations.
  - entity (defining occurrence).
  - scope.
  - visibility (public, protected, private).
  - virtual functions (primitive operations).
  - etc.

Attribute Computation and Tree Traversals

Some systems employ left-to-right, top-down traversals, with localized multiple traversals.

- Inherited attributes computed during declaration processing, symbol table carries inherited information as one global data structure.
- Synthesized attributes on terminals: names, literal values.
- In the presence of overloading, type is both inherited and synthesized: two passes required over expressions.
- Inherited attributes on non-terminals: type.
- Inherited attributes compute during declaration, with localized multiple traversals.
- Some systems employ left-to-right, top-down, and bottom-up traversals.
### Name Resolution

**Basic rule:** inner definition hides outer one with same name. Consistent with systematic renaming.

Data structures reflect scope nesting:
- A tree of scopes: defining occurrences of functions, packages, blocks, loops, records.
- A list of local entities declared in each scope.
- A names table: A list of names.
- A names table: Entry in names table (dynamically) points to innermost occurrences of entity with given name.
- All identifiers with given name point to same names table entry (handled by scanner).

Name resolution does not require any hashing.

### Semantic Actions for Visibility Processing

- A scope is any entity that may have local declarations: function, procedure, record, block.
- 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 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.

### Data Structures for Name Resolution

- **Entity chain:** homonym chain, chars
- **Name chain:** homonym chain, chars
- **Name table:** homonym chain, chars

---

**Complications:**
- Block structure and hiding rules.
- Context and import rules.
- Entity is overloaded, associate entity with set of candidate entities, to be resolved by types and context.
- Compute attribute `occurrence` with the corresponding defining occurrence.

---

**Name Resolution**

- Compute attribute `entity`: associate every identifier (use, occurrence) with the corresponding defining occurrence.
- Block structure and hiding rules.
- Entity is overloaded, associate entity with set of candidate entities, to be resolved by types and context.
Resolving Qualified Names

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

To resolve A.B.C, recursively resolve A.B, then recur for C.

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

apply previous rules.
semantics analysis of while statement.
exit scope.

To resolve A..B, recursive resolve prefx A..B, then
if task, find entry named B.

if pointer, apply rule to designated type (implicit
decrement).

if record or "struct", find component of type named B.

if record, find variable whose scope is A.

until we find a variable whose scope chain for B

Top-Down Processing: All but Expressions

Semantic analysis of package declaration:

Semantic analysis of visible declarations:

Semantic analysis of object declaration:

Enter new scope.

Process (analyze and resolve) visible declarations.

Resolving Qualified Names

Enter new scope.

Process visible declarations.

Process (analyze and resolve) condition.

Process list of statements.

To resolve A.B.C, recursively resolve A.B, then

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

apply previous rules.
semantics analysis of while statement.
exit scope.

To resolve A..B, recursive resolve prefx A..B, then
if task, find entry named B.

if pointer, apply rule to designated type (implicit
decrement).

if record or "struct", find component of type named B.

if record, find variable whose scope is A.

until we find a variable whose scope chain for B

Top-Down Processing: All but Expressions

Semantic analysis of package declaration:

Semantic analysis of visible declarations:

Semantic analysis of object declaration:

Enter new scope.