Syntactic Directed Translation

Translation Schemes
Syntax-Directed Translations

• Rule-based Evaluation & Syntax-Directed Translation
  – Semantic Actions are Embedded in Productions

• Evaluation Order and Parsing
  – Top-Down: Associate Actions with Production Function Code
  – Bottom-Up: Execute Actions on a Reduction

• Single Pass Translation Schemes – Fast!
  – No need to Construct the Parse Tree and then do a Topological Sorting to Find out Feasible Order for Evaluation of

• Issues:
  – Dealing with Embedded Actions
    • May Require Inserting Additional Symbols, Markers
  – Dealing with Inherited Attributes in Bottom-Up Parsing
    • Reach into the Stack for Value of Attribute
    • Position Independence in the Stack
Example: Building an AST

- Assume Constructors for Each Node (e.g., malloc in C)
- Assume Stack holds Pointers to Nodes & Symbols
- Assume YACC Syntax with Refs to Stack Depth

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal ( \rightarrow ) Expr</td>
<td>$$ = $1;</td>
</tr>
<tr>
<td>Expr ( \rightarrow ) Expr + Term</td>
<td>$$ = MakeAddNode($1,$3);</td>
</tr>
<tr>
<td></td>
<td>Expr – Term</td>
</tr>
<tr>
<td></td>
<td>Term</td>
</tr>
<tr>
<td>Term ( \rightarrow ) Term * Factor</td>
<td>$$ = MakeMulNode($1,$3);</td>
</tr>
<tr>
<td></td>
<td>Term / Factor</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
</tr>
<tr>
<td>Factor ( \rightarrow ) ( Expr )</td>
<td>$$ = $2;</td>
</tr>
<tr>
<td></td>
<td>number</td>
</tr>
<tr>
<td></td>
<td>id</td>
</tr>
</tbody>
</table>

Semantic Rules Executed on Reduction of corresponding Production During Parsing
Example — Building an AST

Stack
(Symbols & Symbols)

Input

Parser Action

[Diagram showing a stack with symbols and symbols, and an input sequence: ( num + num ) * num]
Example — Building an AST

```
((num + num) *)
```

*Shift:* push '(' onto stack
Example — Building an AST

\[
\begin{array}{c}
( \\
\quad \cdot \\
\quad \text{num} \\
\quad \cdot \\
\quad ( \\
\end{array}
\]

\text{Shift: push ‘num’ onto stack}
Example — Building an AST

Reduce: Factor → num

$\$$=\text{MakeNumNode}(\$1.\text{token})
Example — Building an AST

\[ ( \text{num} \cdot \text{num} \cdot \text{num} + \text{num} ) \]

Reduce: \( \text{Term} \to \text{Factor} \quad \$$ = \$1 \]
Example — Building an AST

\[
(\text{Expr} \rightarrow \text{Term})
\]

\[
(\text{num} + \text{num}) \cdot \text{num}
\]

\[\text{Reduce: Expr} \rightarrow \text{Term} \quad \$$ = 1\]
Example — Building an AST

```
( [num] + [num] ) * [num]
```

Shift: push ‘+’ onto stack
Example — Building an AST

$((\text{num} + \text{num}) \times \text{num})$

$\text{numNode}$

$\text{num}$

$\text{Expr}$

$\text{Term}$

$\text{Factor}$

$\text{num}$

$\text{Shift: push ‘num’ onto stack}$
Example — Building an AST

Reduce: \( \text{Factor} \to \text{num} \)  \( \$\$ = \text{MakeNumNode}($1$.token) \)
Example — Building an AST

Reduce: Term → Factor

$$ = 1$$
Example — Building an AST

Example expression: 

\((\text{num} + \text{num}) \times \text{num}\)

AST representation: 

```
numNode
  num
  numNode
    Term
      +
        Expr
          Term
            Factor
              num
```

```plaintext
( num + num ) * num
```
Example — Building an AST

\[
\text{Reduce: } \quad \text{Expr} \rightarrow \text{Expr} + \text{Term} \quad $$ = \text{MakeAddNode}($1,$3)
\]
Example — Building an AST

\[(\text{num} \times \text{num}) + \text{numNode} + \text{numNode}]\]

**Shift:** push ‘)’ onto stack

- **Expr**: 
  - **Term**: 
    - **Factor**: `num`
  - `+` 
- **Expr**: 
  - `)`
- **add Node**: 
  - `numNode` 
  - `num` 
  - `numNode` 
  - `num`
Example — Building an AST

```
Expr
  +
Term
  Factor
    num
```

```
( num + num ) * num
```
Example — Building an AST

Reduce: $\text{Expr} \rightarrow (\text{Expr})$

$$ = \$2$$
Example — Building an AST

```
num

add Node
```

```
num

numNode

num

numNode

num
```

```
Expr

Term

Factor

num

num
```

```
( num + num )
```

**Reduce:** Term $\rightarrow$ Factor

$$ = 1$$
**Example — Building an AST**

```
( num + num ) * num
```

**Reduce**: Reduce → Term

```
Expr
Term
Factor
( Expr )
Expr + Term
Factor
num
```

Reduce → Term

```
$$ = 1$
```
Example — Building an AST

```
num num
num num
```

```
Expr
 add Node
 numNode
 num
```

```
Expr
 +
 Term
 Factor
 num
```

```
Term
 *
 Expr
 Expr
 num
```

Shift: push ‘*’ onto stack
Example — Building an AST

\[
\begin{align*}
  & ( \text{num} + \text{num} ) \\
\end{align*}
\]

\[
\begin{align*}
  & \text{num} \\
\end{align*}
\]

\[
\begin{align*}
  & \text{Expr} \quad \text{Factor} \\
  & \text{Term} \\
  & * \\
\end{align*}
\]

\[
\begin{align*}
  & \text{add Node} \\
\end{align*}
\]

\[
\begin{align*}
  & \text{numNode} \quad \text{numNode} \\
  & \text{num} \\
  & \text{num} \\
\end{align*}
\]
Example — Building an AST

Reduce: Factor → num

$\$ = \text{MakeNumNode}($1$.\text{token}$)
Example — Building an AST

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Term} \times \text{Factor} \\
\text{Term} & \rightarrow \text{Factor} + \text{Term} \\
\text{Factor} & \rightarrow \text{numNode}
\end{align*}
\]

\[
(\text{numNode} \times \text{numNode}) \times \text{numNode}
\]

Reduce: Term → Factor

\[
\text{expr} = 1
\]
Example — Building an AST

```
( num + num ) * num
```

Reduce: $\text{Expr} \rightarrow \text{Expr} \ast \text{Term}$

$$ = \text{MakeMultNode}(\$1,\$3)$$
Example: Calculator Revisited

• Use Parser Stack for Symbols and Attribute Values
  – As Parser Algorithm Places Symbols
  – Semantic Actions use Stack locations for Storing Values
    • Reach Into Stack for Values of Symbols
    • When Reduction, Evaluate Semantic Action and Store Value of Production LHS

Expr₀ \rightarrow Expr₁ * Term \ \{ \ Expr₀.val = Expr₁.val \times Term.val \}
### Example: Calculator Revisited

<table>
<thead>
<tr>
<th>Production</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Line</code> $\rightarrow$ <code>Expr</code></td>
<td><code>{ print(Expr.val); }</code></td>
</tr>
<tr>
<td><code>Expr_0</code> $\rightarrow$ <code>Expr_0</code> + <code>Term</code></td>
<td><code>{ Expr_0.val = Expr_1.val + Term.val; }</code></td>
</tr>
<tr>
<td><code>Term_0</code> $\rightarrow$ <code>Term_1</code> * <code>Factor</code></td>
<td><code>{ Term_0.val = Term_1.val * Factor.val; }</code></td>
</tr>
<tr>
<td><code>Factor</code> $\rightarrow$ <code>( Expr )</code></td>
<td><code>{ Factor.val = Expr.val; }</code></td>
</tr>
<tr>
<td><code>digit</code></td>
<td><code>{ Factor.val = digit.lexval; }</code></td>
</tr>
</tbody>
</table>
# Example: Calculator Revisited

<table>
<thead>
<tr>
<th>Production</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Line</strong> → <strong>Expr</strong></td>
<td>{ print(Expr.val); }</td>
</tr>
<tr>
<td><strong>Expr</strong>₀ → <strong>Expr</strong>₀ + <strong>Term</strong></td>
<td>{ Expr₀.val = Expr₁.val + Term.val; }</td>
</tr>
<tr>
<td></td>
<td>{ Expr₀.val = Term.val; }</td>
</tr>
<tr>
<td><strong>Term</strong>₀ → <strong>Term</strong>₁ * <strong>Factor</strong></td>
<td>{ Term₀.val = Term₁.val * Factor.val; }</td>
</tr>
<tr>
<td></td>
<td>{ Term₀.val = Factor.val; }</td>
</tr>
<tr>
<td><strong>Factor</strong> → ( <strong>Expr</strong> )</td>
<td>{ Factor.val = Expr.val; }</td>
</tr>
<tr>
<td></td>
<td>{ Factor.val = digit.lexval; }</td>
</tr>
</tbody>
</table>

### Production Actions

<table>
<thead>
<tr>
<th>Production</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Line</strong> → <strong>Expr</strong></td>
<td>{ print(stack[top].val); }</td>
</tr>
<tr>
<td><strong>Expr</strong>₀ → <strong>Expr</strong>₀ + <strong>Term</strong></td>
<td>{ stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }</td>
</tr>
<tr>
<td></td>
<td>{ }</td>
</tr>
<tr>
<td><strong>Term</strong>₀ → <strong>Term</strong>₁ * <strong>Factor</strong></td>
<td>{ stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }</td>
</tr>
<tr>
<td></td>
<td>{ }</td>
</tr>
<tr>
<td><strong>Factor</strong> → ( <strong>Expr</strong> )</td>
<td>{ stack[top-2].val = stack[top-1].val; top = top - 2; }</td>
</tr>
<tr>
<td></td>
<td>{ stack[top].val = digit.lexval; }</td>
</tr>
</tbody>
</table>
Translation Schemes

• L-Attributed Syntax-Directed Definition where:
  – Embed Semantic Actions in \{ \} 
  – Positioned Between Symbols of Production 
  – Useful for Specifying Translation During Parsing

• Attribute Values Must be Available when Actions Refer to it
  1. Inherited attributes for a symbol on the RHS of a production must be computed in an action before that Symbol.
  2. An action must not refer to a synthesized attribute of a symbol to the right of the action.
  3. A synthesized attribute for the symbol on the RHS of a production can only be computed after all attributes it references have been computed.

• Easily Implemented in Bottom-Up Parsers
L-Attributed Definitions

• A Syntax-Directed Definition is L-Attributed if
  – Each inherited Attribute $X_j, 1 \leq j \leq n$, for $A \rightarrow X_1X_2,\ldots,X_n$ depends only on:
    • The Attributes of the Symbols $X_1, X_2,\ldots,X_{j-1}$ to the left of $X_j$
    • The Inherited Attributes of $A$

• Values Flow from Left-to-Right in the Parse Tree
• Still an S-Attributed Definition
  – Restrictions are for Inherited Attributes Only.
• Can be Evaluated in a Depth-first Traversal
• In Many Cases even in a Single Pass
  – Either as you parse using an LL parsing algorithm
  – Or after building the tree and traversing it…
L-Attributed Def.: Evaluation Order

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

• What to do When?
  – Embedded Actions
  – Inherited Attributes
  – Replacing Inherited Attributes by Synthesized Attributes
L-Attributed Definition while Parsing

Algorithm
LL(1) L-Attributed Evaluation

FOR each predicted production $X_0 \rightarrow X_1X_2...X_n$.
Push $X_0$'s inherited attributes onto semantic stack.
Push $X_1$'s inherited attributes onto semantic stack.
Parse $X_1$, then push $X_1$'s synthesized attributes onto
semantic stack.
Push $X_2$'s inherited attributes onto semantic stack.
Parse $X_2$, then push $X_2$'s synthesized attributes onto
semantic stack ...
Push $X_n$'s inherited attributes onto semantic stack.
Parse $X_n$, then push $X_n$'s synthesized attributes onto
semantic stack.
Pop attributes of $X_1X_2,...,X_n$.
Push synthesized attributes of $X_0$.
ENDFOR
Embedded Actions

• Actions are Executed when Parser Reduces a Production
  – After reductions for the RHS have occurred
  – Values for the Symbols available on the stack

• What to do with Embedded Actions?
  – $A \rightarrow X \{ \text{action} \} Y Z$
  – The action should execute before the actions for $Y$ and $Z$
Embedded Actions

• Actions are Executed when Parser Reduces a Production
  – After reductions for the RHS have occurred
  – Values for the Symbols available on the stack

• What to do with Embedded Actions?
  – $A \rightarrow X \{ \text{action} \} Y Z$
  – The action should execute before the actions for $Y$ and $Z$

• Transform the Grammar adding a Marker Symbol using an empty RHS production for Marker
  – $A \rightarrow X M Y Z$
  – $M \rightarrow \varepsilon \{ \text{action} \}$
Inherited Attributes

• For the production $A \rightarrow X Y$ when the parser reduces $X$’s production, its attributes will be on the top of the stack
  – If $Y$ uses synthesized attributes $X.s$ from $X$ just needs to copy value from the top of the stack into computation of $Y$’s attributes

• **Observation:** Reaching into the Stack Works…
  – If you know the position of the symbol’s attributes
  – Look at the corresponding grammar’s production

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow aAC$</td>
<td>$C.i = A.s$</td>
</tr>
<tr>
<td>$S \rightarrow bABC$</td>
<td>$C.i = A.s$</td>
</tr>
<tr>
<td>$C \rightarrow c$</td>
<td>$C.s = \text{func}(C.i)$</td>
</tr>
</tbody>
</table>

• **Problem:**
  – There maybe a $B$ symbol between $A$ and $C$ and thus the relative position of the synthesized attribute $A.s$ on the stack is not known to compute $C.i$.

• **Solution?**
Inherited Attributes

• Solution:
  – Insert a Marker Symbol just before C in one of the productions
  – Use M to copy inherited to synthesize attributes
  – Redo Actions to use M’s synthesize attributes

Production
S → aAC
S → bABMC
C → c
M → ε

Semantic Rule
C.i = A.s
M.i = A.s; C.i = M.s
C.s = func(C.i)
M.s = M.i
Replacing Inherited with Synthesized Attrib.

• Inherited Attributes are Hard to Handle
• Alternative: Modify Grammar (whenever possible)
• Example:
  – List of Declaration in Pascal.
  – Type as an Inherited Attribute
  – Change Grammar and Type is Synthesized

Decl → List ‘:’ Type
Type → integer | real
List → List ‘,’ id | id

Decl → id List
List → ‘,’ id List | ‘:’ Type
Type → integer | real
Replacing Inherited with Synthesized Attrib.

Decl → List ‘:’ Type
Type → integer | real
List → List ‘,’ id | id

Decl → id List
List → ‘,’ id List | ‘:’ Type
Type → integer | real
Example

• What Does This Scheme Do?

Expr → Term Factor₁
Factor₀ → addOp Term { print(addOp.lexeme) } Factor₁
      | ε
Term → num { print(num.val) }

Input: 9 - 5 + 2

```
Expr
  └── Term
      ├── Factor
      │    └── addOp Term { print(addOp.lexeme) } Factor₁
      │         | ε
      │    └── Term
      │          │ 9
      │          └── Factor
      │             └── Term
      │                 └── Factor
      │                     └── Term
      │                           2
      └── Factor
            └── Term
                  5
```

Expr → Term Factor₁
Factor₀ → addOp Term { print(addOp.lexeme) } Factor₁
      | ε
Term → num { print(num.val) }

Input: 9 - 5 + 2

```
Expr
  └── Term
      ├── Factor
      │    └── addOp Term { print(addOp.lexeme) } Factor₁
      │         | ε
      │    └── Term
      │          │ 9
      │          └── Factor
      │             └── Term
      │                 └── Factor
      │                     └── Term
      │                           2
      └── Factor
            └── Term
                  5
```
Example

• What Does This Scheme Do?

Expr → Term Factor₁

Factor₀ → addOp Term { print(addOp.lexeme) } Factor₁

| ε

Term → num { print(num.val) }

Input: 9 - 5 + 2

Expr

Term

9

{ print('9') }

Factor

-

Term

5

{ print('5') }

Factor

+

Term

2

{ print('2') }

Factor

ε
Example

• What Does This Scheme Do?

Input: 9 - 5 + 2
Output: 9 5 - 2 +

depth-first evaluation order
Summary

• Attribute Grammar
  – Augment CFG with Attributes and Rules
  – Inherited and Synthesized Attributes
  – Find Dependence Graph and Evaluation Order
  – Useful for Semantic Analysis

• Important Class: L-Attributed Grammar
  – Information moves from Left-to-Right
  – Inherited Attributes and Embedded Actions can be Resolved
  – Semantic Actions Executed upon Production Reduce Operations
  – Can be Evaluated Bottom-Up in a Single Pass