Syntactic Directed Translation

Translation Schemes

Copyright 2016, Pedro C. Diniz, all rights reserved. Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.
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

| Goal   | Expr          | $\$ = $1;
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>$\Rightarrow$</td>
<td>$$ \text{MakeAddNode}(1,3)$$;</td>
</tr>
<tr>
<td></td>
<td>$\Rightarrow$</td>
<td>$$ \text{MakeSubNode}(1,3)$$;</td>
</tr>
<tr>
<td></td>
<td>$\Rightarrow$</td>
<td>$$ \text{MakeMulNode}(1,3)$$;</td>
</tr>
<tr>
<td></td>
<td>$\Rightarrow$</td>
<td>$$ \text{MakeDivNode}(1,3)$$;</td>
</tr>
<tr>
<td>Term</td>
<td>$\Rightarrow$</td>
<td>$$ \text{MakeNumNode}(1)$$;</td>
</tr>
<tr>
<td></td>
<td>$\Rightarrow$</td>
<td>$$ \text{MakeIdNode}(1)$$;</td>
</tr>
<tr>
<td>Factor</td>
<td>$\Rightarrow$</td>
<td>$$ \text{MakeIdNode}(1)$$;</td>
</tr>
</tbody>
</table>

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

Stack

(Symbols & Symbols)

Parser Action

Input

Input

Parser Action

(Symbols & Symbols)

num + num ) * num

Input
Example — Building an AST

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

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

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

Shift: push ‘num’ onto stack

\[
( \text{num} + \text{num} ) * \text{num}
\]
Example — Building an AST

\[
(\text{Factor} \cdot \text{Factor})
\]

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

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

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

Reduce: Expr → Term  $$ = 1$$
Example — Building an AST

Example: Building an AST

```
numNode
  num

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

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

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

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

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

Reduce: Factor → num

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

Reduce: Term → Factor

$$ = 1$$
Example — Building an AST

( num + num ) * num

num Node
num

num Node
num

Term
+

Expr

Factor
num

Expr
Term
Factor
num
Example — Building an AST

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

Example expression: \((\text{num} + \text{num}) \times \text{num}\)

Stmt: Push `)` onto stack
Example — Building an AST
Example — Building an AST

Reduce: $Expr \rightarrow ( Expr )$  \hspace{1cm} $$ = 2$
Example — Building an AST

Reduce: Term $\rightarrow$ Factor

$$ = 1$$
Example — Building an AST

Reduce: Reduce → Term

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

\( \text{Reduce: } \text{Reduce} \rightarrow \text{Term} \quad \text{\$\$ = \$1} \)
Example — Building an AST

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

( num + num ) * num
```

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

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

num

Shift: push 'num' onto stack
```
Example — Building an AST

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

Reduce:  Term → Factor  $$$ = $1
Example — Building an AST

```
num
num
num

num
num

mult Node

add Node

numNode

num

num

num

Expr

Expr

Expr

Term

Factor

Factor

num

num

num

num


Reduce:  Expr → Expr * 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

\[
\text{Expr}_0 \rightarrow \text{Expr}_1 \cdot \text{Term} \quad \{ \text{Expr}_0\text{.val} = \text{Expr}_1\text{.val} \cdot \text{Term}\text{.val} \}\]
Example: Calculator Revisited

<table>
<thead>
<tr>
<th>Production</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line ( \rightarrow ) Expr</td>
<td>{ print(Expr.val); }</td>
</tr>
<tr>
<td>Expr(_0) ( \rightarrow ) Expr(_0) + Term</td>
<td>{ Expr(_0).val = Expr(_1).val + Term.val; }</td>
</tr>
<tr>
<td></td>
<td>{ Expr(_0).val = Term.val; }</td>
</tr>
<tr>
<td>Term(_0) ( \rightarrow ) Term(_1) * Factor</td>
<td>{ Term(_0).val = Term(_1).val * Factor.val; }</td>
</tr>
<tr>
<td></td>
<td>{ Term(_0).val = Factor.val; }</td>
</tr>
<tr>
<td>Factor ( \rightarrow ) ( Expr )</td>
<td>{ Factor.val = Expr.val; }</td>
</tr>
<tr>
<td></td>
<td>{ Factor.val = digit.lexval; }</td>
</tr>
</tbody>
</table>
Example: Calculator Revisited

Production | Actions
---|---
Line → Expr | { print(Expr.val); }
Expr₀ → Expr₀ + Term | { Expr₀.val = Expr₁.val + Term.val; }
  | Term | { Expr₀.val = Term.val; }
Term₀ → Term₁ * Factor | { Term₀.val = Term₁.val * Factor.val; }
  | Factor | { Term₀.val = Factor.val; }
Factor → (Expr) | { Factor.val = Expr.val; }
  | digit | { Factor.val = digit.lexval; }

Production | Actions
---|---
Line → Expr | { print(stack[top].val); }
Expr₀ → Expr₀ + Term | { stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }
  | Term | { }
Term₀ → Term₁ * Factor | { stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }
  | Factor | { }
Factor → (Expr) | { stack[top-2].val = stack[top-1].val; top = top - 2; }
  | digit | { stack[top].val = digit.lexval; }
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 $A \rightarrow X_1X_2,...,X_n$ depends only on:
    • The Attributes of the Symbols $X_1,X_2,...,X_{j-1}$ 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
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
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 → X { 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

```
Production   Semantic Rule
S \rightarrow aAC   C.i = A.s
S \rightarrow bABC   C.i = A.s
C \rightarrow c       C.s = func(C.i)
```

• 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

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → aAC</td>
<td>C.i = A.s</td>
</tr>
<tr>
<td>S → bABMC</td>
<td>M.i = A.s; C.i = M.s</td>
</tr>
<tr>
<td>C → c</td>
<td>C.s = func(C.i)</td>
</tr>
<tr>
<td>M → ε</td>
<td>M.s = M.i</td>
</tr>
</tbody>
</table>

Diagram:

```
  S
 /|
 / |
 b A C
```

```
  S
 /|
 / |
 b A B C
```

```
  S
 /|
 / |
 b A B
```

```
  S
 /|
 / |
 b A B M C
```
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.

\[
\text{Decl} \rightarrow \text{List} \ ']' \ Type \\
\text{Type} \rightarrow \text{integer} \mid \text{real} \\
\text{List} \rightarrow \text{List} \ ']' \ id \mid id \\
\]

\[
\text{Decl} \rightarrow \text{id} \ \text{List} \\
\text{List} \rightarrow ',' \ \text{id} \ \text{List} \mid ']' \ Type \\
\text{Type} \rightarrow \text{integer} \mid \text{real} \\
\]
Example

• What Does This Scheme Do?

```plaintext
Expr  → Term Factor_1
Factor_0 → addOp Term { print(addOp.lexeme) } Factor_1
            | ε
Term → num { print(num.val) }
```

Input: 9 - 5 + 2
Example

• What Does This Scheme Do?

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Term} \ Factor_1 \\
\text{Factor}_0 & \rightarrow \text{addOp} \ Term \ {\{\text{print(addOp.lexeme)}\}} \ Factor_1 \\
& \quad | \ \epsilon \\
\text{Term} & \rightarrow \text{num} \ {\{\text{print(num.val)}\}}
\end{align*}
\]

Input: 9 - 5 + 2

```
Expr
  └── Term
      ├── Factor
      │   └── addOp
      │       └── Term
      │           {print('9')}
      │           {print('5')}
      │           {print('+')}
      │           Factor
      │           └── ε
      └── Factor
          └── Term
              {print('2')}
```

```
Expr
  └── Term
      ├── Factor
      │   └── addOp
      │       └── Term
      │           {print('9')}
      │           {print('5')}
      │           {print('+')}
      │           Factor
      │           └── ε
      └── Term
          {print('2')}
```

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
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