Syntax directed Translation

• Models for translation from parse trees into assembly/machine code

• Representation of translations
  – Attribute Grammars (semantic actions for CFGs)
  – Tree Matching Code Generators
  – Tree Parsing Code Generators
Attribute Grammars

- Syntax-directed translation uses a grammar to produce code (or any other “semantics”)
- Consider this technique to be a generalization of a CFG definition
- Each grammar symbol is associated with an attribute
- An attribute can be anything: a string, a number, a tree, any kind of record or object
Attribute Grammars

- A CFG can be viewed as a (finite) representation of a function that relates strings to parse trees.
- Similarly, an attribute grammar is a way of relating strings with “meanings”.
- Since this relation is syntax-directed, we associate each CFG rule with a semantics (rules to build an abstract syntax tree).
- In other words, attribute grammars are a method to decorate or annotate the parse tree.
Example

```
Expr
  /   \\
Expr    B-op
  /     |
Var     +
  /     |
Var     Expr
  /     |
Expr   B-op
  /     |
Expr   Var
  /     |
Expr

```

`a.\text{lexval}=4`

`b.\text{lexval}=3`

`c.\text{lexval}=5`
Example
Example

Expr.val=19
  Expr
    Expr
      Var
        a
        a.lexval=4
        Var.val=4
      Expr
        Expr.val=3
        B-op
        +
        Var
          Var.val=3
          b
          b.lexval=3
      Expr
        B-op
        *
        Var
          Var.val=5
          c
          c.lexval=5
        Expr
          Expr.val=5
Syntax directed definition

Var → IntConstant
    { $0.val = $1.lexval; }
Expr → Var
    { $0.val = $1.val; }
Expr → Expr B-op Expr
    { $0.val = $2.val ($1.val, $3.val); }
B-op → +
    { $0.val = PLUS; }
B-op → *
    { $0.val = TIMES; }
Flow of Attributes in *Expr*

- Consider the flow of the attributes in the *Expr* syntax-directed defn
- The lhs attribute is computed using the rhs attributes
- Purely bottom-up: compute attribute values of all children (rhs) in the parse tree
- And then use them to compute the attribute value of the parent (lhs)
Synthesized Attributes

• **Synthesized attributes** are attributes that are computed purely bottom-up

• A grammar with semantic actions (or syntax-directed definition) can choose to use *only* synthesized attributes

• Such a grammar plus semantic actions is called an **S-attributed definition**
Inherited Attributes

• Synthesized attributes may not be sufficient for all cases that might arise for semantic checking and code generation

• Consider the (sub)grammar:
  
  \[
  \text{Var-decl} \rightarrow \text{Type \ Id-comma-list} ; \\
  \text{Type} \rightarrow \text{int} | \text{bool} \\
  \text{Id-comma-list} \rightarrow \text{ID} \\
  \text{Id-comma-list} \rightarrow \text{ID} , \text{Id-comma-list}
  \]
Example: \( \texttt{int} \ x, \ y, \ z \ ; \)
Example: \( int \ x, \ y, \ z; \)
Syntax-directed definition

Var-decl → Type Id-comma-list ;
   { $2.in = $1.val; }
Type → int | bool
   { $0.val = int; } & { $0.val = bool; }
Id-comma-list → ID
   { $1.val = $0.in; }
Id-comma-list → ID , Id-comma-list
   { $1.val = $0.in; $3.in = $0.in; }
Flow of Attributes in *Var-decl*

- How do the attributes flow in the *Var-decl* grammar?
- **ID** takes its attribute value from its parent node.
- **Id-Comma-List** takes its attribute value from its left sibling **Type**.
- Computing attributes purely bottom-up is not sufficient in this case.
- Do we need synthesized attributes in this grammar?
Inherited Attributes

• **Inherited attributes** are attributes that are computed at a node based on attributes from siblings or the parent

• Typically we combine synthesized attributes and inherited attributes

• It is possible to convert the grammar into a form that *only* uses synthesized attributes
Removing Inherited Attributes

Var-decl

Type-list  ID  ;

Type-list  ID ,

Type-list  ID ,

Type-list  ID ,

Type

Type

int

int x, y, z ;
Removing Inherited Attributes

```
int x, y, z;
Var-decl
  Type-list
    Type-list
      Type-list
        Type-list
          Type
            int
          Type-list
            ID
          Type-list
            ID
          Type-list
            ID
```

Var-decl.val=int
Type-list.val=int
Type-list.val=int
Removing inherited attributes

\[
\begin{align*}
\text{Var-decl} & \rightarrow \text{Type-List} \ D \ ; \\
& \quad \{ \ 0.\text{val} = 1.\text{val}; \ \} \\
\text{Type-list} & \rightarrow \text{Type-list} \ D \ , \\
& \quad \{ \ 0.\text{val} = 1.\text{val}; \ \} \\
\text{Type-list} & \rightarrow \text{Type} \\
& \quad \{ \ 0.\text{val} = 1.\text{val}; \ \} \\
\text{Type} & \rightarrow \text{int} \ | \ \text{bool} \\
& \quad \{ \ 0.\text{val} = \text{int}; \ \} \ & \& \ \{ \ 0.\text{val} = \text{bool}; \ \}
\end{align*}
\]
Direction of inherited attributes

- Consider the syntax directed defns:
  \[ A \rightarrow L \ M \]
  \[
  \{ \$1.in = \$0.in; \$2.in = \$1.val; \$0.val = \$2.val; \}
  \]
  \[ A \rightarrow Q \ R \]
  \[
  \{ \$2.in = \$0.in; \$1.in = \$2.val; \$0.val = \$1.val; \}
  \]
- Problematic definition: \$1.in = \$2.val
- Difference between incremental processing vs. using the completed parse tree
Incremental Processing

- Incremental processing: constructing output as we are parsing
- Bottom-up or top-down parsing
- Both can be viewed as left-to-right and depth-first construction of the parse tree
- Some inherited attributes cannot be used in conjunction with incremental processing
L-attributed Definitions

• A syntax-directed definition is **L-attributed** if for a CFG rule $A \rightarrow X_1..X_{j-1}X_j..X_n$ two conditions hold:
  – Each inherited attribute of $X_j$ depends on $X_1..X_{j-1}$
  – Each inherited attribute of $X_j$ depends on $A$
• These two conditions ensure left to right and depth first parse tree construction
• Every S-attributed definition is L-attributed
Top-down translation

• Assume that we have a top-down predictive parser
• Typical strategy: take the CFG and eliminate left-recursion
• Suppose that we start with an attribute grammar
• Can we still eliminate left-recursion?
Top-down translation

\[
\begin{align*}
E & \rightarrow E + T \\
& \quad \{ \; \$0.\text{val} = \$1.\text{val} + \$3.\text{val}; \; \} \\
E & \rightarrow E - T \\
& \quad \{ \; \$0.\text{val} = \$1.\text{val} - \$3.\text{val}; \; \} \\
T & \rightarrow \text{IntConstant} \\
& \quad \{ \; \$0.\text{val} = \$1.\text{lexval}; \; \} \\
E & \rightarrow T \\
& \quad \{ \; \$0.\text{val} = \$1.\text{val}; \; \} \\
T & \rightarrow ( \; E \; ) \\
& \quad \{ \; \$0.\text{val} = \$1.\text{val}; \; \}
\end{align*}
\]
Top-down translation

\[ E \rightarrow T \; R \]
{ \$2.in = \$1.val; \$0.val = \$2.val; }

\[ R \rightarrow + \; T \; R \]
{ \$3.in = \$0.in + \$2.val; \$0.val = \$3.val; }

\[ R \rightarrow - \; T \; R \]
{ \$3.in = \$0.in - \$2.val; \$0.val = \$3.val; }

\[ R \rightarrow \epsilon \]
{ \$0.val = \$0.in; }

\[ T \rightarrow ( \; E \; ) \]
{ \$0.val = \$1.val; }

\[ T \rightarrow \text{IntConstant} \]
{ \$0.val = \$1.lexval; }
Example: $9 - 5 + 2$
Example: 9 - 5 + 2
Translation Scheme

• A translation scheme is a CFG where each rule is associated with a semantic attribute
• A TS that maps infix expressions to postfix:
  
  $E \rightarrow T \ R$
  
  $R \rightarrow + \ T \ { \{ \ print( \ ‘+’ \ ); \} \ R}$
  
  $R \rightarrow - \ T \ { \{ \ print( \ ‘-’ \ ); \} \ R}$
  
  $R \rightarrow \varepsilon$
  
  $T \rightarrow \text{id} \ { \{ \ print( \ \text{id}.lookup \ ); \} \}$
LR parsing and inherited attributes

• As we just saw, inherited attributes are possible when doing top-down parsing
• How can we compute inherited attributes in a bottom-up shift-reduce parser
• Problem: doing it incrementally (while parsing)
• Note that LR parsing implies depth-first visit which matches L-attributed definitions
LR parsing and inherited attributes

- Attributes can be stored on the stack used by the shift-reduce parsing.
- For synthesized attributes: when a reduce action is invoked, store the value on the stack based on value popped from stack.
- For inherited attributes: transmit the attribute value when executing the `goto` function.
Example: Synthesized Attributes

\[
\begin{align*}
T & \rightarrow F \quad \{ \; \$0.\text{val} = \$1.\text{val} ; \; \} \\
T & \rightarrow T \ast F \\
& \quad \{ \; \$0.\text{val} = \$1.\text{val} \ast \$3.\text{val} ; \; \} \\
F & \rightarrow \text{id} \\
& \quad \{ \; \text{val} := \text{id}.\text{lookup}() ; \\
&\hspace{1em} \text{if} \ (\text{val}) \; \{ \; \$0.\text{val} = \$1.\text{val} ; \; \} \\
&\hspace{1em} \text{else} \; \{ \; \text{error} ; \; \} \; \} \\
F & \rightarrow ( \; T \; ) \quad \{ \; \$0.\text{val} = \$1.\text{val} ; \; \}
\end{align*}
\]
Trace “(id\text{val}=3)^*id\text{val}=2”

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(id) * id $</td>
<td>Shift 5</td>
<td>a.Push id.val=3;</td>
</tr>
<tr>
<td>0 5</td>
<td>id * id $</td>
<td>Shift 8</td>
<td>{ $0.val = $1.val }</td>
</tr>
<tr>
<td>0 5 8</td>
<td>) * id $</td>
<td>Reduce 3 F→id, pop 8, goto [5,F]=1</td>
<td>a.Pop; a.Push 3;</td>
</tr>
<tr>
<td>0 5 1</td>
<td>) * id $</td>
<td>Reduce 1 T→F, pop 1, goto [5,T]=6</td>
<td>{ $0.val = $1.val }</td>
</tr>
<tr>
<td>0 5 6</td>
<td>) * id $</td>
<td>Shift 7</td>
<td>a.Pop; a.Push 3;</td>
</tr>
<tr>
<td>0 5 6 7</td>
<td>* id $</td>
<td>Reduce 4 F→(T), pop 7 6 5, goto [0,F]=1</td>
<td>{ $0.val = $2.val }</td>
</tr>
<tr>
<td></td>
<td>* id $</td>
<td>3 pops; a.Push 3</td>
<td></td>
</tr>
</tbody>
</table>
Trace ‘‘(id_{val=3})*id_{val=2}’’

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>* id $</td>
<td>Reduce 1 T→F, pop 1, goto [0,T]=2</td>
<td>{ $0.val = $1.val }</td>
</tr>
<tr>
<td>0 2</td>
<td>* id $</td>
<td>Shift 3</td>
<td>a.Pop; a.Push 3</td>
</tr>
<tr>
<td>0 2 3</td>
<td>id $</td>
<td>Shift 8</td>
<td>a.Push mul</td>
</tr>
<tr>
<td>0 2 3 8</td>
<td>$</td>
<td>Reduce 3 F→id, pop 8, goto [3,F]=4</td>
<td>a.Push id.val=2</td>
</tr>
<tr>
<td>0 2 3 4</td>
<td>$</td>
<td>Reduce 2 T→T * F, pop 4 3 2, goto [0,T]=2</td>
<td>a.Pop a.Push 2</td>
</tr>
<tr>
<td>0 2</td>
<td>$</td>
<td>Accept</td>
<td>3 pops; $0.val = $1.val * $2.val; }</td>
</tr>
</tbody>
</table>

3 pops; a.Push 3*2=6
Example: Inherited Attributes

\[
E \rightarrow T \ R \\
\quad \{ \, \$2.in = \$1.val; \, \$0.val = \$2.val; \, \} \\
R \rightarrow + \ T \ R \\
\quad \{ \, \$3.in = \$0.in + \$2.val; \, \$0.val = \$3.val; \, \} \\
R \rightarrow \varepsilon \quad \{ \, \$0.val = \$0.in; \, \} \\
T \rightarrow ( \ E ) \quad \{ \, \$0.val = \$1.val; \, \} \\
T \rightarrow \text{id} \quad \{ \, \$0.val = \text{id}.lookup; \, \}
\]
Productions

1: E → T R
2: R → + T R
3: R → ε
4: T → (E)
5: T → id

0: S' → • E
   E → • T R
   T → • (E)
   T → • id

8: S' → E •
Reduce 0

1: E → T • R
   R → • + T R
Reduce 1

2: E → T R •

3: T → (• E)
   E → • T R
   T → • (E)
   T → • id

Reduce 2

4: R → + • T R
   T → • (E)
   T → • id

Reduce 3

5: R → + T • R
   R → • + T R
Reduce 4

7: T → id •
Reduce 5

9: E
10: T
+ R

Productions

Reduce 0
Reduce 1
Reduce 3
Reduce 5
Reduce 2
Trace \( \text{``}\text{id}_{\text{val}=3} + \text{id}_{\text{val}=2}\text{''} \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id + id $</td>
<td>Shift 7</td>
<td>{ $0.\text{val} = \text{id}.\text{val}</td>
</tr>
<tr>
<td>0 7</td>
<td>+ id $</td>
<td>Reduce 5 T→id pop 7, goto [0,T]=1</td>
<td>pop; attr.Push(3)</td>
</tr>
<tr>
<td>0 1</td>
<td>+ id $</td>
<td>Shift 4</td>
<td>$2.\text{in} = $1.\text{val}</td>
</tr>
<tr>
<td>0 1 4</td>
<td>id $</td>
<td>Shift 7</td>
<td>$\text{R.in} := (1)\text{.attr}</td>
</tr>
<tr>
<td>0 1 4 7</td>
<td>$</td>
<td>Reduce 5 T→id pop 7, goto [4,T]=5</td>
<td>{ $0.\text{val} = \text{id}.\text{val}</td>
</tr>
<tr>
<td>0 1 4 5</td>
<td>$</td>
<td>Reduce 3 R→ε goto [5,R]=6</td>
<td>pop; attr.Push(2); }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{ $3.\text{in} = $0.\text{in}+$1.\text{val}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(5).\text{attr} = (1).\text{attr}+2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$0.\text{val} = $0.\text{in}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$0.\text{val} = (5).\text{attr} = 5 }</td>
</tr>
</tbody>
</table>
Trace \( "\text{id}_{\text{val}=3} + \text{id}_{\text{val}=2}" \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Attributes</th>
</tr>
</thead>
</table>
| 0 1 4 5 6 | $     | Reduce 2 R \rightarrow + T R  
Pop 4 5 6, goto [1,R]=2 | \{ $0.\text{val} = $3.\text{val}  
pop; \text{attr.Push}(5); \} |
| 0 1 2   | $     | Reduce 1 E \rightarrow T R  
Pop 1 2, goto [0,E]=8         | \{ $0.\text{val} = $3.\text{val}  
pop; \text{attr.Push}(5); \} |
| 0 8     | $     | Accept                          | \{ $0.\text{val} = 5  
attr.top = 5; \}                                               |
Marker Non-terminals

\[ E \rightarrow T \ R \]
\[ R \rightarrow + \ T \ \{ \ \text{print( ‘+’)}; \ \} \ R \]
\[ R \rightarrow - \ T \ \{ \ \text{print( ‘-’)}; \ \} \ R \]
\[ R \rightarrow \epsilon \]
\[ T \rightarrow \text{id} \ \{ \ \text{print( id.lookup )}; \ \} \]

Actions that should be done after recognizing T but before predicting R
Marker Non-terminals

\[
E \rightarrow T \ R \\
R \rightarrow + \ T \ M \ R \\
R \rightarrow - \ T \ N \ R \\
R \rightarrow \epsilon \\
T \rightarrow \text{id} \ \{ \ \text{print( id.lookup );} \ \} \\
M \rightarrow \epsilon \ \{ \ \text{print( ‘+’ );} \ \} \\
N \rightarrow \epsilon \ \{ \ \text{print( ‘-’ );} \ \}
\]

Equivalent SDT using
marker non-terminals
Tree Matching Code Generators

- Write tree patterns that match portions of the parse tree
- Each tree pattern can be associated with an action (just like attribute grammars)
- There can be multiple combinations of tree patterns that match the input parse tree
Tree Matching Code Generators

• To provide a unique output, we assign costs to the use of each tree pattern
• E.g. assigning uniform costs leads to smaller code or instruction costs can be used for optimizing code generation
• Three algorithms: Maximal Munch ($\S 9.12$), Dynamic Programming ($\S 9.11$), Tree Grammars
Maximal Munch: Example 1

\[ a \cdot (b \cdot c) \]

\[ a.\text{lexval}=4 \]
\[ b.\text{lexval}=3 \]
\[ c.\text{lexval}=5 \]
Maximal Munch: Example 2

Top-down
Fit the largest tile
Recursively descend

a.\text{lexval}=4
b.\text{lexval}=3
c.\text{lexval}=5
Maximal Munch: Example 2

Checking for semantic errors with Tree-matching
Tree Parsing Code Generators

• Take the prefix representation of the syntax tree
  – E.g. (+ (* c1 r1) (+ ma c2)) in prefix representation uses an inorder traversal to get + * c1 r1 + ma c2

• Write CFG rules that match substrings of the above representation and non-terminals are registers or memory locations

• Each matching rule produces some predefined output

• Example 9.18 (Dragon book)
Code-generation Generators

• A CGG is like a compiler-compiler: write down a description and generate code for it

• Code generation by:
  – Adding semantic actions to the original CFG and each action is executed while parsing, e.g. yacc
  – Tree Rewriting: match a tree and commit an action, e.g. lcc
  – Tree Parsing: use a grammar that generates trees (not strings), e.g. twig, burs, iburg
Summary

• The parser produces concrete syntax trees
• Abstract syntax trees: define semantic checks or a syntax-directed translation to the desired output
• Attribute grammars: static definition of syntax-directed translation
  – Synthesized and Inherited attributes
  – S-attribute grammars
  – L-attributed grammars
• Complex inherited attributes can be defined if the full parse tree is available