CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop
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  Expr
  |  +   |
Var  
  |
a
```

```
Expr
  /\  /
Expr  B-op  Expr
  |  *   |
Var  |
  |
b
```

```
Expr
  /\  /
Var  |
  |
c
```

```
a.lexval=4
b.lexval=3
c.lexval=5
```
Example

```
Expr
  /\    /
Expr  B-op  Expr
  |      |
Var + Expr B-op Expr
  |     |    |
a       Var  b
     /\    |
Expr  Expr
  |    |
Var * Var
  |  |
b c
```

a.lexval=4
Var.val=3
b.lexval=3
Var.val=5
c.lexval=5
Expr.val=3
Expr.val=5
Expr.val=4
Var.val=3
Var.val=5
Example

```
Expr.val=19
   / \      / \      / \       / \
  Expr B-op Expr
 /      /      /      /      /
Var + Var Expr B-op Expr
     |      |      |      |
a b  c
```

- `Expr.val` values:
  - Expr.val=19
  - Expr.val=15
  - Expr.val=15

- `Var.val` values:
  - Var.val=4
  - Var.val=3
  - Var.val=5

- `a.lexval`:
  - a.lexval=4

- `b.lexval`:
  - b.lexval=3

- `c.lexval`:
  - c.lexval=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; }

In yacc: { $$ = $1 }
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:

  Var-decl → Type Id-comma-list ;
  Type → int | bool
  Id-comma-list → ID
  Id-comma-list → ID , Id-comma-list
Example: \texttt{int} \texttt{x, y, z ;}

```
int x, y, z;
```

```
Var-decl
  Type
  Id-Comma-List
```

```
Id-Comma-List
  Type
  ID ,
  Id-Comma-List
```

```
Id-Comma-List
  ID ,
  Id-Comma-List
```

```
Id-Comma-List
  ID ,
  Id-Comma-List
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```

```
ID
```
Example: \( \text{int } x, y, z \);
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; }
Syntax-directed definition

Var-decl → Type Id-comma-list ;

In yacc: Var-decl → Type { $<val>$ = $1 } Id-comma-list

Type → int | bool
{ $0.val = int; } & { $0.val = bool; }

Id-comma-list → ID
{ $1.val = $0.in; } ---→ In yacc: { $1 = $<val>0 }

Id-comma-list → ID , Id-comma-list
{ $1.val = $0.in; $3.in = $0.in; }

12-12-08
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

int x, y, z ;
Removing Inherited Attributes

\[
\text{int } x, y, z ;
\]
Removing inherited attributes

Var-decl → Type-List ID ;
  { $0.val = $1.val; }
Type-list → Type-list ID ,
  { $0.val = $1.val; }
Type-list → Type
  { $0.val = $1.val; }
Type → int | bool
  { $0.val = int; } & { $0.val = bool; }
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
Syntax-directed defns

• Two important classes of SDTs:
  1. LR parser, syntax directed definition is S-attributed
  2. LL parser, syntax directed definition is L-attributed
Syntax-directed defns

• LR parser, S-attributed definition
  • Implementing S-attributed definitions in LR parsing is easy: execute action on reduce, all necessary attributes have to be on the stack

• LL parser, L-attributed definition
  • Implementing L-attributed definitions in LL parsing is similarly easy: we use an additional action record for storing synthesized and inherited attributes on the parse stack
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

E → E + T
   { $0.val = $1.val + $3.val; }

E → E - T
   { $0.val = $1.val - $3.val; }

T → IntConstant
   { $0.val = $1.lexval; }

E → T
   { $0.val = $1.val; }

T → ( E )
   { $0.val = $2.val; }
Top-down translation

E → T R
   { $2.in = $1.val; $0.val = $2.val; }
R → + T R
   { $3.in = $0.in + $2.val; $0.val = $3.val; }
R → - T R
   { $3.in = $0.in - $2.val; $0.val = $3.val; }
R → ε   { $0.val = $0.in; }
T → ( E ) { $0.val = $2.val; }
T → IntConstant { $0.val = $1.lexval; }
Example: 9 - 5 + 2
Example: $9 - 5 + 2$
Dependencies and SDTs

- There can be circular definitions:
  
  \[
  A \rightarrow B \{ \ \text{$0.val = $1.in; $1.in = $0.val + 1;} \ \}
  \]

- It is impossible to evaluate either $0.val or $1.in first (each value depends on the other)

- We want to avoid circular dependencies

- Detecting such cases in all parse trees takes exponential time!

- S-attributed or L-attributed definitions cannot have cycles
Dependency Graphs

E → T.val → R.in
IntConst

9

- → T.val
IntConst

9

+ → T.val
IntConst

2

R.in

ε

5

4

6

Dependency Graphs

- A dependency graph is drawn based on the syntax directed definition
- Each dependency shows the flow of information in the parse tree
- There are many ways to order these dependencies
- Each ordering is called a topological sort of the dependency edges
- A graph with a cycle has no possible topological sorting
The graph shown to the left has many valid topological sorts, including:

- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 3, 7, 8, 5, 11, 10, 2, 9
- 5, 7, 3, 8, 11, 10, 9, 2 (least number of edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 7, 5, 11, 2, 3, 8, 9, 10
Dependency Graphs

E

IntConst

T.val

R.in

IntConst

T.val

IntConst

R.in

T.val

IntConst

ε
Dependency Graphs

• A topological sort is defined on a set of nodes $N_1, \ldots, N_k$ such that if there is an edge in the graph from $N_i$ to $N_j$ then $i < j$

• One possible topological sort for previous dependency graph is:
  – 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

• Another possible sorting is:
  – 4, 5, 7, 8, 1, 2, 3, 6, 9, 10, 11, 12
Syntax-directed definition with actions

- Some definitions can have side-effects:
  
  $E \rightarrow T \ R \ \{ \ printf("%s", \ $2); \ \}$

- Can we predict when these side-effects will occur?

- In general, we cannot and so the translation will depend on the parser
Syntax-directed definition with actions

• A definition with side-effects:
  E → T R { printf("%s", $2); }

• We can impose a condition: allow side-effects if the definition obeys a condition:
  – The same translation is produced for any topological sort of the dependency graph

• In the above example, this is true because the print statement is executed at the end
SDTs with Actions

• A syntax directed definition that maps infix expressions to postfix:

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

• A buggy syntax directed definition that tries to map infix expressions to prefix:

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

Problematic for left to right processing. Translation on the parse tree is possible.
Marker non-terminals

• Convert L-attributed into S-attributed definition
• Prerequisite: use embedded actions to compute inherited attributes, e.g.

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

• For each embedded action introduce a new marker non-terminal and replace action with the marker

\[ R \rightarrow + T \ M \ R \ \{\$0.\text{val} = \$-1.\text{val}\} \]

\[ M \rightarrow \epsilon \ \{\ \$0.\text{val} = \$-1.\text{val} +\$-3.\text{in}; \} \]

note the use of \(-1\), \(-2\), etc. to access attributes
Marker Non-terminals

E → T R
R → + T { print( ‘+’ ); } R
R → - T { print( ‘-’ ); } R
R → ε
T → id { print( id.lookup ); }

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

E → T R
R → + T M R
R → - T N R
R → ε
T → id { print( id.lookup ); }
M → ε { print( ‘+’ ); }
N → ε { print( ‘-’ ); }

Equivalent SDT using marker non-terminals
Impossible Syntax-directed Definition

E → { print( ‘+’ ); } E + T
E → T
T → { print( ‘*’ ); } T * R
T → F
T → id { print $1.lexval; }

Tries to convert infix to prefix

Causes a reduce/reduce conflict when marker non-terminals are introduced.
Extra Slides
Syntax-directed defns

- LR parser, S-attributed definition
- more details later ...
- LL parser, L-attributed definition

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T')T'F</td>
<td>id)*id$</td>
<td>$T \rightarrow F T' { $2.in = $1.val }$</td>
</tr>
<tr>
<td>$T')T'id</td>
<td>id)*id$</td>
<td>$F \rightarrow id { $0.val = $1.val }$</td>
</tr>
<tr>
<td>$T')T'</td>
<td>*)id$</td>
<td>The action record stays on the stack when $T'$ is replaced with rhs of rule</td>
</tr>
</tbody>
</table>

action record: $T'.in = F.val$
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

T → F \( \{ \ $0.\text{val} = \ $1.\text{val}; \ \} \)
T → T * F
\( \{ \ $0.\text{val} = \ $1.\text{val} * \ $3.\text{val}; \ \} \)
F → id
\( \{ \ \text{val} := \ \text{id}.\text{lookup}(); \)
  if (val) \{ \ $0.\text{val} = \ $1.\text{val}; \ \}
  else \{ \ \text{error}; \ \} \}
F → ( T ) \{ \ $0.\text{val} = \ $2.\text{val}; \ \}
Productions

1: T → F
2: T → T*F
3: F → id
4: F → (T)

0: S’ → • T
   T → • F
   T → • T * F
   F → • id
   F → • (T)

1: T → F •

2: S’ → T •
   T → T • * F

3: T → T * • F
   F → • id
   F → • (T)

4: T → T * F •

5: F → ( • T )
   T → • F
   T → • T * F
   F → • id
   F → • (T)

6: F → (T •)
   T → T • * F

7: F → (T) •

8: F → id •

$ Accept

Reduce 1

Reduce 2

Reduce 3

Reduce 4
Trace \((id_{val=3})\cdot id_{val=2}\)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>a.Push id.val=3; { $0.val = $1.val }</td>
</tr>
<tr>
<td>0 5</td>
<td>( id ) * id</td>
<td>Shift 5</td>
<td>a.Pop; a.Push id.val=3; { $0.val = $1.val }</td>
</tr>
<tr>
<td>0 5 8</td>
<td>id ) * id</td>
<td>Shift 8</td>
<td>a.Pop; a.Push id.val=3; { $0.val = $1.val }</td>
</tr>
<tr>
<td>0 5 1</td>
<td>) * id</td>
<td>Reduce 3 F→id, pop 8, goto [5,F]=1</td>
<td>a.Pop; a.Push id.val=3; { $0.val = $2.val }</td>
</tr>
<tr>
<td>0 5 6</td>
<td>) * id</td>
<td>Reduce 1 T→ F, pop 1, goto [5,T]=6</td>
<td>a.Pop; a.Push id.val=3; { $0.val = $2.val }</td>
</tr>
<tr>
<td>0 5 6 7</td>
<td>* id</td>
<td>Shift 7</td>
<td>a.Pop; a.Push id.val=3; { $0.val = $2.val }</td>
</tr>
<tr>
<td></td>
<td>* id</td>
<td>Reduce 4 F→ (T), pop 7 6 5, goto [0,F]=1</td>
<td>3 pops; a.Push 3</td>
</tr>
</tbody>
</table>
Trace \( \langle \text{id}_{\text{val}=3} \rangle \text{id}_{\text{val}=2} \rangle \)

<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.\text{val} = $1.\text{val} } \ a.\text{Pop}; \ a.\text{Push} 3</td>
</tr>
<tr>
<td>0 2</td>
<td>* id $</td>
<td>Shift 3</td>
<td>\ a.\text{Push} \text{id} \text{.val}=2</td>
</tr>
<tr>
<td>0 2 3</td>
<td>id $</td>
<td>Shift 8</td>
<td>\ a.\text{Pop} \ a.\text{Push} 2</td>
</tr>
<tr>
<td>0 2 3 8</td>
<td>$</td>
<td>Reduce 3 F → id, pop 8, goto [3,F]=4</td>
<td>{ $0.\text{val} = $1.\text{val} \times $3.\text{val}; }</td>
</tr>
<tr>
<td>0 2 3 4</td>
<td>$</td>
<td>Reduce 2 T → T * F, pop 4 3 2, goto [0,T]=2</td>
<td>3 pops; \ a.\text{Push} 3\times2=6</td>
</tr>
<tr>
<td>0 2</td>
<td>$</td>
<td>Accept</td>
<td></td>
</tr>
</tbody>
</table>
Example: Inherited Attributes

E → T R
  \{ $2.in = $1.val; $0.val = $2.val; \}
R → + T R
  \{ $3.in = $0.in + $2.val; $0.val = $3.val; \}
R → ε \{ $0.val = $0.in; \}
T → ( E ) \{ $0.val = $1.val; \}
T → id \{ $0.val = id.lookup; \}
### Productions

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

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

3. \( R \rightarrow \epsilon \) \{ $0.val = $0.in; \}

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

5. \( T \rightarrow id \) \{ $0.val = id.lookup; \}

---

### Attributes

1. \( \{ \) $0.val = id.lookup \}

2. \( \{ \) pop; attr.Push(3) $2.in = $1.val $2.in := (1).attr \}

3. \( \{ \) $0.val = id.lookup \}

4. \( \{ \) pop; attr.Push(2); \}

5. \( \{ \$3.in = $0.in+$1.val (5).attr := (1).attr+2 $0.val = $0.in $0.val = (5).attr = 5 \} \)

---

<table>
<thead>
<tr>
<th>Action</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce 5 ( T \rightarrow id )</td>
<td>pop 7, goto [0,T]=1</td>
</tr>
<tr>
<td>Shift 4</td>
<td>{ pop; attr.Push(3) $2.in = $1.val $2.in := (1).attr }</td>
</tr>
<tr>
<td>Shift 7</td>
<td>{ $0.val = id.lookup }</td>
</tr>
<tr>
<td>Reduce 5 ( T \rightarrow id )</td>
<td>pop 7, goto [4,T]=5</td>
</tr>
<tr>
<td>{ $0.val = id.lookup }</td>
<td>{ pop; attr.Push(2); }</td>
</tr>
<tr>
<td>Reduce 3 ( R \rightarrow \epsilon )</td>
<td>goto [5,R]=6</td>
</tr>
<tr>
<td>{ $3.in = $0.in+$1.val (5).attr := (1).attr+2 $0.val = $0.in $0.val = (5).attr = 5 }</td>
<td></td>
</tr>
</tbody>
</table>
### Trace “$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</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 7</td>
<td>id + id $</td>
<td>Shift 7</td>
<td>{ $0.val = id.lookup }</td>
</tr>
<tr>
<td></td>
<td>+ id $</td>
<td>Reduce 5 $T \rightarrow id</td>
<td>{ pop; attr.Push(3) }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop 7, goto [0,T]=1</td>
<td>$2.in = $1.val</td>
</tr>
<tr>
<td>0 1</td>
<td>+ id $</td>
<td>Shift 4</td>
<td>$2.in := (1).attr }</td>
</tr>
<tr>
<td>0 1 4</td>
<td>id $</td>
<td>Shift 7</td>
<td></td>
</tr>
<tr>
<td>0 1 4 7</td>
<td>$</td>
<td>Reduce 5 $T \rightarrow id</td>
<td>{ $0.val = id.lookup }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop 7, goto [4,T]=5</td>
<td>{ pop; attr.Push(2); }</td>
</tr>
<tr>
<td>0 1 4 5</td>
<td>$</td>
<td>Reduce 3 $R \rightarrow \varepsilon</td>
<td>{ $3.in = $0.in+$1.val }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>goto [5,R]=6</td>
<td>(5).attr := (1).attr+2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$0.val = $0.in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$0.val = (5).attr = 5 }</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 4 5 6</td>
<td>$</td>
<td>Reduce 2 R → + T R</td>
<td>{ $0.val = $3.val</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pop 4 5 6, goto [1,R]=2</td>
<td>pop; attr.Push(5); }</td>
</tr>
<tr>
<td>0 1 2</td>
<td>$</td>
<td>Reduce 1 E → T R</td>
<td>{ $0.val = $3.val</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pop 1 2, goto [0,E]=8</td>
<td>pop; attr.Push(5); }</td>
</tr>
<tr>
<td>0 8</td>
<td>$</td>
<td>Accept</td>
<td>{ $0.val = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>attr.top = 5; }</td>
</tr>
</tbody>
</table>
LR parsing with inherited attributes

Consider:

S → AB
{ $1\.in = 'x';
  $2\.in = $1\.val }

B → cbB
{ $0\.val = $0\.in + 'y'; }
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, Dynamic Programming, Tree Grammars
• Section 8.9 (Purple Dragon book)
Maximal Munch: Example 1

```
a + (b * c)
```

- `a.lexval = 4`
- `b.lexval = 3`
- `c.lexval = 5`
Maximal Munch: Example 1

Top-down
Fit the largest tile
Recursively descend

a.lexval=4

b.lexval=3
c.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

  • Section 8.9.3 (Purple 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