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

```plaintext
Expr
  /\Expr  B-op  Expr
 /  \      /     \     
Var  B-op  Var
     /   *     
  a      b   c
  a.lexval=4 b.lexval=3 c.lexval=5

Example

Expr
  /\Expr  B-op  Expr
 /  \      /     \     
Var  B-op  Var
     /   *     
  a      b   c
  a.lexval=4 b.lexval=3 c.lexval=5
```

a.lexval=4

b.lexval=3

c.lexval=5

a.lexval=4

b.lexval=3

c.lexval=5

a.lexval=4

b.lexval=3

c.lexval=5

Var.val=4

Var.val=3

Var.val=5

Expr.val=4

Expr.val=5

Expr.val=5

Expr.val=3
Example

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:
  
  \[\text{Var-decl} \rightarrow \text{Type Id-comma-list ;}\
  \text{Type} \rightarrow \text{int | bool}\
  \text{Id-comma-list} \rightarrow \text{ID}\
  \text{Id-comma-list} \rightarrow \text{ID , Id-comma-list}\

Example: \text{int x, y, z ;}
Example: \texttt{int x, y, z ;}

Syntax-directed definition

\begin{verbatim}
Var-decl \rightarrow Type Id-comma-list ;
  \{ $2.in = $1.val; \}

Type \rightarrow \texttt{int} \mid \texttt{bool}
  \{ $0.val = \texttt{int}; \} \& \{ $0.val = \texttt{bool}; \}

Id-comma-list \rightarrow ID
  \{ $1.val = $0.in; \}

Id-comma-list \rightarrow ID , Id-comma-list
  \{ $1.val = $0.in; $3.in = $0.in; \}
\end{verbatim}
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; }

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 ID ,
  Type ID ,
  Type ID ,
  int x, y, z ;
```
Removing Inherited Attributes

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; }
```

int x, y, z ;
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
  \( A \rightarrow X \ldots X_{j-1} X_j X_{j+1} \ldots X_n \) two conditions hold:
  – Each inherited attribute of \( X_j \) depends on \( X_1 \ldots X_{j-1} \)
  – 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

<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:**

\[ T'.in = F.val \]
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 → e { $0.val = $0.in; }
T → ( E ) { $0.val = $2.val; }
T → IntConstant { $0.val = $1.lexval; }

Example: 9 - 5 + 2
Example: 9 - 5 + 2

```
Example: 9 - 5 + 2
```

```
E.val

T.val

IntConst - T.val

IntConst + T.val

R.val

R.val

R.val

E.val

6

6

6

6

6

6

6

6

6

6

6

6

6

6

6

6
```

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

- 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

• 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:
  \( E \rightarrow T \ R \ \{ \ \text{printf("\%s", $2);} \ \} \)
• 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:
\[ E \rightarrow T \ R \{ \text{printf}("%s", \$2); \} \]

• 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:

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

Problematic for left to right processing. Translation on the parse tree is possible.

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

\[
\begin{align*}
T \rightarrow & F \quad \{ \$0.\text{val} = \$1.\text{val}; \} \\
T \rightarrow & T \ast F \\
& \quad \{ \$0.\text{val} = \$1.\text{val} \ast \$3.\text{val}; \} \\
F \rightarrow & \text{id} \\
& \quad \{ \text{val} := \text{id}.\text{lookup}(); \\
& \quad \quad \text{if} (\text{val}) \{ \$0.\text{val} = \$1.\text{val}; \} \\
& \quad \quad \text{else} \{ \text{error}; \} \} \\
F \rightarrow & ( T ) \quad \{ \$0.\text{val} = \$2.\text{val}; \}
\end{align*}
\]
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 5</td>
<td>(id)*id</td>
<td>Shift 5</td>
<td>a.Push id.val=3; { $0.val = $1.val }</td>
</tr>
<tr>
<td>0 5 8</td>
<td>*id</td>
<td>Shift 8</td>
<td>a.Pop; a.Push 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 3; { $0.val = $1.val }</td>
</tr>
<tr>
<td>0 5 6</td>
<td></td>
<td>Reduce 1 T→F, pop 1, goto [5,T]=6</td>
<td>a.Pop; a.Push 3; { $0.val = $2.val }</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>3 pops; a.Push 3</td>
</tr>
</tbody>
</table>
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 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.Pop a.Push (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>{ $0.val = $1.val * $3.val; }</td>
</tr>
<tr>
<td>0 2</td>
<td>$</td>
<td>Accept</td>
<td>3 pops; a.Push (3*2=6)</td>
</tr>
</tbody>
</table>

Example: Inherited Attributes

\[
\begin{align*}
E & \rightarrow T \ R \\
& \quad \{ \text{$_2$.in = $1$.val; $0$.val = $2$.val; } \} \\
R & \rightarrow + \ T \ R \\
& \quad \{ \text{$_3$.in = $0$.in + $2$.val; $0$.val = $3$.val; } \} \\
R & \rightarrow \epsilon \quad \{ \text{$0$.val = $0$.in; } \} \\
T & \rightarrow ( \ E ) \quad \{ \text{$0$.val = $1$.val; } \} \\
T & \rightarrow \text{id} \quad \{ \text{$0$.val = \text{id}.lookup; } \}
\end{align*}
\]
0: S' → E
E → T R
T → (E)
T → id

1: E → T R
R → + T R
R → ε

2: E → T R

3: T → (E)
E → T R
T → (E)
T → id

4: R → + T R
T → (E)
T → id

5: R → + T R
R → + T R
R → ε

6: R → + T R

7: T → id

8: S' → E

Reduce 0
Reduce 1
Reduce 2
Reduce 3
Reduce 4
Reduce 5
Trace "id_{val=3} + id_{val=2}"
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 pop; attr.Push(5); }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pop 4 5 6, goto [1,R]=2</td>
<td>{ $0.val = $3.val pop; attr.Push(5); }</td>
</tr>
<tr>
<td>0 1 2</td>
<td>$</td>
<td>Reduce 1 E → T R</td>
<td>{ $0.val = 5 attr.top = 5; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pop 1 2, goto [0,E]=8</td>
<td></td>
</tr>
<tr>
<td>0 8</td>
<td>$</td>
<td>Accept</td>
<td></td>
</tr>
</tbody>
</table>

LR parsing with inherited attributes

Consider:

\[ S \rightarrow AB \]
\{ $1.in = 'x'; \\  \\  \\  \\  $2.in = $1.val \}  \\

\[ B \rightarrow cbB \]
\{ $0.val = $0.in + 'y'; \}  \\

Parse stack at line 3:
\[ ['x'] A ['x'] c b B \]
\{ $1.in = 'x' $2.in = $1.val \}  \\

Parse stack at line 4:
\[ ['x'] A B \]
\[ ['xy'] \]
Marker non-terminals

- Convert L-attributed into S-attributed definition
- Prerequisite: use embedded actions to compute inherited attributes, e.g.
  \[ R \rightarrow + T \{ \$3.\text{in} = \$0.\text{in} + \$2.\text{val}; \} R \]
- For each embedded action introduce a new marker non-terminal and replace action with the marker
  \[ R \rightarrow + T M R \{ \$0.\text{val} = -1.\text{val} \} \]
  \[ M \rightarrow \varepsilon \{ \$0.\text{val} = -1.\text{val} + -3.\text{in}; \} \]

Marker Non-terminals

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

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

\[
E \rightarrow TR \\
R \rightarrow + TMR \\
R \rightarrow - TNT \\
R \rightarrow \epsilon \\
T \rightarrow \text{id} \{ \text{print}(\text{id}.\text{lookup}); \} \\
M \rightarrow \epsilon \{ \text{print}(\text{`+'}); \} \\
N \rightarrow \epsilon \{ \text{print}(\text{`-'}); \} \\
\]

Equivalent SDT using \textit{marker non-terminals}

Impossible Syntax-directed Definition

\[
E \rightarrow \{ \text{print}(\text{`+'}); \} E + T \\
E \rightarrow T \\
T \rightarrow \{ \text{print}(\text{`*'}); \} T * R \\
T \rightarrow F \\
T \rightarrow \text{id} \{ \text{print} \$1.\text{lexval}; \} \\
\]

Tries to convert infix to prefix

Causes a reduce/reduce conflict when marker non-terminals are introduced.
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

Top-down
Fit the largest tile
Recursively descend
Maximal Munch: Example 2

```plaintext
class kwclass ID { method_list } x = 0 \ x_j

method_list method_list

method_decl method_list

method_list method_list

method_decl

return_type ID { body }

main
```

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-attribute grammars
- Complex inherited attributes can be defined if the full parse tree is available