Syntax-Directed Translation

Introduction

- Translation of languages guided by context-free grammars
- Attach attributes to the grammar symbols
  - Values of the attributes are computed by semantic rules associated with the grammar productions
  - Attribute may represent:
    - Number, type, string, memory location, label, etc.
- Syntax-directed definitions:
  - Each production has a set of semantic rules associated with it
- Conceptually we parse the input token stream, build the parse tree and then traverse the tree as needed to evaluate the semantic rules.
  - The rules may generate code, save information in the symbol table, issue error messages, etc.
Introduction

- In some cases we don’t have to follow this outline literally
  - We may evaluate the rules during parsing
- Synthesized attribute:
  - The value of the attribute at a parent node can be found from the attributes of its children
    - It can be evaluated by traversing the tree bottom-up
- Inherited attribute:
  - The value of the attribute at a node can be found from the attributes at its parent node and/or its sibling nodes
    - Can be evaluated by traversing the tree top-down
- S-attributed definition:
  - A syntax-directed definition where all attributes are synthesized (the “S” stands for synthesized).

Example: Simple Desk Calculator

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>L → En</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>E → E₁ + T</td>
<td>E.val := E₁.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val := T.val</td>
</tr>
<tr>
<td>T → T₁ * F</td>
<td>T.val := T₁.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val := F.val</td>
</tr>
<tr>
<td>F → (E)</td>
<td>F.val := E.val</td>
</tr>
<tr>
<td>F → digit</td>
<td>F.val := digit.lexval</td>
</tr>
</tbody>
</table>
Synthesized Attributes

- Synthesized attributes are very common in practice
  - Bison uses this type of attribute
- A syntax-directed definition that uses synthesized attributes exclusively is said to be an *S-attributed definition*
- A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom-up

Example: Annotated parse tree for 3*5+4n

```
L
|   +
E.Val = 19  T.Val = 4
|   |
E.Val = 15  T.Val = 15
|   |
T.Val = 15  F.Val = 4
|   |
T.Val = 3   F.Val = 5
|   |
F.Val = 3   digit.lexval = 4
|   |
digit.lexval = 3
```
Inherited Attributes

- Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears.

Example

- Using inherited attributes to parse a declaration and add type information to the symbol table.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → TL</td>
<td>L.in := T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.Type := integer</td>
</tr>
<tr>
<td>T → real</td>
<td>T.Type := real</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in := L.in addtype (id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addtype (id.entry, L.in)</td>
</tr>
</tbody>
</table>
Bottom-up Parsing

- If all attributes are synthesized then a bottom-up parser can evaluate the attributes while it parses
- Add attribute fields to the nonterminals on the stack
- Before each production apply the semantic rule associated with the production
- E.g. Assume the semantic rule associated with the $A \rightarrow XYZ$ production is: $A.a := f(X.x, Y.y, Z.z)$
Bottom-up Parsing

- Before the reduction the parser stack looks like:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X.X</td>
</tr>
<tr>
<td>Y</td>
<td>Y.y</td>
</tr>
<tr>
<td>Z</td>
<td>Z.Z</td>
</tr>
</tbody>
</table>

- After the reduction the parser stack looks like:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A.A</td>
</tr>
</tbody>
</table>

This is the method that Bison uses for maintaining attributes
Bison Generated Parsers

- Bison maintains a semantic stack which, in tandem with the parser stack, helps Bison manipulate and determine the meaning of the program it compiles.
- The values in the semantic stack represent the *values or meanings* of all the grammar symbols.
- Exactly what constitutes *values* for the entries in the semantic stack depends heavily on the semantics of the language being compiled.

Semantic Stack

- Manipulation of the semantic stack is triggered by the parser during *reduce* moves.
- *Values* of symbols on the rhs of the production are popped off the semantic stack.
- A *value* for the symbol on the lhs is determined from these values and the semantic rules of the language, and a new value is pushed back on the stack.
Semantic Stack

- All symbols in the grammar have a value associated with them in the semantic stack even though some of these values may be NULL (indicating no information is needed about this symbol)
- Not all reductions require semantic actions

Semantic Rules

- Bison allows you to specify the semantic action associated with the symbol on the LHS of the production
  - These actions may return a value and may utilize the values returned by previous actions
- The lexical analyzer can return values for tokens, if desired, through the use of the global variable yylval
Semantic Rules

- An action is specified by writing arbitrary C/C++ statements enclosed in {}
  
  Exp : Exp PLUS Exp
  
  \{ $$ = binaryOp( \$1, A_Plus, \$3); \}
  
  - To return a value for the symbol Exp on the lhs, a value is assigned to $$
  
  - To refer to values of symbols on the rhs, the variables $1, $2, …, are used
  
  ▪ These refer to the values returned by the components of the rhs of the production reading from right to left

Semantic Actions

- If no action is specified after a particular production, the default action
  
  \{ $$ = $1; \}
  
  is performed
Union

- The values on the semantic stack do not have to be of a single type
  - You can define the type of the value using the `%union` statement in Bison
- Bison must know which type to associate with each symbol in the grammar
  - This is done using the `%type` statement
    - Beware, once you starting using `%type`, you are forced to type all symbols even though you may not be using them
  - You are required to use `%type` statements in your project

%union and %type

```
%union
{
  int int_val;
  char *name_ptr;
  CONS *cons_ptr;
  ID *id_ptr;
  TYPE *ty_ptr;
  EXPR *exp_ptr;
}

%type <int_val> INTCONSTS YfHead
%type <id_ptr> IdList FieldList
```