Lab 4 Abstract Syntax Tree

**Purpose:**

In this lab you will create an Abstract Syntax Tree (AST) as you parse. The AST should contain all the relevant information from the source file so that later stages of compilation can do their work without referring to the source file.

**Process:**

For this lab, we will use bison – a parser generator. The bison input file defines the CFG. Each production can have a code snippet that gets executed each time the production is used in a reduction. The primary purpose of the code snippet is to build a value for the Non-Terminal on the left hand side of the production. For our purposes, it will construct an AST node to represent the left hand side. To construct this node, you need values for items on the right hand side. bison gives you variables that you can use to access these values.

Consider the following production:

stmt: WHILE '(' expr ')' stmt

The right hand side contains five items. Each of these items have a bison variable that you can use to access its value. The variables are called $1, $2, … $5. The bison variable for the left hand side is $$. The code snippet to construct an AST node for a while statement might look something like this:

$$ = new WhileNode();

$$->SetExpr($3);

$$->SetStmt($5);

// or...

$$ = new WhileNode($3, $5);

Where do the right hand side variables get their values? There are two possibilities. If the variable represents a terminal, the value is assigned in yylex() (via the code in lang.l). If the variable represents a non-terminal, the value is assigned by a code snippet for a previously invoked production. In the example above, since this is a bottom-up parser, the expr and stmt have already been found, so the code snippets for those productions has already been executed. The $3 that represents the expr comes from the $$ for the code for the production that recognized the expr.

**Process Part 2:**

You will need to define your AST nodes. Your AST nodes must be subclasses of the cAstNode I provided in the tar file. Your nodes must have appropriate constructors, and getters and setters for the various bits of information stored in your node. Any children of your node MUST be stored in the m\_children vector defined in cAstNode. You can use the AddChild method for this. Finally, you must override the appropriate virtual functions.

In the Output section of this file, I detail the “correct” output for various parts of the grammar. Most of the work for creating the correct output is handled by the ToString in cAstNode. Some of the virtual functions control the behavior of ToString, so you will have to override them to produce the correct output.

You do not need to define all your node types at once. The test suite is set up so that you only need a minimum number of nodes to pass the test. Later tests add more nodes. Start with the bare minimum required to get Test 1 to pass. Once you get Test 1 to pass, you should be on the right track to finish the lab.

Note: I supplied a solution for Test 1 in the tar file. Be sure to study this solution as it gives you clues on how to work on the rest of the tests.

I’d suggest you use the following process:

1. Move your IncreaseScope and DecreaseScope calls from the scanner to the parser. The lang.y file I gave you gives hints as to where to do this.
2. Delete tokens.h and include the automatically generated langparse.h instead.
3. Pick what test you are going to work on (start with Test 1)
4. Identify what new non-terminals are used by that test
5. Create node types for those non-terminals. Add appropriate includes (including in astnodes.h)
6. Modify the %union and %type statements in lang.y to reference the new node types
7. Add code snippets in the productions for the new non-terminals to create appropriate nodes using your new node types.
8. Repeat steps 3-7

**Some Details:**

Last week, you used a symbol class. That class must now be a subclass of cAstNode. The ToString method supplied in last week’s lab for cSymbol can be removed. It is superseded by the ToString in cAstNode.

The standard types (char, int, float) must have symbols defined for them. You must insert the symbols into the outermost scope before you start parsing. When encountering one of these keywords, the scanner must set yylval to point to the correct symbol. You can wait until Test 4 to add these symbols.

When you create symbols for struct declarations and array declarations, the symbols must be marked as types. To do this, I’d suggest you and a field to your cSymbol class and add a getter and setter for that field. In lang.l, whey you encounter an identifier, if the symbol already exists, you need to return either IDENTIFIER or TYPE\_ID based on the field you added. This will become necessary in test8.

**Output:**

The parse function will return the root of the AST to the caller (on success). The caller (main) will print the AST. This happens by calling ToString on the root node of the AST. This will recursively call ToString on all nodes in the tree. The output is in XML. The xmllint utility is used to pretty print the XML prior to comparing against the correct output. By using xmllint to pretty print, you don’t need to worry much about spacing and newlines in your output. xmllint should standardize the spacing.

The following table shows the definitions of the XML tags used in the output. If a tag ends with “\>”, then the tag is an inline tag. If the tag for a node contains attributes, then the class for that node must override the AttributesToString function to return the attributes in the order specified below (see cIntExprNode.h in the tar file for an example of how to do this). In general, inline tags have attributes, and non-inline tags don’t. ToString chooses to do an inline tag or not based on whether the node has children. If the node does not have children, it will generate an inline tag.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Node | tag name | Inline | Attributes | Sample | Children |
| cArrayDeclNode | array\_decl | no | size | <array count=”5”> | cSymbol (type), cSymbol (name) |
| cAssignNode | assign | no | none | <assign> | cVarExprNode, cExprNode |
| cAstNode |  |  |  | never appears directly |  |
| cBinaryExprNode | expr | no | none | <expr> | cExprNode, cOpNode, cExprNode |
| cBlockNode | block | no | none | <block> | cDeclsNode, cStmtsNode |
| cDeclNode |  |  |  | never appears directly |  |
| cDeclsNode | decls | no | none | <decls> | cDeclNode (multiple) |
| cFloatExprNode | float | yes | value | <float value="1.2" /> | none |
| cFuncDeclNode | func | no | none | <func> | cSymbol (type), cSymbol (name), cParamsNode (params), cDeclsNode (locals), cStmtsNode |
| cFuncExprNode | funcCall | no | none | <funcCall> | cSymbol (name), cParamListNode |
| cIfNode | if | no | none | <if> | cExprNode, cStmtsNode (if), cStmtsNode (else) |
| cIntExprNode | int | yes | value | <int value="15" /> | none |
| cOpNode | op | yes | value | <op value='\*' /> | none |
| cParamsNode | Args | No | Node | <args> | cDeclNode (multiple) |
| cParamListNode | params | no | none | <params> | cExprNode (multiple) |
| cPrintNode | print | no | none | <print> | cExprNode |
| cProgramNode | program | no | none | <program> | cBlockNode |
| cReturnNode | return | no | none | <return> | cExprNode |
| cStmtNode |  |  |  | never appears directly |  |
| cStmtsNode | stmts | no | none | <stmts> | cStmtNode (multiple) |
| cStructDeclNode | struct\_decl | no | none | <struct\_decl> | cDeclsNode, cSymbol (name) |
| cSymbol | sym | yes | id, name | <sym id="5" name="temp" /> | none |
| cVarDeclNode | var\_decl | no | none | <var\_decl> | cSymbol (type), cSymbol (name) |
| cVarExprNode | varref | no | none | <varref> | cSymbol (multiple – for struct field refs) |
| cWhileNode | while | no | none | <while> | cExprNode, cStmtNode |