

### 3. Decorating a Parse Tree and Building a Symbol Table

#### Coursework and Goals for This Part of the Course:

- By the end of this section, you will be able to...
  - discuss issues that revolve around names in programming languages
  - describe the four common kinds of storage location for variables, giving examples from languages such as C++ and the CSCI 3155 Language
  - explain the difference between a variable's lifetime and its scope
  - determine precisely which variable declaration will be used for an identifier in any given program (for either static or dynamic scoping)
  - determine whether examples of common compiler actions (such as binding) are primarily static or dynamic
  - identify alias in languages such as C++ and the CSCI 3155 Language
  - use the tree and symbol\_table classes to build a parse tree that is decorated with the attributes defined in Section 3.6 of this guide
- Read Sebesta §3.4, 5.3, 5.4, 5.8, 5.9
- Lecture topics:
  - What we have now: The lexer and the parser
  - A side-trip to Chapter 5:
    - Names
    - Variables
      - Name
      - Storage location
        - Static
        - Stack dynamic
        - Explicit heap dynamic
        - Implicit heap dynamic
      - Data Type
      - Lifetime
      - Value
      - Scope
  - Binding time
  - Scoping
    - Static scoping
    - Dynamic scoping
  - Our tree class
  - The lexer's role in building a parse tree
  - The parser's role in building a parse tree
  - Our symbol table class
  - The traverser
    - Maintaining the symbol table
    - Attaching attributes to the parse tree
  - Homework discussion including using a makefile
- Bison Reference: [www.delorie.com/gnu/docs/bison/bison\\_toc.html](http://www.delorie.com/gnu/docs/bison/bison_toc.html)
- Homework 3: Building and Traversing the Parse Tree

### 3.0. Names, Variables, Bindings, Scope

Before diving into the material for this guide, you'll need to be familiar with the concepts from Chapter 5 of Sebesta's book. A partial list of those concepts is given in the lecture topics on page 3.1 of this guide.

#### 3.1. A Tree Class

After completing Homework 2, you have a parser function, `yyparse`, that determines whether an input file is a valid CSCI 3155 Program. Our next task is to modify `yyparse` so that when the input file is valid, the parser will build a parse tree. At the end of a successful parse, we will have a pointer to a tree that is from the `colorado::tree` class in the `tree.h` and `tree.cxx` files in [www.portmain.com/proglang/CU++/Part3](http://www.portmain.com/proglang/CU++/Part3).

Here are some things to note as you read through the documentation of the tree class:

- 3.1.1. The tree class is part of the `colorado` namespace. Therefore, when you declare a pointer to a tree variable, you must use the fully qualified name `colorado::tree*` or make sure that you have written the directive: `using namespace colorado;` Keep in mind that you should never put a `using` directive in a header file.
- 3.1.2. The tree class doesn't allow an empty tree (with no nodes). Every tree must have at least one node, its root. It may also have zero or more subtrees. The number of subtrees is obtained through the `many_children` member function. A pointer to any specific subtree is obtained through the `child` member function.
- 3.1.3. Every node of a tree has a string label. The label of the root of a tree is set through the `set_label` member function and retrieved through the `label` member function. In the case of our parse trees, most of the labels will be nonterminal symbols from the grammar, such as "`<stmt>`" or "`<expr>`". For a terminal node, however, we will use the lexeme (such as "`while`" or "`42.8`") as the label.
- 3.1.4. Every node of a tree may have attributes of various types attached to it. Each attribute has a string called the *key* that is first used to set the attribute and later used to retrieve a copy of the same attribute. For example, suppose that `p` is a pointer to a tree. This line of code sets an integer attribute of the root of that tree using a key of "`Line`" and a value of 42:

```
p->set_attribute<int>("Line", 42);
```

Notice that the data type of the attribute (`int`) must appear as part of the function name, `set_attribute<int>`. Later, we can retrieve the `Line` attribute with the `attribute` member function. For example:

```
// Prints the Line attribute of the root of *p:  
cout << p->attribute<int>("Line");
```

## 3.2. Lexer Modifications

The first step in building a parse tree is to build a one-node tree each time that the lexer finds a token. This one-node tree contains information about the token and will later be a leaf in the complete parse tree. The easiest place to build these tiny trees is in the lexer itself. Flex supports this approach by allowing us to attach a single data value to each token that is found and returned by `yylex`. Here are the required steps:

- 3.2.1. Tell flex the data type of the values that will be attached to each token. This is done with a `#define` directive that defines a data type called `YYSTYPE`. In our case, we will attach a tree pointer to each token, so the required definition is:

```
#define YYSTYPE colorado::tree*
```

This definition occurs in the C++ prologue of the flex specification after including `tree.h`. It must occur before including `cs3155.tab.h` (otherwise `YYSTYPE` will be defined as an integer in `cs3155.tab.h`).

- 3.2.2. Each return statement in your lexer (that returns a token number) must now be preceded by C++ code that builds the one-node tree and saves a pointer to this tree in a special variable called `yylval`. (Flex defines `yylval` to be a variable of the type `YYSTYPE`.)

The one-node trees that you build will each have the token's lexeme as its label. This string can always be obtained from a flex-maintained variable called `yyltext`, and it must be used as the first argument for the tree constructor that you call.

This means that each pattern/action in your flex specification file will be expanded to build the one-node tree, such as:

```
while      Code to set yylval to point to a new one-node tree; return WHILE;
```

The code that builds the tree and sets `yylval` may be placed right in the pattern/action, just before the return statement. However, if the action code spans more than one line in the flex specification file, then the first line must be an open curly bracket, and the last line (right after the return statement) must be a close curly bracket.

Suggestion: Write a function that's responsible for building the one-node tree. Put a prototype for this function in the C++ Prologue, and put the implementation of the function in the C++ Epilogue of the flex specification file. Then you can simply call this function before each return statement in the pattern/action pairs.

### 3.3. Parser Modifications

In order to build a parse tree, the parser must also be modified so that every time it performs a reduction, it also builds a piece of the parse tree. These parse trees are attached to each nonterminal as it is parsed. Here are the required steps:

- 3.3.1. Tell bison the data type of the values that will be attached to each nonterminal. This is done with the same `#define` of `YYSTYPE` that you used in the lexer modification.
- 3.3.2. Each grammar rule in your parser must now be followed by C++ code that builds a parse tree and attaches it to the nonterminal that was just parsed. The C++ code to build this parse tree must appear in curly brackets on the lines that immediately follow each grammar rule.

Within this C++ code, you may use the unusual bison variable names `$1`, `$2`, `$3...` to get the pointers to the trees that have already been built for the symbols on the right side of the rule. For example, in the rule `<parmdefn> → INOUT <typeexpr> IDENTIFIER`, the variable `$1` is a pointer to the one-node tree that the lexer built for the `INOUT` token. The variable `$2` is a pointer to the tree that the parser built itself for the `<typeexpr>` nonterminal. The variable `$3` is a pointer to the tree that the lexer built for the `IDENTIFIER` token.

Your code must set one other special variable called `$$`. This variable must be set to point to the portion of the parse tree that you are building for this reduction. For example, here is the code to build part of the parse tree for the `<parmdefn>` grammar rule, using the string `"<parmdefn>"` for the label of its root:

```
parmdefn      : INOUT typeexpr IDENTIFIER
              {
                $$ = new tree("<parmdefn>", 3, $1, $2, $3);
                ...and code to set attributes
              }
              ;
```

By the way, I found it easiest to write a function that's responsible for setting the attributes of the newly created tree. This function is called at the spot that says "...and code to set attributes."

- 3.3.3. When the start symbol (`<program>`) is parsed, you will have created the whole parse tree. At this point, you must set a global variable named `parse_tree_root_ptr` to point to this whole tree. Some questions you'll need to answer: What is the data type of `parse_tree_root_ptr`? Where will you declare it? In the bison specification file, where will it get its assigned value?

### 3.4. The Symbol Table

After the parse tree is built, we'll do a traversal to carry out a variety of checks. The traversal will be done by a function, `void traverse( )`, that you write in a file called `cs3155.traverser.cxx`. During the traversal, we'll maintain a *symbol table*, which is a data structure from `syntab.h` and `syntab.template` in [www.portmain.com/proglang/CU++/Part3](http://www.portmain.com/proglang/CU++/Part3)

Here are some things to note as we discuss the symbol table class during lectures:

- 3.4.1. The symbol table is a template class. Your traversal program will maintain a single global symbol table variable (named `st`) of type `colorado::symbol_table<tree*>`. This kind of symbol table can keep track of identifiers that have been defined in a program. Each identifier can have a pointer to a tree (a `tree*`) associated with it as its value (because `tree*` is the value of the template parameter). For our traversal, the associated `tree*` will always be the definition tree for the identifier (which has a nonterminal symbol such as `<vardefn>` or `<funcdefn>` at its root).
- 3.4.2. As you traverse the parse tree, each time you reach a definition (such as a variable definition), you will insert the definition's identifier into the symbol table along with a pointer to the whole definition tree. For example, to insert a function definition:

```
st.insert(the_identifier_name, funcdefn__, the pointer to the defn tree);
```

The second argument in this function call tells what kind of definition is being inserted. It may be any of these lhs values from `cs3155.enum.h`: `funcdefn__`, `vardefn__`, `parmdefn__`.

- 3.4.3. Before you can use the symbol table, you must register the different kinds of identifiers that will be put in the table. The registration is done with this member function:

```
void register_kind(
    int kind,
    int start_offset,
    int delta,
    bool reset_for_new_scope
);
```

The first parameter, `kind`, is one of the lhs values, such as `vardefn__`. The rest of the parameters provide information about the storage binding of that particular kind of item, as follows:

The `start_offset` tells where items of this kind will start within a particular memory section. For example, variables will be stored in a function's frame starting at an offset of `-4` bytes from the start of the frame; so `start_offset` is `-4` for a `vardefn__`.

The `delta` value tells how far the second item of this sort will be from the first. For example, each variable in a function's frame is stored four bytes below the previous variable, so `delta` is `-4` for a `vardefn__`.

The Boolean value, `reset_for_new_scope`, indicates whether a new memory area is created for this kind of item whenever a new scope is entered during the traversal. This is true for the `vardefn__` and `parmdefn__` only, not for any of the other types of definitions.

Here are the complete arguments for our particular kinds of definitions:

```
st.clear( );
st.register_kind(funcdefn__, 0, 1, false);
st.register_kind(vardefn__, -4, -4, true);
st.register_kind(parmdefn__, 16, 4, true);
```

- 3.4.4. During the tree traversal, whenever you enter a new scope, you must activate `st.enter_scope( )`. When you leave the scope, you must activate `st.exit_scope( )`. This allows the symbol table to correctly maintain information about the storage binding of variables.

Note: You enter the global scope before starting the traversal, and exit at the end. You enter a function (or procedure) scope immediately after inserting the function (or procedure) definition into the symbol table. You then traverse all of the subtrees of the definition. After this, you exit the scope.

### 3.5. Tree Attributes

As you are traversing the parse tree, you will be maintaining the symbol table. You must also add certain attributes to each node. Here is a list of the attributes that must be set. The names and data types of these attributes (used in to the tree's `set_attribute` method), must be exactly as shown in this list.

#### 3.5.1. *Addressable*

Data type of this attribute: `bool`

Nodes that have this attribute: `<expr>`

Computed by: `traverser`

For now, just set this to `false` for all expression nodes.

### 3.5.2. *Bytes*

Data type of this attribute: int

Nodes that have this attribute: <defnlist>, <parmseq>, <program>

Computed by: traverser

For a definition list, this is the total number of bytes required by its variables. For a <parmseq>, this is the total number of bytes required by its parameters. For a <program>, this is the total number of bytes required for its global variables.

### 3.5.3. *Definition*

Data type of this attribute: tree\*

Nodes that have this attribute: all IDENTIFIER nodes

Computed by: traverser

This is a pointer back to the definition parse tree (for example <vardefn> or <funcdefn>) for this identifier. It can be obtained from the symbol table.

### 3.5.4. *Depth*

Data type of this attribute: int

Nodes that have this attribute: all IDENTIFIER nodes

Computed by: traverser

This integer tells the depth at which the identifier was declared. Any definition at the global level of the program has a depth of zero. Entering a scope increases the depth of new definitions by one; exiting a scope returns the depth to its previous level. The value of the Depth can always be obtained from the symbol table.

### 3.5.5. *Errors*

Data type of this attribute: int

Nodes that have this attribute: all nodes

Computed by: traverser

This integer is the total number of errors that the traverser finds at this node or below it. For now, we are detecting only a few errors, each of which must call a function that prints an appropriate error message to cerr. If you pass the node as a parameter to this function, then the function could also increment the Errors count. The current possible errors are:

- Internal compiler error (such as an expr node that has an unexpected value for its LHS attribute). These should not occur, but it's useful to have a message that we can use.
- Unknown type name. The built-in data types that we have are |int|, |float|, |bool| and |string|. Any other type name that appears must be defined in the current scope.
- Unknown identifier. This indicates that an identifier was used in an expression, but is not available in the current scope. Note: Within the expression that initializes a variable, that variable itself may not be used. This means that you must not insert a variable definition into the symbol

table until after you have already traversed the initialization expression for that variable.

- Duplicate identifier declared. The same identifier was declared two or more times in the same scope.

### 3.5.6. *Kind*

Data type of this attribute: lhs

Nodes that have this attribute: all IDENTIFIER nodes

Computed by: traverser

This value is one of the kinds of an identifier, such as `vardefn__`. It tells what kind of definition was used to declare this identifier. Its value can be obtained from the symbol table. If the identifier is not declared, then please set this attribute to the constant `ILLEGAL_NONTERMINAL` from `csx3155.enum.h`.

### 3.5.7. *LHS*

Data type of this attribute: lhs

Nodes that have this attribute: every nonterminal node

Computed by: parser

This value the left-hand side of the grammar rule that was used the build this part of the parse tree.

### 3.5.8. *Line*

Data type of this attribute: int

Nodes that have this attribute: all nodes

Computed by: lexer or parser

This integer tells the approximate line number where the token or larger structure appears. In my lexer, I use the flex-maintained variable `yylineno` to set this value in the one-node token tree that is built for each token. In my parser, I use the `Line` attribute from the first child (if there is one) or I use `yylineno` (if there are no children).

### 3.5.9. *Offset*

Data type of this attribute: int

Nodes that have this attribute: all IDENTIFIER nodes

Computed by: traverser

This integer tells where the identifier resides in memory (as an offset from the start of its memory block). Its value can be obtained from the symbol table.

### 3.5.10. *RHS*

Data type of this attribute: rhs

Nodes that have this attribute: every nonterminal node

Computed by: parser

This value the right-hand side of the grammar rule that was used the build this part of the parse tree.

### 3.5.11. *Token*

Data type of this attribute: int

Nodes that have this attribute: all terminal nodes

Computed by: lexer

This integer is the token number for the node.

### 3.5.12. *Type*

Data type of this attribute: const tree\*

Nodes that have this attribute: <expr> nodes

Computed by: traverser

For now, just set this to the NULL pointer, but make sure that you do this *after* setting the Type of any subexpressions.

## 3.6. Homework: Building a Traverser

There is no programming homework for this section. However, you are responsible for understanding the way that I modify the lexer (see Section 3.3 of this guide), modify the parser (Section 3.4) and build a traverser (as described in Section 3.5). The traverse function must be a void function with no parameters in a file called `cs3155.traverer.cxx`. It traverses the tree that `parse_tree_root_ptr` points to. I will put my work in the directory [www.portmain.com/proglang/CU++/Part3](http://www.portmain.com/proglang/CU++/Part3). This directory includes a small test program (`test-parse2.cxx`).

Note: the attributes rules must use the exact names from Section 3.5 (including capitalization). Any error messages must be printed exactly as shown in Section 3.5.5. In general, please don't make any changes to the specification. If you need clarifications, of any of the work that I do, please check with the instructors.

Once I have compiled the `test-parse2`, you I run it with any of the sample CSCI 3155 programs from [www.portmain.com/proglang/CU++](http://www.portmain.com/proglang/CU++). However, don't assume that the test programs form a complete test set for your parser. Note: The Makefile actually creates two executable versions of `test-parse2`. One version prints just the tree; the other also prints attribute information.

## 3.7. Exam Practice

Ex 3.1. Describe the design issues for names in programming languages.

Ex 3.2. Give an example of an attribute of a variable that has a static binding time. Give another example of an attribute that has a dynamic binding time.

Ex. 3.3. In C++, give examples of variables that have each of the four different types of storage binding.

Ex. 3.4. What is the difference between the lifetime and the scope of a variable?

Ex. 3.5. What is the output of the following CSCI 3155 program:

```
global :integer: x(0);

procedure f( )
{
    var :integer: x(1);
    write x;
    g( );
}

procedure g( )
{
    write x;
}

function :integer: main( )
{
    f( );
    return 0;
}
```

Ex. 3.6. Explain how and why the output of the previous program changes if we use dynamic scoping.

Ex. 3.7. Consider the two declarations of x in Exercise 3.5. For each case, what are all the properties of x's symbol table entry?

Ex. 3.8. Are the entries from Exercise 3.7 put into the symbol table before or after the initialization values (0 and 1) are traversed by the traverser?

Ex. 3.8. Define all the attributes that will be attached to the identifier x that occurs in the body of g for Exercise 3.5? Also define the values of each of these attributes.

Ex. 3.9. Modify the program from Exercise 3.5 so that there is a third declaration of x with a Depth of 3.

Ex. 3.10. In the CSCI 3155 programming language, how many bytes are required for the storage of a global variable? How many for an ordinary local variable of a function? How many for a static variable that's defined in a function.

Ex. 3.11. Write a CSCI 3155 program with a procedure called annoy. The first time that annoy is called, it prints zero. Each subsequent time the annoy procedure is called, it prints a larger integer than the previous time. Do not use any global variables.

Ex. 3.12. What kind of CSCI 3155 variables are stored on the run-time stack? What kind of CSCI 3155 variables are stored in the code segment or data segment? When does a

CSCI 3155 program use explicit heap-dynamic memory? When does it use implicit heap-dynamic memory?