CSE 504 Programming Assignment #3

Abstract Syntax Tree

Read the following description carefully before starting on the assignment. The tips section (Section 7) may be especially useful for this project. A list of what materials are available (Section 8) and what you are expected to submit (Section 9) are also given. Finally, make sure you understand and follow the submission (Section 10) requirements.

1 Description

In this assignment, you will build the abstract syntax tree (AST) for the `E--` compiler. The structure of the AST is given below. You will build the AST by adding actions to a parser for `E--`. The lexical analysis module that you will need for this project is also provided. In addition, symbol table related sources are provided, as are several other utility sources.

2 Overview

The purpose of this assignment is to construct an abstracted representation of the input program that can later be processed by subsequent phases of compilation, such as type checking and code generation. This abstract representation, called the *Abstract Syntax Tree (AST)*, is an abstracted version of the parse tree which retains only information that is relevant for processing by the subsequent phases. (For instance, parenthesis serve no purpose in a tree representation, and hence need not be represented in the AST.)

In effect, the action associated with each grammar production becomes the code for constructing the AST corresponding to the lhs nonterminal of that production, given the AST's of the nonterminals on the rhs. Constructing such an intermediate structure yields considerably more flexibility and structure in the way a compiler is implemented.

The information needed by subsequent phases of compilation can be classified into three major categories:

- *values* that can be assumed by the variables in a program. These can be of primitive types such as integer and boolean, or user-defined types. Values are captured by the `Value` class and its subclasses (if any).
- *symbols* that are defined in the program, such as events, functions and variables. Information relating to various symbols are captured by the `SymTabEntry` class and its subclasses. The `SymTabEntry` objects are stored in *symbol tables.*
- *executable code* that is captured in `AstNode` class and its subclasses.

These categories are further described below.

3 Values

The different types of values in `E--` are identified by a `TypeTag` defined in `Type.h` file. The type information itself is stored in a class `Type`, which consists of a type tag and a type description. A complete implementation of this class is provided to you, so you dont have to implement this class. You will, however, need to understand the code so that you can use it correctly.

The `Value` class needs to store a tag representing the type of the value, and the actual value itself, which will differ depending upon the type. In C++, we can either use a union to represent the different types of values, or create different subclasses to hold values of different types (e.g., `IntValue`, `FloatValue`, `StringValue`, etc.) Given the simplicity of the `Value` class, the additional flexibility and power offered by the subclassing approach is not that helpful.

In this assignment, you will use the `Value` class to store the values of the constants (identified by the grammar nonterminal **literal**) that are processed by the parser.

4 Symbols

The different classes of symbols are identified by a tag drawn from the following enumerated datatype:

```
enum Kind {UNKNOWN_KIND, GLOBAL_KIND, CLASS_KIND, FUNCTION_KIND, VARIABLE_KIND, EVENT_KIND,
          RULE_KIND, BLOCK_KIND, RULE_BLOCK_KIND, EVENT_BLOCK_KIND, ERROR_KIND};
```

No visibility related qualifiers are used.

In this assignment, you will be provided a generic `SymTabEntry` class that contains information common to the different types of symbols. You should well understand the usage of `SymTab`, `SymTabEntry`, and `SymTabMgr` class before you start other parts of implementation. This implementation of `SymTabEntry` has to be refined in order to include additional information that is specific to each symbol type. We will do this by defining one subclass of `SymTabEntry` for each type of symbol. Specifically, we will need to define `GlobalEntry`, `BlockEntry`, `RuleBlockEntry`, `VariableEntry`, `ClassEntry`, `FunctionEntry`, and `EventEntry`. Further details about these classes can be found in the header file `STEClasses.h`.

In this assignment, you will have to provide actions in the Bison input file that will construct the appropriate `SymTabEntry` objects when the different types of symbols are processed, and enter them into the symbol table corresponding to the current scope. You will have to manage the scopes as well — recall that the entry and exit from different scopes was handled by the driver program in your first assignment. In this assignment, your parser action code will make these calls. The following scopes are to be supported:

- Function scope spans from the beginning of its formal parameter declarations to the end of function body (if present).
- Event scope is entered immediately after paarsing the event name in an event declaration. (No scopes are to be entered while parsing event names within rules.)
- Each rule defines its own scope. This scope should be entered on parsing the first event name (or TOK_ANY) in the rule, which will be part of an event pattern.

5 Abstract Syntax Tree

We will build abstract syntax trees only for paterns, statements and rules that occur in a program. The different classes of expressions and statements are stored in various subclasses of the `AstNode` class. These subclasses are described in `Ast.h`.

In order to manipulate the different kinds of nodes uniformly, we will define a bunch of virtual functions on the base class **AstNode**. Of these, you will be providing only the `print` function for this assignment. In subsequent assignments, you will supply the remaining virtual functions such as `typeCheck`, `codeGen` or `execute`. Use of this design will considerably simplify the structure of your compiler and will make it much easier to complete your next few assignments.

6 Important Notes

6.1 Handling of Names

Before the construction of the AST, all names are resolved into pointers to appropriate entries in the symbol table that correspond to these names. This means that every symbol must be declared before use in E--. Use of undeclared symbols should result in meaningful error messages.

6.2 Managing Scopes

You need to ensure that appropriate calls to `SymTabMgr::enterScope` and `SymTabMgr::leaveScope` are issued when entering and leaving various scopes. Of particular significance is the rule scope, which should be entered at the beginning of a rule and exited at its end. Note, however, that there is no easy way to know when you are at the beginning of a rule. One way to recognize this is when you parse the first event in a rule. That you are looking at the first event can be identified by the fact that at that time you are in the global scope. (For subsequent events, you would have already entered the rule scope.)

6.3 Error Handling

Your program is expected to behave reasonably in the face of syntactic and semantic errors. You will not be detecting too many errors of the latter kind for this assignment – most semantic errors are left to be detected by the type-checking phase. However, you should detect and deal with duplicate definitions of symbols in this assignment.

Your program is expected to continue parsing and constructing AST even after errors have been detected. Use NULL pointers in place of AST subtrees that could not be constructed due to the error. I would expect the print routines to print something reasonable even for programs containing errors.

6.4 Dealing with Declarations

Note that we will not be building any ASTs for declarations. Processing of declarations should simply result in construction of a SymTabEntry object and its insertions into the current scope.

6.5 Naming Conventions in Code

I use the following conventions for names in C++, and suggest that you do the same in order to make it easy to read your code:

- constants and enum values use all-caps names, e.g., REF_EXPR_NODE. The words are separated by _.
- types are represented by mixed-case names that start with a capital letter. Each word within the name starts with a capital.
- variables and member functions are represented by mixed-case names that start with a lower-case letter.
- member variables use mixed case, start with a lower-case and end with an '_', e.g., `typeTag_`.

6.6 Interface with the Driver

The driver program has the same name as before, but it has a couple of lines more of code that call the `print` function on `GlobalEntry`. This call should result in printing of the entire program.

6.7 Printing the AST

To simplify comparison with the standard output, please try to write your print method implementations in such a way that they match the standard output closely. Redundant spaces are usually not a problem, since you can use the `-w` option to `diff` to suppress flagging discrepancies due to whitespace characters. While printing binary expressions, please make sure that parenthesis are used. In particular, an expression $a + b + c$ should be printed as $((a + b) + c)$.

If you want to look at the output, it is useful to indent it as you normally indent your programs. This can be accomplished by passing a second argument `indent` to all of the print functions. These functions should make sure that they print `indent` spaces before they start printing any thing on a new line. Moreover, this number should be incremented before printing a nested block, and decremented on return. Some of the print-related code is already provided to you. You don't need to completely match the amount of indent to the reference output sequences in test, though.

6.8 Parameter Passing Conventions

In general, most constructors will take pointers as arguments, and store the pointers. Our convention is that the constructed class becomes the "owner" of this pointer, and needs to manage it, and the caller of the constructor will not use the pointer any more. This approach will avoid repeated copying of the AST data structures, which makes things faster as well as simpler (because we do not have to implement the copy constructor or assignment operation).

7 Tips

- As usual, start small. Try building AST's for a small portion of the grammar. Expressions with only integer constants is a good place to start.
- Given the classes we have defined for different kinds of symbol table entries, values and AST nodes, there needs to be very little code in the action part of the rules. In the most common case, the body would consist of just a single call to a constructor for the object of appropriate type. For instance, the code for the rule **Expr** $\rightarrow$ **Expr** + **Expr** looks something like `$$ = new BinOpNode(PLUS, $1, $3);` Overall, I expect only about 200 lines of action code for the entire grammar.
- Unlike previous assignments, a considerable amount of code is already given to you for this assignment – abut 2500 lines of code. In comparison, if you write your code carefully, you will nee to write only 500 to 800 lines of code. As such, please devote a good fraction of your time to reading and understanding the code given to you.

8 Available Material

- The header and implementation files for symbol-table related routines. These include `SymTabMgr.h`, `SymTabEntry.h`, `SymTab.h`, as well as the corresponding C-files.
- The lexical analyzer code in `E--_lexer.C`
- A sample `Makefile`: this assignment makes use of some of the features added to C++11. So to compile the source files, you will need a compiler that supports C++11 features. We have tested the source code on GNU C++ compiler version 4.6.3 and LLVM-3.3 Clang++ compiler. If your system does not have either of these compilers, or compilers of higher version than above mentioned, then please make every effort to get them. If you still cannot get them, then please let us know.
- The header file `STEClasses.h` for subclasses of `SymTabEntry` class.
- The header file `Value.h` and `Value.C` that implements this class.
- The header file `Type.h` and `Type.C` that implements this class.
- The header file `Ast.h` that defines the `AstNode` class and its subclasses.
- The parser specification file `E--_parser.y++`. This file also contains the appropriate `%union` and `%type` declarations you need.
- The class `ProgramElem` is a base class for both `SymTabEntry` and `AstNode`, and its header file (including its implementation) is in `ProgramElem.h`.
- A few other miscellaneous files are provided as well, and these provide a few utility functions.
- Sample input files `in*` and the respective driver output files `out*` are in `tests` or `errtests` subdirectory.

9 Deliverables

- a Makefile that builds the module and links it with all of the provided source files, libraries and your own implementation files to create an executable called "`demo`".
- The implementation files `STEClasses.C` and `Ast.C`. I don't anticipate needing any changes to the header files, so unless you give a good reason why, your .C files should compile with the header files given to you.
- The modified `E--_parser.y++` file with the action components filled with code that takes appropriate actions to build the AST corresponding to each of the productions.
- Any other files that are necessary to successfully recompile your program.

- If you found any errors with the sample output that we have provided, or if you have made some assumptions while writing your code, then make sure that you write a README file about such things. We will consider it during grading. But be sure that the errors that you found or the assumptions that you made are reasonable and logical. Any unreasonable or illogical errors and assumptions will not have any positive effect on the grading. Instead, we might reduce some points from your score.

It is absolutely important that you NOT change the other files given to you. If your code does not compile with the files we give you, then you may not get any credit for your submission.

10 Submission Requirements

Please note the following submission requirements. These are not simply conventions, but are **requirements**. They make grading of the projects easier, and lets us spend more time wading through the code to award partial credit!

- The Makefile must be such that it can successfully recompile your program with only the files you submit. Make sure that there are no absolute references in your Makefile to files in your own directory.
- You **will not** be submitting .o files for any module. Therefore, you should submit all source files needed to compile your program. If you use modules such as those for the earlier assignments, **you MUST ensure that your Makefile refers to these files**.
- *We will automatically overwrite your submission with the original content of the files provided to you,* with the exception of the files indicated in the *Deliverables* section. If the code does not compile or work now, then you won't receive any credit for your work.
- You need to submit a compressed archive file named <STUDENT_ID>.tgz which contains a single directory containing all the required files. i.e.,

```
$ mkdir 123456789
Copy all the necessary files into 123456789
$ tar czf 123456789.tgz 123456789
```