CSE 504 Programming Assignment #2

E-- Syntax Analyzer

1 Overview

E-- is a rule-based language that is designed for event monitoring. It is not meant to be a full programming language, but does permit basic expressions, arithmetic and functions.

E-- events are identified by a name and a set of parameters. E-- rules are of the form $pat \longrightarrow action$, where *pat* is a pattern over events. A pattern may use regular expression operators. The *action* component will be executed when the *pat* component matches an observed sequence of events.

E-- compiler is supposed to generate code for event monitoring. This code will be linked together with a runtime library that will deliver events that are to be monitored. This generated code is supposed to invoke the actions associated with a pattern, whenever the incoming event stream matches the pattern.

The semantics of E-- rules will be described in class in the near future. For the purposes of this assignment, you need to be concerned with only the syntax of E--, which is described in detail below.

2 Notational Conventions

Anything inside [and] is optional. We will enclose the [and] within single quotes where we want to escape this special interpretation. Other characters such as ":" , ";" , "{", "}", "(", "&" and ") " have no meaning, and must appear where they are placed in the syntax structure. By definition a list must contain at least one item unless otherwise stated. *Italicized* symbols are those that are tokens which will come directly from the lexical analysis phase (i.e. they are terminal symbols of the grammer).

3 Operators

For the purpose of defining a grammar for E--, the following symbols are considered as operators, and associated with the following information regarding their precedence and associativity. Each row in the following list specifies operators with the same priority and associativity information. Operators at a given precedence level have a higher priority over all other operators with lower precedence levels.

Operators	Precedence Level	Associativity
=	1	non-associative
\/	2	left
:	3	left
**	4	left
pattern not	5	right
||	6	left
&&	7	left
!	8	right
==, !=, >, <, >=, <=	9	non-associative
|, ^	10	left
&	11	left
~	12	right
<<, >>	13	left
+, −	14	left
*, /, %	15	left
unary minus	16	right
[	17	left
.	18	left

The precedence information is used in the following manner. If you have an expression $E_1 op_1 E_2 op_2 E_3$, then it is equivalent to

- $(E_1 op_1 E_2) op_2 E_3$, either if op_1 has higher priority over op_2, or if both op_1 and op_2 have the same priority and are left-associative
- $E_1 op_1 (E_2 op_2 E_3)$ either if op_2 has higher priority over op_1, or if both op_1 and op_2 have the same priority and are right-associative

Same reasoning holds even when you consider unary operators. The intuitive idea is that a symbol with higher precedence "binds more tightly" than one with lower precedence. Thus $!E_1 \&\& E_2$ is equivalent to $(!E_1) \&\& E_2$. Similarly, $a + b[i]$ is equivalent to $a + (b[i])$, $a.b[i]$ is equivalent to $(a.b)[i]$, and $a.b(c).d(e).f[i]$ is equivalent to $((((a.b)(c)).d)(e).f)[i]$.

4 E-- Syntax

1. A **Specification** consists of a (possibly empty) list of bf Declarations and a non-empty list of **Rules**.
2. Statements, declarations, and rules are terminated by a semicolon. Semicolons are optional following a closing brace.
3. A **Declaration** declares a **Class**, **Function**, **Event** or a **Variable**.
4. A class declaration consists of the keyword *class* and a name for the class. Classes represent an abstract, external datatype in E--. Class objects can be parameters to external functions or events, but not variables.

5. A function declaration consists of a **Type**, function name, and zero or more comma-separated **FormalParam**'s enclosed by a parenthesis. Optionally, a function may have a body, which consists of a (possibly empty) sequence of variable declarations and one or more statements, all of them enclosed by braces.

6. A **FormalParam** consists of a **Type** followed by a variable name.

7. A **Variable** declaration consists of a **Type**, followed by a comma-separated list of variable names. Each variable name may have an optional initialization, which consists of `TOK_ASSIGN` and an **expr**.

8. An **Event** declaration consists of the keword *event* followed by an event name and comma-separated list of **FormalParam**'s enclosed by a parenthesis. A special event *any* matches any event, and need not be followed by parenthesis.

9. A **Rule** consists of an **EventPattern** and a **Statement** separated by *TOK_ARROW*.

10. An **EventPattern** can be a **PrimitivePat** or be obtained from other event patterns using one of the operators `!` (negation), `:` (concatenation), `\/` (alternation) or `**` (closure). These operators have the usual precedences and associativities of regular expression operators. Parentheses may be used to override these precedences and associativities.

11. A **PrimitivePat** consists of an event name, followed by event formal parameters (variable names, i.e., identifiers) enclosed within parentheses and separated by commas. A primitive pattern may be optionally followed a condition, which consists of a `TOK_BITOR` followed by an **expr**.

12. A **Statement** is one of the following

    - **IfStatment**, of the form *if* **expr Statement**, followed optionally by *else* **Statement**
    - **EmptyStmt**, which is empty.
    - **FunctionInvocation**, which consists of a function name followed by a comma-separated, parenthesis-enclosed, possibly empty list of **Expr**
    - **Assignment** of the form **RefExpr** = **Expr**
    - **ReturnStmt** of the form *return* **Expr**
    - **CompoundStatement** that consists of a sequence of one or more **Statement**'s that is enclosed in braces.

13. The **RefExpr** is simply a variable naame

14. An **Expr** is one of:

- **Literal**
- **RefExpr**
- **Assignment**
- **FunctionInvocation**
- *op* **Expr**, for unary operator *op*
- **Expr** *op* **Expr**, for binary operator *op*

The precedence and associativity of different operators were specified earlier. Parentheses could be used to override these precedences and associativity.

15. A **Type** is a base type (one of *void, bool, string byte, int,* or *double*), or a type name. A base type could be preceded by the keyword *unsigned*.

5 Description

In this assignment, you will construct a parser for the above grammar using `Bison`, a parser-generator tool. The parser should include print statements that indicate the reductions performed. Typically, the print statement will simply be the name of the left-hand-side of the production. In some cases, however, the right-hand side will be printed. Refer to the `print.txt` file for a complete list of all the print statements in the parser used to produce the standard output files. You need to refer to this since you need to match your outputs exactly to those in the standard output files.

In addition, your parser should have good error-handling. Good error-handling means that (a) understandable error messages are produced, and (b) the parser recovers from almost all syntax errors and can continue to parse the rest of the program. The exact list of error messages found in the standard output files is shown in `error.txt`.

6 Integrating with Lexical Analyzer

The parser needs to be integrated with the lexical analyzer that you had built in programming assignment #1. Following is a brief description of how the integration will be done.

All terminal symbols in the grammar must be declared using the `%token` directive of the parser-generator `bison`. When you do this, `bison` will specify the token numbers for each token in a "interface header" file. This generated file will take the role of `E--.tab.h` that was given to you for the last assignment. In addition to the token numbers, the interface must specify the types for attributes (remember YYSTYPE in `E--.tab.h`?). The types for attributes is specified using the `%union` directive of `bison`. You will specify the attribute type as:

```
%union {
   char* cVal;
   unsigned int   uVal;
   double dblVal;
};
```

You may get several conflicts from Bison that involve **error** productions. You can try to reduce these, but in general, you will not be able to eliminate them. For the purposes of grading, we will ignore conflicts involving error productions. Other than these, you should get only one other conflict, a shift-reduce conflict arising due to the dangling-else problem. If you get any more, then there must be something wrong with the way you have translated the grammar provided to you. Go back and check it - – most probably, you missed some precedence or associativity information. If this does not work, use `-v` option of `bison` to get a "verbose" file `E--.tab.C.output`. This file contains the productions, states and transitions used by the parser. This file will also tell you where the conflicts are, i.e., which states and which productions. (Tip: there are several conflict messages at the top of the file. You should ignore them, as they do not provide much useful information. The conflicting transitions are enclosed in brackets, so you can get to them by searching for for open brackets (i.e., '[')). From this information, you should be able to make out what the offending grammar rules are and how they should be modified. If, after several attempts, you are unable to resolve the problem, contact the Instructor or the TA.
IMPORTANT: *To avoid running into difficulties with conflicts in the grammar*, you should add the grammar rules gradually - – as few at a time as possible. Then run `bison` to ensure that you do not run into any conflicts. If you do run into conflicts this way, then the problem is most likely to be in the rules that you just added.

7 Available Material

The following material is available:

- The driver source file `driveParse.C`
- The lexical analyzer file `E--_lexer.C`. If you got your lexical analyzer to work completely in the last assignment, then you can use your lexical analyzer rather than the one that is provided in this directory. (But there are few changes to be made: tokens TOK_QMARK, TOK_LBRACK, TOK_RBRACK, TOK_ENUM, TOK_BIT, TOK_INTERFACE, and TOK_USE are not used any more. New token TOK_PAT_STAR ("**") is introduced. The token TOK_SYNTAX_ERROR is changed to TOK_LEX_ERROR.)
- A sample Makefile that automates the building of your executable `demo`. Modify it as you deem fit. Assignment requires one of GNU C++ compiler (g++) or LLVM's Clang C++ compiler (clang++). According to what

is available on your system, you would need to update CC=.. line from Makefile. By default, Makefile uses g++ compiler. Other than this, I do not expect that you will need to make changes to this file, but do make sure that you understand it, just in case you need to make some changes.

- Sample input files `in*` and the respective output files `out*` in the `test` subdirectory. Sample syntactically incorrect `in*` input files and the respective errors `err*` and output files `out*` in the `errtest` subdirectory.
- A sample `E--_parser.y++` file, with all of the token declarations and a very small subset of the grammar rules. As given, it can be used to build `demo`, which can then process the test file `in01`.

8 Deliverables

The parser handin must contain the parser specification `E--_parser.y++`, the lexical analyzer `E--_lexer.C` and `Makefile`. The Makefile must create an executable called `demo`. Please read carefully the submission requirements below.

9 Submission Requirements

Please note the following submission requirements. These are not simply conventions, but are **requirements**. If you have trouble with the set-up, post your problem to discussion group.

- 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**.
- The submission process remains as in assignment 1.