CS 445 – Spring 2023

Homework 5

Due on or before 2 Apr at 11:59 PM Pacific Time.

1. The problem

In this assignment we will make all error and warning messages have the same format and add modifications to keep syntax errors from halting syntactic analysis in the compiler. Lots of code is provided for you to use, so you just need to fold in the code and adapt it to your grammar and token names. There are several parts to this assignment:

1. Modify your program to improve error messages by using `yyerror` and error count.
2. Modify your grammar to use the bison `error` token and the `yyerrok` macro.

1.1 Improve Error Messages

Using yyerror permits us to catch all errors that are returned from the parser. Here are some examples of error messages that might be passed into the yyerror function if YYERROR_VERBOSE is set:

```
syntax error, unexpected '+'
syntax error, unexpected ELSE
syntax error, unexpected '<=', expecting '('
syntax error, unexpected '<=', expecting WHILE
syntax error, unexpected ID, expecting '('
syntax error, unexpected ID, expecting WHILE
syntax error, unexpected '+', expecting ',' or ';'
syntax error, unexpected '/', expecting BOOL or INT or VOID
syntax error, unexpected ID, expecting $end or BOOL or INT or VOID
```

For this part of the assignment we will add a line number, format the error message like it appears in the syntactic analysis phase, and some extra information to the message where appropriate. The yyerror.cpp and yyerror.h files for this assignment on the course website can be used to translate these messages as specified. You can use this code as-is, or improve it if you want as long as your output is the same as the provided examples. This code contains a tiny sort routine that sorts the expected tokens so that a uniform error message can be printed regardless of the order of your token declarations in your parser.y file. Note that this code uses names for tokens that may be different from the names that your code is using, so you may need to adjust some token names to get the code to work.

If you aren't enabling the YYERROR_VERBOSE macro in your yacc specification already, you need to do so in order to get the yyerror.cpp code to work. You can just ust do a `#define YYERROR_VERBOSE` in your parser.y file to get it to work. In the yerror.cpp code, the `line` symbol is the line number where the parser is located when an error is encountered, and the `msg` symbol is the extended error message that comes from the parser when your YYERROR_VERBOSE macro is enabled.

Your compiler will continue to have two global variables to count the number of errors and warnings that were encountered. In this assignment we will modify things so that the count of errors and warnings also includes warnings and errors from the scanner and the syntax analyzer.

Your compiler now has three phases: lexical analysis (scanner), syntax analysis, and semantic analysis. Each of these phases can produce its own errors. Below is a list of the list of the modifications that are necessary for this assignment (some of these you may have already completed) for each phase:

- Lexical Analysis: All warnings and errors increment the appropriate counter.

- Syntax Analysis: You will be adding tokens to your bison grammar to allow the parser to match an error, automatically revert to a "known good state" and continue parsing. You will also add the yyerrok macro to synchronize as needed. A list of possible edits to your grammar is shown in the grammarmods.txt document on the course website.

- Semantic Analysis: Your compiler should proceed to the semantic analysis phase only if there are no errors encountered in the syntax analysis phase. The errors encountered in this phase will increase the global error count.

Your compiler should continue to keep a running count of all warnings and errors from all phases and report them at the end of compilation in the order and format shown below:

```
Number of warnings: 2
Number of errors: 496
```

1.2 Modifying Your Grammar

For this assignment you will need to modify your grammar to add the error token so that semantic analysis can continue past errors. This is done to help the user get as much information as possible about their program in a single compilation. A list of possible edits to your grammar is shown in the grammarmods.txt file on the course website. The edits are shown in the order in which they appear in my grammar, which is very similar to the standard C- grammar. In some cases, only the beginning of the actions associated with a given production are shown. Sometimes the only thing that is added in this list is the addition of the macro `yyerrok`, and this means that the `yyerrok` macro should be used somewhere in your actions for the corresponding production. Your token names will likely be different than the ones that are used, but these names will be translated into "nice" names so that your implementation names are hidden. Some of the productions in your grammar may be slightly different and that is fine. On the final evaluation of this assignment you will be graded on the actual error messages that your compiler generates, not where you place your `error` tokens. Try to come as close as possible to the output that is provided in the examples. The text of the errors should be exactly the same. You may miss 10% of the messages or have some extra messages and your score will not be reduced because of it. Just make sure that the text of the error messages matches exactly.

Once you have added the error token to your grammar bison will report many shift/reduce and reduce/reduce conflicts. This is expected. Your compiler should not halt due to a syntax error.

2. Deliverables and Submission

Your homework will be submitted as an *uncompressed* .tar file that contains no subdirectories. Your .tar file must contain all files and code necessary to build and test your compiler. If you use ourgetopt, *be sure to include the ourgetopt files in your .tar archive***.**

The .tar file is submitted to the class submission page. You can submit as many times as you like. **The last file you submit before the deadline will be the only one graded.** No late submissions are accepted for this assignment. For all submissions you will receive email at your uidaho address showing how your file performed on the pre-grade tests. The grading program will use more extensive tests, so thoroughly test your program with inputs of your own. The error messages that your c- emits will be sorted before comparison with the expected output so that the order in which your messages are printed is not as important as it otherwise would be.

Your program must run with no runtime errors such as segmentation violations (SIGSEGVs).