CS445 Introduction to Compilers

Assignment 4 (Quick and Dirty Syntax Error Repair) DUE: Mon Nov 18 at 5 PM PDT

180 points

Do not put off this assignment. Assigment 5, the code generation assignment, is due by finals! You need time to work on that.

There are several parts to this assignment.

- nice yyerror and error count. I will supply most of the code. The hard paart is to integrate it with your running parser.
- insert the error token into your grammar. I will give you a list of Bison statements. You copy them into the right place in your grammar.

Nicer Errors

The function yyerror catches all the message that come back from the parser. We will replace Bison's version with our own amped up version that will also make it easier to do output comparison.

Here are some examples of the raw syntax error messages that could come into yyerror from the parser if YYERROR_VERBOSE is set:

```
Invalid input character
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 BOOLEAN or INT or CHAR
syntax error, unexpected ID, expecting $end or BOOLEAN or INT
```

We will put in a line number and format the message to look like in the errors in the semantic analysis section. In this tar: yyerror tar are several useful string routines, a newer version of the symbol table which I will borrow as a string mapping function, and finally the yyerror function itself. You will need to work these into your code to produce the same syntax error messages that I do.

Yyerror Function

This function takes an error message from bison as a (char *) and prints it. It is in the file yyerror.cpp in the tar above. It comes with a header file. In it are declared four external variables. Two from the scanner called line and lastToken and two from the parser called numErrors and numWarnings.

initYyerror Function

This function is also defined in the yyerror.cpp file. It must be called before parsing begins to zero out the error and warning counts and set up a mapping between your token classes and stadardized set of nice names. **Do not change the nice name strings** You can use your own classes for the keys. More about that later.

Line Number

In **line** is the line number where the parser is working at the time of the error. It should be a global. It will be referred to in the yyerror routine to print nice symantic errors.

lastToken

lastToken is the current token being examined. Figure out how to capture that if you aren't already. It should be a global passed between the scanner code and yyerror. It has the the value of yytext of token last scanned. This is necessary for the syntax errors but insufficient for the semantic errors which need the line number stored with each node as we have discussed. This global will be used by yyerror to produce more informative errors.

Set YYERROR_VERBOSE

```
#define YYERROR_VERBOSE
```

This will get you nicer error messages to disect.

Error Count

There will be two global variables defined in the parser and counting the number of errors and number of warnings. At the end of assignment 3 we only had errors. Now we will also have warnings for practice. Warnings will not stop the successful compilation of the program. Keep count of both kinds and report at the end of the compile as in this example:

```
Number of warnings: 1279
Number of errors: 496
```

I will explain shortly where you get errors and warnings and what happens as a result.

Compiler Phases

Our compiler now was three phases: lexical analysis, syntax analysis, and semantic analysis. Each phase can produce its own errors. Here is the description of what you need to do:

- **Lexical analysis:** In this phase each input character that is not part of a legal token is **reported as a warning**. yylex **does not return a token** for these "bad characters" and the lexical analysis continues to run. Example warnings for C-:

```
WARNING(1): Invalid input character: @.  Character ignored.
WARNING(107): Invalid input character: ^C.  Character ignored.
```

These errors will increase the global warning count.

Getting this to work is relatively easy. In the scanner, create a rule after all legal token patterns that matches any remaining character. **Do NOT** do a return after the error. This way no token is passed back.

- **Syntax analysis:** The syntax analysis errors will continue to work as before (except as noted below) but **these errors will also increase the global error count.** We will be adding error tokens to our bison grammar to allow the parser to match even error creating input and keep running. By just adding the error token productions below into your parser and integrating the yyerror function into your parser the syntax error messages should come out very pretty. This will require you to edit the initYyerror() function to set the niceNames to match your tokens for any tokens you have spelled out rather than using a single character.
- **Semantic analysis: Only if there are no errors** in the syntax analysis then you proceed to semantic analysis. The errors in semantic analysis will increase the global error count.
- **At end of compile:** Print the total number of errors and warnings from all phases. For example

  ```
  Number of warnings: 28
  Number of errors: 496
  ```

  or

  ```
  Number of warnings: 0
  Number of errors: 0
  ```

Inserting Error Tokens

We want to add error tokens so syntactic analysis continues past errors. We do this to help the user get as much useful information about their program as we can in one compile.

Add the following error productions to your grammar: A list of required error productions The productions are clustered into groups where the first line is a reference line from your Bison and the remaining lines are what you need to insert there. The only action provided is setting the return pointer to NULL and possibly the `yyerrok`. You can add whatever other actions you like in the braces.

The result of adding these tokens will be that may have many shift/reduce conflicts, and reduce/reduce conflicts. I will ignore them.

You may have to adjust your grammar a little to get these to fit. On the final grading assignment you will be graded on the actual error messages you generate, not where you put your error tokens.

NOTE: the order in which tokens are reported is related to the order in which the tokens appear in your productions. If you don't get a match because of token order then reorder your productions.

Different names for tokens also produce a conflict. Try to name them the same as I do to make it easier for both of us.

Example Input and Output

```
  1 main(int *; int x[; int y]; int x[10; int y 10]) {
  2        int 'x';
  3        int *;
  4        fred(+, +, 34);
  5        *=3;
  6        }
  7 
  8 main(int x[, int y], int x[10, int y 10]) {
  9        fred(+, +, 34);
 10        *=3;
 11        }
```

```
ERROR(1): Syntax error.  Unexpected '*'.  Expecting id.
ERROR(1): Syntax error.  Unexpected ';'.  Expecting ']'.
ERROR(1): Syntax error.  Unexpected ']'.  Expecting ')'.
ERROR(1): Syntax error.  Unexpected number: 10.  Expecting ']'.
ERROR(1): Syntax error.  Unexpected number: 10.  Expecting ')'.
ERROR(1): Syntax error.  Unexpected number: 10.  Expecting ')'.
ERROR(2): Syntax error.  Unexpected character constant: 'x'.  Expecting id.
ERROR(3): Syntax error.  Unexpected '*'.  Expecting id.
ERROR(4): Syntax error.  Unexpected '+'.
ERROR(4): Syntax error.  Unexpected ';'.  Expecting ')'.
ERROR(5): Syntax error.  Unexpected '='.
ERROR(8): Syntax error.  Unexpected ','.  Expecting ']'.
ERROR(8): Syntax error.  Unexpected int.  Expecting ')'.
ERROR(9): Syntax error.  Unexpected '+'.
ERROR(9): Syntax error.  Unexpected ';'.  Expecting ')'.
ERROR(10): Syntax error.  Unexpected '='.
```

Example Runs

Coming soon Here are **some example test runs**.