CS210 - Programming Languages
Homework #5 - Spring 2024

Due on or before Sunday, April 14 at 11:59:59 PM

Tools such as `lex` (or `flex`) can be used to automatically generate a scanner for a given language. The scanner generated by `lex` partitions the input into lexemes as per the specification provided to lex. This specification describes the *lexical structure* of a grammar.

For some applications, nothing more is needed than a specification of the lexical structure of a grammar. However, it is usually the case that an application needs to recognize sequences of lexemes and perform specific actions based upon these sequences. Such applications require knowledge regarding the *phrase structure* of the grammar that generates the language.

Tools such as `yacc` (or `bison`) can generate a parser that recognizes valid strings in a given language generated by an input grammar. Thus `yacc` is referred to as a *parser generator*. The name `yacc` is an acronym for Yet Another Compiler Compiler. A compiler compiler is a tool that is used to build compilers. `Yacc` generates parsers for a class of grammars known as `LALR(1)`. The input to the parser generated by `yacc` is almost always the lexemes that are produced by the scanner generated by `lex`. In fact, `lex` and `yacc` are so frequently used together that lots of people forget that they can be used independently.

`Yacc` was first built in 1972 by Stephen C. Johnson. There are many, many references for `yacc` and `bison` on the web, and if you're interested in books "Lex & Yacc" by John R. Levine, Tony Mason, and Doug Brown (O'Reilly and Associates, 1990) is widely held in high regard. `Yacc` takes as input a grammar that you specify, and generates a parser that recognizes valid "sentences" in that grammar.

The specification for `yacc` is provided in a file that by convention has `.y` as its extension. The structure of the file containing the specification for `yacc` is very similar to the structure of the file containing the specification for `lex`. This is because the file structure of `lex` was patterned after the file structure of `yacc`. The general format for a `yacc` specification is:

```
declarations
%%
rules
%%
program segments
```

The `declarations` and `program segments` sections may be empty. The `declarations` section contains declarations of the tokens used in the grammar, the types of values used by the parser, and other miscellany. The `declarations` section may also contain a literal block of C code (contained in %{ %} just like in a `lex` specification) that is copied into the top of the generated parser. The `rules` section contains the productions of the grammar for which `yacc` will generate a parser. The productions in this section are presented in a form similar to `BNF`. The `program segments` section contains C code that is copied verbatim into the generated parser toward the end of the file.

As is the case with `lex`, when you use `yacc` the output is a file containing code in the C programming language. The file that `yacc` generates is named `y.tab.c` by default. You can also use `yacc` to generate a header file containing declarations and symbol type definitions that are used by both `lex` and `yacc` by specifying the `-d` option when invoking `yacc`.

The Homework

On the course website you will find a project containing the files necessary to build a working calculator using `lex` and `yacc`. For this homework you are to modify this working calculator to add functionality as specified later in this document. The calculator project is composed of the following files:

- Makefile
  - This is the file that is used to build the project. You can build the project by just typing `make` on the command line.
- calc.l
  - This file contains the `lex` specification for the calculator project.
- calc.y
  - This file contains the `yacc` specification for the calculator project.
- sym.h
  - This file contains information describing the representation of symbols in the calculator language. Information contained in this file is used by both calc.l and calc.y.

If you download and build this project without modifying any files, you will be presented with a working (albeit limited) calculator. The following is a session using the calculator after it has been built:

```
# calc
2/3
= 0.666667
a = 9 / 5
a
= 1.8
b = a + 2
b
= 3.8
c = (a + (b * 3))
c
= 13.2
#
```

In the example above, the bold text represents input from the user and text that does not appear in bold is the output from the calculator. You should download and build the calculator and verify that it works as depicted above before you begin working on this assignment. To terminate a session when running the calculator, simply type `CTRL-D` or whatever key combination represents `EOF` on your installation.

Your objective for this assignment is to add the following functionality to the existing calculator:

1: Zero Division Check

The existing calculator does not perform a check to ensure that division by 0 is prohibited. Please modify your calculator to perform a check to ensure that division by 0 is not allowed. If you perform a division by 0 in the existing calculator, your results will be as follows:

```
# calc
4 / 0
= inf
a = 9 - (3 * 3)
a
= 0
4 / a
= inf
#
```

In order to earn full credit for this problem, you must modify your calculator so that it detects division by 0, emits a message of the form "divide by zero" when such an operation is detected, and prohibits the operation from being performed. A session running a working calculator that successfully performs this check is shown below:

```
# calc
4 / 0
divide by zero
= 4
a = 9 - (3 * 3)
a
= 0
4 / a
divide by zero
= 4
#
```

2: Unlimited Number of Symbols

The existing calculator only allows the user to define at most 3 symbols. The names for these symbols may be arbitrarily long, but there can be at most 3 defined. Please modify your calculator so that it does not impose any limit on the number of symbols that may be in use at any given time. There will of course be limitations imposed by an operating system, or the amount of RAM available to the process hosting the calculator, etc.; these are expected and you don't really have any control over them. To earn full credit for this problem, you must modify your calculator so that symbols are stored in a linked list or a hash table or some other dynamic structure that places no practical upper bound on the number of symbols that can be defined at any given time. Please note that for this homework you are expressly forbidden from using the Standard Template Library (STL) container classes (e.g. <vector>) in **any** form. Use of the STL container classes in any form on this assignment will result in the loss of at least 50% of the total possible points available for this assignment.

The following session shows the behavior of the existing calculator:

```
# calc
width = 198.22
height = 1901.43
area = width * height
depth = 24.3
Too many symbols
#
```

In the session above, the calculator has detected that the user has attempted to use more than 3 symbols, so it emits the message "Too many symbols" and immediately terminates. This behavior is undesirable for many reasons.

A session running a working calculator that permits an unlimited number of symbols is shown below:

```
# calc
width = 198.22
height = 1901.43
area = width * height
depth = 24.3
volume = area * depth * 1.35
volume
= 1.23643e+07
...
```

Note that this calculator does not emit the "Too many symbols" message and terminate.

3: Constant Symbols

All symbols in the existing calculator are readable and writable. It is often the case that predefined constant symbols are used in calculations with the expectation that these constants may not be overwritten. Please modify your calculator so that it starts up with predefined constant symbols PI and PHI whose values cannot be modified. These constant symbols are expected to be used in arbitrary expressions, but may not be overwritten.

A session running a working calculator that contains the predefined constants PI and PHI is shown below:

```
# calc
PI
= 3.14159
PHI
= 1.61803
r = 11.4
circlearea = PI * r * r
circlearea
= 408.281
height = 9.23
cylvol = circlearea * height
cylvol
= 3768.43
PI = 23
assign to const
PI
= 3.14159
#
```

Your calculator must define the constant symbols PI and PHI and ascribe to these symbols the values 3.14159 and 1.61803 as shown above. Your calculator must then permit PI and PHI to be used in any expression that doesn't attempt to alter their values. If the user attempts to change the value of PI and/or PHI, your calculator must emit the message "assign to const" and prohibit the attempted alteration as shown above. Please note that your program must *not* terminate if the user attempts to change the value of a constant symbol.

4: Symbol Inventory

In the existing calculator, a user must query the name of each symbol that is in use in order to determine the value currently associated with that symbol. This can be a tedious and error-prone process when a user wants to query the values of multiple symbols. You must modify your calculator to allow the user to see the values of all symbols that are being used at any given time. This command must list the values of both constant and mutable symbols. The command for querying the values of all symbols is an expression containing just a question mark. A session showing a working calculator that implements the symbol inventory command is shown below:

```
# ./calc
?
num-syms: 2
        PHI => 1.61803
        PI => 3.14159
a = PI / PHI * (PI / 2)
a
= 3.04988
b = a - PI
9 / 5
= 1.8
```

```
a
= 3.04988
b
= -0.0917121
?
num-syms: 4
        PHI => 1.61803
        PI => 3.14159
        a => 3.04988
        b => -0.0917121
c = b * (PHI + 1)
d = (((c / (a - PHI) * b) + 2.6 * PI * PI) * ((a + b + c) * PHI))
c
= -0.240105
d
= 112.922
?
num-syms: 6
        PHI => 1.61803
        PI => 3.14159
        a => 3.04988
        b => -0.0917121
        c => -0.240105
        d => 112.922
#
```

Your calculator must output all symbols in `ASCII` alphabetical order (aka ASCIIbetical order) as shown above, ordered by the symbol name. Your calculator must format the output of the symbol inventory command as shown above. The output of this command must contain a string of the form "num-syms: N" where N is the number of symbols that are currently in use (as shown above.) The output of this command must then list all symbols, each on a separate line, prefixed by a single `TAB` character. The separator => must appear between the name of the symbol and the value currently associated with that symbol, and a single space must appear between the name of the symbol and the => separator, and between the => separator and the current value of the symbol.

Your homework must be submitted as an uncompressed .tar archive containing no subdirectories via `cscheckin`. When you have completed your homework, use `tar` to make an archive of your project files, naming the archive "hw5.tar" (without quotes). Then submit it to your instructor.

The work you submit for this homework must be entirely your own. Your homework must build and run on the course server. If your homework does not build on the course server with the Makefile that you have provided, at least 50% of the total score for this homework will be deducted. You may not use the STL in any form for this assignment. You should have already learned how to design and implement trees and lists in previous courses, so use that knowledge to complete this assignment.