CS 343 – Structure of Programming Languages
Winter 2013

Programming Project #2
Due Date: Friday, March 15, 2013

Programming Language: C
Language Generation Tools: lex, yacc

Objectives: C programming with linked lists, basic proficiency with lex/yacc compiler generation tools, and hands on experience developing an interpreter that implements a simple (but not entirely trivial!) programming language.

In this assignment you will implement Calc+, an interpreted imperative language. You will use a combination of C programming, and the lex/yacc tools to accomplish this. The Calc+ language sports the following nifty features:

- evaluates integer arithmetic expressions
- ability to store the results of expressions in variables
- ability to print out the values of variables or expressions

Calc+ Language Description

All statements are terminated with a semicolon token. Whitespace is ignored.

Identifiers:
Valid identifiers in Calc+ consist of 1 or more alphabetical characters. Identifiers are case-sensitive. The following identifiers are reserved words and cannot be used as variable names: print, exit. Variables are not declared in advance. The interpreter can allocate memory for them as they are encountered during interpretation. All variables are of type integer, and are initialized to 0 when they are encountered. The following variable names would be valid:

```
numberOfCats
a
WOW
```

The following would not be valid variable names:

`var1` (digits not allowed)
`print` (print is a keyword)
`name_cnt` (underscore not allowed)

Expressions:
Valid arithmetic operators are +, -, *, and /. Normal precedence rules apply, and can be altered by proper placement of parenthesis in the expression. Expressions can be formed with using integer literals and identifiers. An expression all by itself (e.g. not in a print or assignment

statement (see below) followed by a semicolon is considered a valid statement, and its interpretation is to evaluate the expression and print the resulting value to standard output.

The following are valid expressions:

```
pi * r * r
(a + 100) / cnt
b – a * c
3
a
```

Note that in the second example, the addition happens before the division, and in the third example, the multiplication happens before the subtraction. If a semicolon would be added to the end of any of the above expressions, the interpreter would simply evaluate the expression and print the result.

Assignment Statements:

Assignment statements are formed as follows:

```
{identifier} = {expression} ;
```

where identifier and expression are defined above. The semantics of this statement are to compute the value of the expression on the right hand side of the = terminal, and assign it to the memory location corresponding to the identifier on the left hand side of the = terminal. Assignment statements generate no output.

Print Statements:

Print statements are formed as follows:

```
print {expression};
```

where expression is defined above. The semantics of this statement are to evaluate the expression and print it to standard output. As discussed above in the expression section, the equivalent behavior and be produced by simply entering the expression followed by a semicolon. For example:

```
{expression} ;
```

Exit Statement:
The exit statement is formed as follows:

```
exit ;
```

This statement causes the interpreter to terminate interpretation and exit.

Example Calc+ Program

The following is an example of a valid Calc+ program:

```
Total = 3 + 100 + 1020 + 10 + 4;
cnt = 5;
print Total / cnt;
avg = Total / cnt;
print avg;
print (3 + 100 + 1020 + 10 + 4) / cnt;
(3 + 100 + 1020 + 10 + 4) / cnt;
exit;
```

Implementation Guidelines and Requirements

You are to implement your interpreter in C, using both lex and yacc to handle lexical analysis and parsing/semantic analysis, respectively. Proceed by identifying all the terminal symbols and making sure you have a lex scanner that can recognize them. Next, write a yacc grammar that meets the specification described above. It might be good to focus initially on just getting the grammar right and not worrying about the actual interpretation of the statements (e.g. the actions associated with each production could simply print out debug messages.) Once you have the grammar correct you can go back in and add action code that properly interprets the input.

(Hint: The yacc grammar you came up with in Homework #10 is a good place to start!)

In addition to the parse tree itself (which is hidden in the code generated from your yacc grammar!) one of the more interesting and important data structures in a language processor (compiler or interpreter) is the symbol table. This data structure basically maps identifier names to the actual implementation details of that identifier. Lookups on this data structure need to be reasonably efficient to support efficient execution of programs that might references thousands of different variables.

You are to implement your interpreter using the guidelines provided below.

Symbol Table Implementation.

A fairly common approach to implementing a symbol table is to use a hash table. Figure 1. below sketches out the hash table data structure you are to implement.

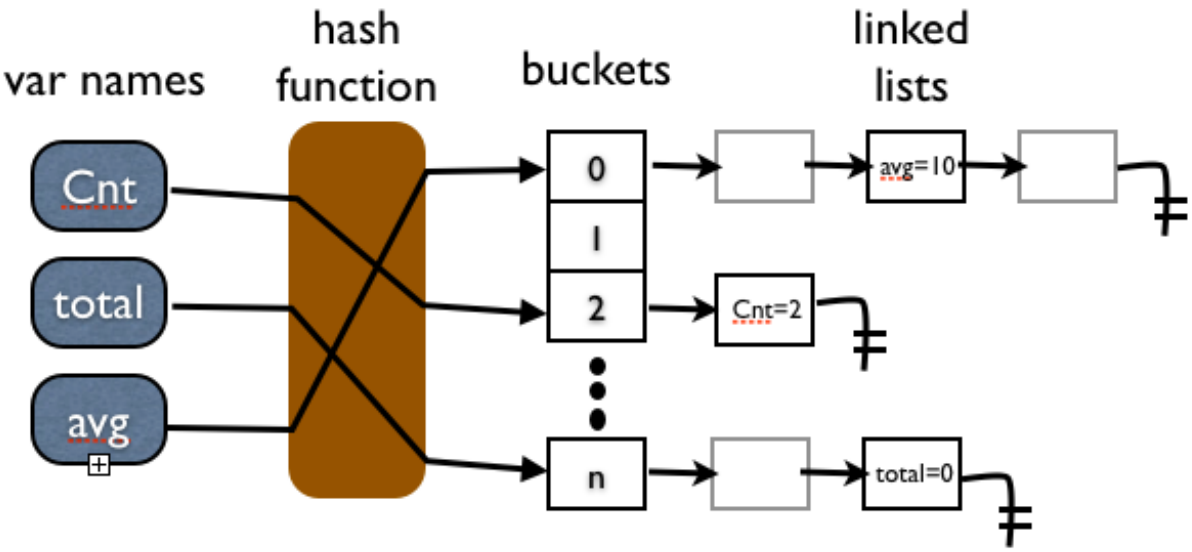

Figure 1. Hash table implementation using linked lists.

In this implementation, every identifier encountered in the input code is deterministically hashed into one of n buckets. Associated with each bucket is a simple linked list containing a node for each identifier mapped into the bucket. Each node would contain the necessary information about that identifier. In Calc+, it is sufficient to store the name and value of the identifier. The former is needed to identify which node holds the information for a given identifier, and the latter is used to persist the value of the identifier between references. Implement your symbol table using Figure 1 as your model.

Implementation Requirement: In your symbol table data structure, you must implement a linked list from scratch, using pointers and structs in C. The exact details of the list (LIFO, FIFO, etc.) are up to you.

The hash function must be deterministic (e.g. same input must map to the same bucket every time). For performance reasons, you also want it to be fairly uniform in terms of distribution of identifier values to buckets. If too many symbols get mapped to the same bucket, performance of your interpreter can suffer. You are free to innovate in this area. One suggestion would be to add up the ASCII values of the characters in the token, and then mod it by the number of buckets + 1. Feel free to try several different hash functions and perhaps evaluate your hash performance by doing a million or so lookups on a stream of identifier references randomly generated on a set of thousands of identifiers.

Understanding the bigger picture.
So how does the symbol table above fit into the bigger picture? It's quite simple. Whenever the action of a production in your grammar references an identifier and you need the current value, you will simply look up the identifier's value as it is currently stored in the symbol table. If you do not find an entry in the table for that identifier (e.g. it's the first time it was references) you will need to create a node for it and insert it into your symbol table and assume its value is 0.

Whenever the action of a production in your grammar needs to assign a value to an identifier (e.g. when executing an assignment statement), you need to look it up in the symbol table and update the stored value for that identifier (or insert it with that value if this is the first time the identifier is referenced in the current program.

NOTE: Make sure you test carefully with the provided test data files, and that you submit the correct source files when you are finished. Students will NOT be given a second chance to resubmit. I will be grading what is submitted to Blackboard, and that is final. I very much regret having to be inflexible on this, but the last project cost me many hours of extra grading time due to entirely avoidable testing and/or submission mistakes made by students.

Deliverables:

1. **Submit a zip archive of your project including any C source, lex, and yacc files to Blackboard.**
    - **Make sure your name is present in the comments section.**
2. **Because of possible portability issues, make sure your program compiles on EOS machines. This is where I will compile, execute, and test your program for grading.**