Assignment #8: Minimal BASIC

Due: Friday, December 1 at 5:00 p.m.

In this assignment, your mission is to build a minimal BASIC interpreter, starting with the code for the integer expression evaluator presented in Chapter 14. This assignment is designed to accomplish the following objectives:

- To give you practice working with trees and function pointers.
- To provide you with a better sense of how programming languages work on the inside. Learning how programming languages work helps you develop valuable intuition about the programming process.
- To offer you the chance to adapt an existing program into one that solves a different but related task. The majority of programming that people do in the industry consists of modifying existing systems rather than creating them from scratch.

What is BASIC?

The programming language BASIC—the name is an acronym for Beginner's All-purpose Symbolic Instruction Code—was developed in the mid-1960s at Dartmouth College by John Kemeny and Thomas Kurtz. It was one of the first languages designed to be easy to use and learn and remains popular as a medium for teaching programming at the elementary and secondary school level.

In BASIC, a program consists of a sequence of numbered statements, as illustrated by the simple program below:

```
10 REM Program to add two numbers
20 INPUT n1
30 INPUT n2
40 LET total = n1 + n2
50 PRINT total
60 END
```

The line numbers at the beginning of the line establish the sequence of operations in a program. In the absence of any control statements to the contrary, the statements in a program are executed in ascending numerical order starting at the lowest number. Here, for example, program execution begins at line 10, which is simply a comment—the keyword **REM** is short for **REMARK**—indicating that the purpose of the program is to add two numbers. Lines 20 and 30 request two values from the user, which are stored in the variables **n1** and **n2**, respectively. The **LET** statement in line 40 is an example of an assignment in BASIC and sets the variable **total** to be the sum of **n1** and **n2**. Line 50 displays the value of **total** on the console, and line 60 indicates the end of execution. A sample run of the program therefore looks like this:

```
? 2
      ? 3
      5
```

Line numbers are also used to provide a simple editing mechanism. Statements need not be entered in order, because the line numbers indicate their relative position. Moreover, as long as the user has left gaps in the number sequence, new statements can be added in between other statements. For example, to change the program that adds two numbers into one that adds three numbers, you would need to make the following changes:

1. Add a new line to read in the third value by typing in the command

```
35 INPUT n3
```

   This statement then goes logically between line 30 and line 40.

2. Type in a new assignment statement, as follows:

```
40 LET total = n1 + n2 + n3
```

   This statement replaces the old line 40, which is no longer part of the program.

You can delete lines from a program by typing in the line number with nothing after it on the line.

Expressions in BASIC

The **LET** statement illustrated by line 40 of the addition program has the general form

LET *variable* **=** *expression*

and has the effect of assigning the result of the expression to the variable. In Minimal BASIC, expressions are pretty much what they are for the sample interpreter in Chapter 13 that forms the starting point for this assignment. The only difference is that the assignment operator is no longer part of the expression structure. Thus, the simplest expressions are variables and integer constants. These may be combined into larger expressions by enclosing an expression in parentheses or by joining two expressions with the operators **+**, **-**, *****, and **/**, just as in the interpreter presented in the reader.

Control statements in BASIC

The statements in the addition program illustrate how to use BASIC for simple, sequential programs. If you want to express loops or conditional execution in a BASIC program, you have to use the **GOTO** and **IF** statements. The statement

GOTO *n*

where *n* is a line number transfers control unconditionally to line *n* in the program. If line *n* does not exist, your BASIC interpreter should generate an error message informing the user of that fact.

The statement

IF *conditional-expression* **THEN** *n*

performs a conditional transfer of control. On encountering such a statement, the BASIC interpreter begins by evaluating the conditional expression. In Minimal BASIC, conditional expressions are simply pairs of arithmetic expressions separated by the operators **<**, **>**, or **=**. If the result of the comparison is true, control passes to line *n*, just as in the **GOTO** statement; if not, the program continues with the next line in sequence.

For example, the following BASIC program simulates a countdown from 10 to 0:

```
10 REM Program to simulate a countdown
20 LET T = 10
30 IF T < 0 THEN 70
40 PRINT T
50 LET T = T - 1
60 GOTO 30
70 END
```

Even though **GOTO** and **IF** are sufficient to express any loop structure, they represent a much lower level control facility than that available in C and tend to make BASIC programs harder to read.

List of statements in Minimal BASIC

The section lists the set of statements recognized by the minimal BASIC interpreter.

REM This statement (which is short for **REMARK**) is used for comments. Any text on the line after the keyword **REM** is ignored.

LET This statement is BASIC's assignment statement. The **LET** keyword is followed by a variable name, an equal sign, and an expression. As in C, the effect of this statement is to assign the value of the expression to the variable. In BASIC, assignment is not an operator and may not be nested inside other expressions.

PRINT In the minimal version of the BASIC interpreter, the **PRINT** statement has the form:

> **PRINT** *exp*

where *exp* is an expression. The effect of this statement is to print the value of the expression on the console and then return to the next line.

INPUT In the minimal version of the BASIC interpreter, the **INPUT** statement has the form:

> **INPUT** *var*

where *var* is a variable read in from the user. The effect of this statement is to print a prompt consisting of the string **" ? "** and then to read in a value to be stored in the variable.

GOTO This statement has the syntax

> **GOTO** *n*

and forces an unconditional change in the control flow of the program. When the program hits this statement, the program continues from line *n* instead of continuing with the next statement.

IF This statement provides conditional control. The syntax for the **IF** statement is:

> **IF** *conditional-expression* **THEN** *n*

where the conditional expression is formed by combining arithmetic expressions with the operators **=**, **<**, or **>**. If the condition holds, the next statement comes from

the line number following **THEN**. If not, the program continues with the next line in sequence.

END Marks the end of the program. Execution halts when this line is reached.

The **LET**, **PRINT**, and **INPUT** statements can be executed directly by typing them without a line number, in which case they are evaluated immediately. Thus, if you type in (as Microsoft cofounder Paul Allen did on the first demonstration of BASIC for the Altair)

```
PRINT 2 + 2
```

your program should immediately respond with 4. The statements **GOTO**, **IF**, **REM**, and **END** are only legal if they appear as part of a program, which means that they must be given a line number.

Basic commands

In addition to the statements shown above, BASIC accepts several commands that control its operation. These commands cannot be part of a program and must therefore be entered without a line number.

RUN This command starts program execution beginning at the lowest-numbered line. Unless the flow is changed by **GOTO** and **IF** commands, statements are executed in line-number order. Execution ends when the program hits the **END** statement or continues past the last statement in the program.

LIST This command lists the steps in the program in numerical sequence.

HELP This command provides a simple help message describing your interpreter.

QUIT Typing **QUIT** exits from the BASIC interpreter. The easiest way to implement this function is to call the ANSI library function **exit(0)**, although CodeWarrior still requires you to activate the **Quit** command in the **File** menu to dismiss the console window.

Summary of statements and commands

The following statements can only appear in a program and thus must always have line numbers:

```
REM
GOTO
IF
END
```

The following can be used as statements in a program (with line numbers) or as standalone commands (without line numbers):

```
LET
PRINT
INPUT
```

The following commands cannot be part of a program and thus never have line numbers:

```
RUN
LIST
HELP
QUIT
```

Example of use

Figure 1 below shows a complete session with the BASIC interpreter. The program is intended to display the terms in the Fibonacci series less than or equal to 1000; the first version of the program is missing the **PRINT** statement, which must then be inserted in its proper position.

Figure 1. Sample run of the basic interpreter

```
 Minimal BASIC -- Type HELP for help.

      100 REM Program to print the Fibonacci sequence
      110 LET max = 1000
      120 LET n1 = 0
      130 LET n2 = 1
      140 IF n1 > max THEN 190
      150 LET n3 = n1 + n2
      160 LET n1 = n2
      170 LET n2 = n3
      180 GOTO 140
      190 END

      RUN
      145 PRINT n1

      LIST
      100 REM Program to print the Fibonacci sequence
      110 LET max = 1000
      120 LET n1 = 0
      130 LET n2 = 1
      140 IF n1 > max THEN 190
      145 PRINT n1
      150 LET n3 = n1 + n2
      160 LET n1 = n2
      170 LET n2 = n3
      180 GOTO 140
      190 END

      RUN
      0
      1
      1
      2
      3
      5
      8
      13
      21
      34
      55
      89
      144
      233
      377
      610
      987
```

Getting started

The **`Assignment 6`** folder on the CS Catacombs and the web-site contains a slightly modified version of the code for the expression evaluator described in Chapter 14. The complete project consists of the following modules:

```
eval.c
exp.c
interp.c
parser.c
scanadt.c
symtab.c
```

Although this expression evaluator is not BASIC, it is a good starting point. It can read, parse, and evaluate expressions that are similar to those in BASIC. The main differences are as follows. **Most students who have difficulty with this assignment have failed to understand one or more of these points**:

The starter evaluator does not deal with programs. There are no line numbers, no **`IF`**'s, and no looping. Each line is executed only once.

The starter evaluator has only one kind of input, an expression to be evaluated. A BASIC interpreter accepts commands for immediate evaluation (like **`LIST`**) and numbered lines that are added to the current program but not executed immediately.

In the starter evaluator, the **`=`** operator is simply another operator, dealt with by the parser and evaluator just like any other. In particular, embedded assignments are not ruled out. In BASIC, however, you should treat **`=`** as part of the **`LET`** command. Embedded assignments are not allowed, so the parser just deals with arithmetic operators.

The **`RUN`** command causes a BASIC interpreter to leave its read/evaluate interaction with the console and enter and execution mode where it executes the lines of the program in order of line number or as directed by **`IF`** and **`GOTO`** statements.

There is an important difference between the version of the expression evaluator supplied with this module and the one given in the text is that the module **`interp.c`** contains additional code to recover from errors. If you are entering a program and make a syntax error, you do not want your entire evaluator to bomb out with an error. Even so, it is extremely convenient in the code to call **`Error`** to produce the error messages. Thus, the best thing to do in the implemenation is to add code that allows the interpreter to detect and recover from calls to **`Error`**, using an approach called **exception handling.**

The details of exception handling are not the focus of this assignment. The code that you need is already included in the implementation of the main program, which looks like this:

```
main()
{
    scannerADT scanner;
    expressionADT exp;
    string line;
    int value;

    InitVariableTable();
    scanner = NewScanner();
    SetScannerSpaceOption(scanner, IgnoreSpaces);
    while (TRUE) {
        try {
            printf("=> ");
            line = GetLine();
            if (StringEqual(line, "quit")) exit(0);
            SetScannerString(scanner, line);
            exp = ParseExp(scanner);
            value = EvalExp(exp);
            printf("%d\n", value);
          except(ErrorException)
            printf("Error: %s\n",
                   (string) GetExceptionValue());
        } endtry
    }
}
```

The new feature in this code is the **try** statement, which is defined in the **exception.h** interface. The **try** statement operates by executing the statements in its body, which consists of all statements up to the **except** clause. If no errors occur, the program exits from the **try** body and goes back to the **while** loop to read another command. If **Error** is called at any point in the execution of the **try** body—no matter how deeply nested that call might be—control passes immediately to the **except** clause, which displays the error message. Because the error exception has been handled by the **try** statement, however, the program does not simply exit as is usually the case when calling **Error**. The program instead continues through the end of the **try** statement, after which it goes back to read the next command.

Your BASIC interpreter will be almost unusable without this feature. You should keep of this **try ... endtry** block in your code and modify the details within it to suit your needs. The details of how it is implemented are beyond the scope of CS106B, but it is a "black box" that will make your life easier on this assignment.

Requirements

Your task in this assignment is to throw away most of the code in **`interp.c`** and replace it with code that implements the BASIC interpreter described in this handout. As you write your program, you should make sure that your code obeys the following requirements:

1. *It should maintain the modular structure of the expression evaluator program.*. If you are adding extensions for the BASIC contest (see Handout #40), you may want to subdivide the code a bit further.
2. *The statements and the commands should be implemented using the command table approach described in Section 11.7 of the text.* In other words, when your program encounters a statement like **`LET`** or a command like **`RUN`**, it should invoke the corresponding code by looking up the command name in a symbol table and then calling a function pointer embedded in the value of that symbol. Note that the command functions you store in the table will probably need to have a different prototype from the command functions in the chapter.
3. *You should keep changes to the supplied eval, exp, and parser modules to a minimum.* The only change you need to make is to **remove the `=` operator from the parser**, because BASIC does not allow embedded assignments.

Helpful hints

1. Although it might seem like the right data structure for the statements in a program would be a doubly linked list to facilitate insertion and deletions, you can actually get away with **using an array** for the statements, indexed by the line number, which run from 1 to 999. Finding the next statement following the current one requires linear searching when you use this strategy, but the program as a whole is significantly easier to write.
2. When you read a stored statement into a program, you have some choices as to how to proceed. For efficiency, it would be nice to store some parsed representation of the command as well, in much the same way that the Darwin **`species`** module does. For the purposes of this assignment, however, it is sufficient to store the text of each line as a string and then reparse it every time that statement is executed. You need to store the text of the line in any case so that you can make the **`LIST`** command work.
3. Testing your Minimal BASIC interpreter on programs can be a big pain, because runtime errors that force you to restart your interpreter also wipe out any statements you have entered. To save yourself the tedium of typing all the statements again, the easiest thing to do is to create a file containing the BASIC program you're using as a test case—such as the countdown or Fibonacci programs given in this handout—and then use copy and paste to enter the statements into your interpreter. If you're feeling more ambitious, you might consider implementing a file loading and saving system, which would be a good testing method and worth some little amount of extra credit.
4. Start early, and ask questions if you get stuck.