Assignment #6: Minimal BASIC

This assignment designed by Eric Roberts.

Due: Mon, May 22nd in class

We're going to try something a little different in the schedule for the remaining two assignments. This assignment is officially due next Monday, but you may hand it in up to the beginning of Wednesday's lecture without it being counted late. Your final assignment will go out on that Monday and will be due the following Wednesday, the last day of class. Both assignments are reasonably challenging and although either can be done in a week, I'm leaving it up to you to decide which one to give a little extra time to. For some of you, you might find it easier to get #6 out of the way, leaving yourself with more time for #7. For others, you might find it more convenient to apportion the time the opposite way. Although it's always tempting to just go for the later deadline, realize that leaving it until Wednesday will cut into the time you have to do #7 (and thus potentially encroaches on preparing for finals). This is a little funky, but I hope giving you this latitude allows you to schedule as it works best for you!

To be clear on all the various scenarios: #6 is due on Monday, but turning it in on Wednesday will still be counted as "on-time." Turning in #6 on Friday May 23rd is one class day late, the following Wednesday (since class doesn't meet on Monday for Memorial Day) is two days late, that same Wednesday is on-time for #7 submissions. The absolute last day to submit either program is Friday, June 2, #6 would be 3 days late and #7 would be 1 day late.

The assignment

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 both easy to use and learn and remains popular as a medium for teaching programming in elementary and secondary schools.

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.

Assignments 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. You may wonder why the keyword **LET** is used in BASIC. If you look at the program above (and the description of the language below), you will see that all statements in BASIC begin with a keyword. That simplifies the process of writing an interpreter. For each statement, the first token is the keyword that determines what action to take.

One other thing to notice about BASIC programs is that variables do not have to be declared before they can be used. The first appearance of a variable (in a **LET** or **INPUT**) serves as its declaration and initialization. In our Minimal BASIC, we will only support variables of integer type.

Arithmetic expression in BASIC

In Minimal BASIC, expressions are built out of integers, variable names, parentheses, and the four operators **+**, **-**, *****, and **/**. 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 an operator. The precedence of the operators is the same as in C: multiplication and division have higher precedence than addition and subtraction. Operators of equal precedence may be evaluated in any order. Parentheses may be used in the conventional way to control the order of evaluation.

Chapter 14 of the text discusses an expression evaluator that operates somewhat similarly to what you will need for BASIC and we give you the code from the chapter as your starting point. Although much of it will work without changes, you do need to understand it thoroughly so you can properly adapt it for your purposes. One difference you should be sure you understand is that in the text, the assignment operator **=** can be part of an expression, whereas in BASIC, the **=** is a "throwaway" character in a **LET** statement, and the expression that follows it can involve the arithmetic operators but not the **=** operator. Thus you cannot have "embedded assignments" in BASIC like you can in C, and you will have to be sure that your interpreter enforces this.

For use in the **IF** statement (described below) you will need to extend the expression evaluator to allow conditional expressions. A conditional expression consists of two arithmetic expressions joined by one of the relational operators and evaluates to true or false. The precedence of the relational operators is lower than any of the arithmetic operators, so an expression such as **3 + 4 > 3 * 2** parses as you would expect into **(3 + 4) > (3 * 2).**

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. Unlike C, BASIC does not allow any arithmetic expression to be used a test, only a compound expression joined by a relational operator is valid.

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.

Summary of statements in Minimal BASIC

The section summarizes 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.

Statements in programs vs. statements for immediate execution

Normally, all the statements mentioned above appear as lines of a program, and each begins with a line number. You type in all the lines of the program, and when you run the program, execution proceeds in order of line number unless this is changed by a **GOTO** or **IF**. This is a lot like executing a C program, except that there is no "compile" step.

BASIC allows some statements to be executed in another way, without even being part of a program. This takes a little getting used to by C programmers! 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**.

If you type

```
LET x = 2
```

then the value of the variable **x** is set to **2** in spite of the fact that no program is running. This assignment will be in effect when the next program is run. For example, suppose you type the **LET** statement above (setting **x** to **2**), then type

```
10 LET y = 3
20 PRINT x + y
30 END
```

The value **5** would be printed. This ability to mix “immediate” execution of statements and execution as part of a running program may seem curious, but that is the way it works in BASIC. Not all statements can be executed in "immediate" mode. 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.

Interpreter commands

In addition to the statements shown above, BASIC accepts several commands that control the interpreter’s 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 (prints to the console) 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 on the Mac, 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
```

A day in the life of a BASIC interpreter

Hopefully, at this point, you have a grasp on what BASIC is like, what the various statements do, and how they can be used. What you may be wondering about is exactly what you are supposed to write. Your job is to write an interpreter for BASIC. An interpreter is a program that spends its life reading commands from the user and acting on them, as suggested by the following pseudocode:

```
while (TRUE) {
     read command from user
     act on command
}
```

The structure of an interpreter is pretty simple. The text discusses an interpreter that evaluates expressions, but in our case, the commands to be interpreted are from BASIC. If the user types a

PRINT command, something gets printed. If the user types a **LET** command, a value gets assigned to a variable. If the user types **QUIT**, the interpreter exits. At this "outer" level, then, your interpreter does what all interpreters do: read, act, read, act, etc. There is, however, more to the story than that.

First we should remember that one thing the user may type is a line that is to be part of a program; i.e., a statement that begins with a line number. The interpreter still reads it and acts on it, but in the case of a numbered statement, the "action" is just to store the line away for later use. You won't even analyze the line when it is typed, just store it away in an array of strings. When that line is later executed, that it when the interpreter figures out what to do with it.

The next thing to think about is that one of the commands is **RUN**. The "action" for that command is to run the program that the user has typed in as numbered statements. Execution starts with the lowest numbered line, and proceeds until an **END** statement is encountered or the execution continues past the last numbered line. What kind of code executes a BASIC program? That turns out to be an interpreter too! Its job is something like this:

```
while (!doneWithProgram) {
     find next statement
     execute statement
}
```

This is a lot like the read/act loop that we discussed earlier. We are going to call this the "inner" interpreter. To be specific: The "outer" interpreter interacts with the user, reading commands and acting on them. One of those commands is **RUN**, which calls the "inner" interpreter. The "inner" interpreter executes the lines of a stored program one at a time, until the program stops. Then control returns to the outer interpreter.

Don't think that there is anything recursive here. The two interpreters are different. To see this, consider what happens when the line to be interpreted is a numbered statement. For the outer interpreter, it means that the user has just typed the line, and the action is just to store it away. For the inner interpreter, it means that the line has been retrieved from storage, and the action is to figure out what it means and execute it. Even though the two interpreters are different, it will turn out that if you decompose your program properly, you can share most of the heavy-duty code between them.

Example of use

Figure 1 on the next page 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. Note that the user has typed blank lines in a few places just for clarity; they cause no action.

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
```

Exception handling

The **Assignment 6** starter folder contains a slightly modified version of the code for the expression evaluator described in Chapter 14. The only difference between the version of the interpreter 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 BASIC interpreter 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 implementation 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, and you should copy the **try** statement into your code.

Various hints and suggestions

- Your interpreter should be case-insensitive. **LET** and **let** should mean the same thing, as should variable names like **X** and **x**. Traditionally, BASIC is uppercase. If the user types in lowercase letters, you can either change everything to uppercase as you read it, or store the original for listing purposes, but ignore case during execution.

- Your interpreter can be limited to programs that have under 1000 lines. 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.
- If the target of **`GOTO`** or **`THEN`** is a non-existent line (i.e. no statement has been stored at that index for the program), it is an error. You do not search for the next following statement.
- A valid identifier (i.e. a variable name used in **`INPUT`** or **`LET`**) must start with an alphabetic letter and then can be followed by any combination of letters and digits. Thus **`A5`** is a valid identifier, **`A#`** is not, neither is **`5`**.
- 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. This means that when a malformed program statement is typed in, it is not parsed, just stored as a string, and only when that line is later evaluated will any errors be found in parsing it.
- The **`scannerADT`** as given in our text does not scan negative numbers as one token. It will separately scan the **`-`** into one token and then the digits that follow it into a separate token. The parser assumes **`-`** is a binary operator and doesn't recognize the unary form. Together this means, you cannot use **`LET`** to assign a variable to a negative integer constant. You do not have to fix this, it is fine if it this reports a parsing error. (To get around this limitation, you can assign the variable to the expression **`0-number`**)
- Starting from a working program that accomplishes solves a different problem (in this case, the simple expression evaluator) is both a blessing and a curse. The expression ADT and parsing and evaluation functions will be a great help, but you may find yourself swimming in code at first and unsure of how to proceed. The biggest part of your work will be tearing apart the interp.c (basically throwing it out) and re-structuring into modules for the BASIC interpreter. You might find it easiest to start by developing the foundation for the outer interpreter read/act loop and implementing the simple interpreter-specific commands (**`QUIT`**, **`HELP`**, etc.). Next incorporate the commands that can operate in immediate mode and spend some time verifying it with the **`LET`** and **`PRI=NT`** commands until you are sure the basic expression parsing and evaluation is correct. Now go on to add stored program handling and the inner interpreter that runs the stored program.

Requirements

As you write your program, you should make sure that your code obeys the following requirements:

1. *It should maintain good modular structure.* Note that we are not breaking up the project into modules for you, it is part of your job to devise appropriate divisions for the task at hand. The starting expression evaluator sets a good example for what we expect. If nothing else, a module that stores, retrieves, and deletes program steps based on line numbers would be a useful subdivision.
2. *The statements and the commands should be implemented using the command table approach described in Section 11.7.* 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 modules to a minimum.* Although you will want to study the provided code for the parser, expression, and evaluation modules to be sure you understand how to use them, you should not need to make many significant changes to the modules.
4. *You should unify as much as possible between the inner and outer evaluators*. For example, both must parse and evaluate expressions, make variable assignments, etc. With a well-thought-out design, there is no reason why they can't share all the common code.
5. *A small amount of static global state is permissible.* For some of your modules, you may find it convenient to store some state in global variables private to the module. It is be no means required that you use global variables, and they certainly can be avoided, but limited, reasonable use of globals will not be frowned upon for this assignment.
6. *You are not required to free memory.* Where possible, you should be reasonable about your use of memory and can avoid creating and throwing things, for example, by creating and re-using one scanner for all your parsing rather than creating multiple of them. But in truth, scanning and tokenizing is going to create a lot of dynamically-allocated strings that will litter your program heap. We are not asking you to free these, since we think it will get in the way of the main focus of the assignment. You may want to bump the heap size of your project if you find yourself running short of memory during testing.

Testing

Testing is an extremely important part of writing any interpreter/compiler. You want to be sure that your program accepts all valid input and rejects any invalid requests. The input space is large and so this will require thinking carefully about what cases need to be caught and making sure you call the Error function to report each. To get you thinking along the right lines, here are some cases to consider for the **`LET`** and **`IF`** commands. You are highly encouraged to come up with lists like this for all of the commands to ensure you are covering all the bases.

`LET`
- Should be okay if used either in executing program or in immediate mode
- Should be okay to re-assign value of a previously assigned identifier
- Error if expression references a non-existent variable
- Error if missing = after the identifier
- Error if invalid identifier (not alphanumeric)
- Error to assign to non-L-value (LET 3 + 4 = 5)
- Error if no expression follows the =
- Error from malformed expression (mismatched parens, missing terms, etc.)
- Shouldn't allow assignments to be nested (LET a = b = 4)
- Error if junk after the end of the expression

`IF`
- Error if used outside context of executing program (i.e. in immediate mode)
- Error if conditional expression references a non-existent variable
- Error if test is arithmetic, not conditional, expression (IF 3 + 4 THEN)
- Error if missing THEN keyword
- Error if target line missing after THEN
- Error if token after THEN isn't a number
- Error if line number doesn't exist in program
- Error if junk after the end of the line number

Testing your Minimal BASIC interpreter on even small programs can be a 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.

Accessing Files

On the class web site, there are two folders of starter files: one for Mac CodeWarrior and one for Visual C++. Each folder contains these files:

exp.h/.c	Expression ADT as given in Chapter 14.
parser.h/.c	Expression parsing functions as given in Chapter 14.
eval.h/.c	Expression evaluation function as given in Chapter 14.
scanadt.h/.c	Scanner ADT from Chapter 8.
symtab.h/.c	SymbolTable ADT from Chapter 11.
interp.c	Main program module for the expression evaluator.
MiniBasic Demo	Compiled version of a working program.

Deliverables

At the beginning of Monday's lecture, turn a manila envelope containing a printout of your code for all three modules and a floppy disk containing your entire project. Everything should be clearly marked with your name, CS106B and your section leader's name! Once again, remember to keep a backup in a safe place. Never turn in the only electronic copy of a program!

BASIC: (n.) a computer one-word oxymoron — Tonkin's first computer dictionary