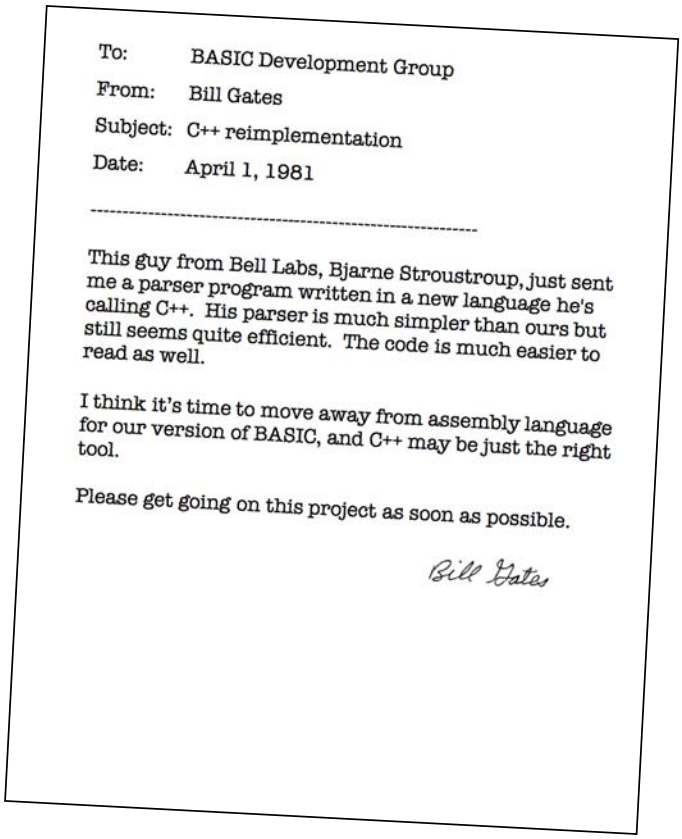
Eric Roberts
CS106B

Handout #40
May 11, 2009

Assignment #5—Minimal Basic

Due date: Friday, May 22



To: BASIC Development Group
From: Bill Gates
Subject: C++ reimplementation
Date: April 1, 1981

This guy from Bell Labs, Bjarne Stroustroup, just sent me a parser program written in a new language he's calling C++. His parser is much simpler than ours but still seems quite efficient. The code is much easier to read as well.

I think it's time to move away from assembly language for our version of BASIC, and C++ may be just the right tool.

Please get going on this project as soon as possible.

Bill Gates

In 1975, Bill Gates and Paul Allen started the company that would become Microsoft by writing a BASIC interpreter for the first microcomputer, the Altair 8800 developed by the MITS corporation of Albuquerque, New Mexico. By making it possible for users to write programs for a microcomputer without having to code in machine language, the Altair and its implementation of BASIC helped to start the personal computer revolution.

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 increase your familiarity with expression trees and class inheritance.
- To give you a better sense of how programming languages work. Learning how an interpreter operates—particularly one that you build yourself—provides useful insights into 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. Although BASIC has now pretty much disappeared as a teaching language, its ideas live on in Microsoft's Visual Basic system, which remains in widespread use.

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 is inserted into the program 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 with the updated version.

In classical implementations of BASIC, the standard mechanism for deleting lines was to typing in a line number with nothing after it on the line. Note that this operation actually deleted the line and did not simply replace it with a blank line that would appear in program listings.

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*

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 *condition* **THEN** *n*

performs a conditional transfer of control. On encountering such a statement, the BASIC interpreter begins by evaluating *condition,* which in the minimal version of BASIC consists of two arithmetic expressions joined by one of 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. The replacement of these low-level forms with higher level constructs like **if**/**else**, **while**, and **for** represented a significant advance in software technology in which programs represented much more closely the programmer's mental model of the control structure.

Summary of statements available in the minimal BASIC interpreter

The minimal BASIC interpreter implements only six statement forms, which appear in Figure 1 on the next page. The **LET**, **PRINT**, and **INPUT** statements can be executed directly by typing them without a line number, in which case they are evaluated

Figure 1. Statements implemented in the minimal version of BASIC

REM This statement 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, replacing any previous value. 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 this statement is:

IF exp_1 *op* exp_2 **THEN** *n*

where exp_1 and exp_2 are expressions and *op* is one of the conditional 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. This statement is usually optional in BASIC programs because execution also stops if the program continues past the last numbered line.

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 legal only if they appear as part of a program, which means that they must be given a line number.

Figure 2. Commands to control the BASIC interpreter

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.
CLEAR	This command deletes the program so the user can start entering a new one.
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 system function **exit(0)**.

Commands recognized by the BASIC interpreter

In addition to the statements shown above, BASIC accepts the commands shown in Figure 2. These commands cannot be part of a program and must therefore be entered without a line number.

Example of use

Figure 3 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 10000; the first version of the program is missing the **PRINT** statement, which must then be inserted in its proper position.

A tour of the starter folder

The starter folder on the CS106B assignments page contains the following files:

basic.cpp This file is a stub implementation of the main program for the BASIC interpreter. This version doesn't include any code for storing program statements or executing statements. All it does in the current version is read in expressions, evaluate them, and display the result in exactly the way that the **interp.cpp** program does in Chapter 14. You need to replace the stub implementation with one that properly updates the program and executes the commands.

exp.h
exp.cpp These files are the interface and implementation for the expression module described in Chapter 14; the versions of the files in the starter folder are copies of the ones in the book. The only changes you will need to make to these files involves a simple extension to the **EvalState** class, which you will need to keep track of the current line number in the program. That change is described in the section "Keeping track of the program sequence" on page 14.

parser.h
parser.cpp These files implement the precedence-based expression parser and are supplied to you just as they appear in Chapter 14. The only changes you need to make here are (1) to remove the code from the parser that recognizes the **=** operator because BASIC does not allow embedded assignments, and (2) to add a function **ParseStatement** that reads tokens from the scanner to create a **statementT** in much the same way that the **ParseExp** function reads tokens to create an **expressionT**. This second change is described in the section "Parsing statement forms" on page 13.

Figure 3. Sample run of the basic interpreter

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

100 REM Program to print the Fibonacci sequence
110 LET max = 10000
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 = 10000
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
1597
2584
4181
6765

QUIT
```

program.h
programpriv.h
program.cpp

These files define the **Program** class, which is responsible for storing the statements in a BASIC program in a way that allows the interpreter to run programs quickly. The starter file contains a complete version of the **program.h** header file, which defines each of the operations that the **Program** class must implement. The **programpriv.h** and **program.cpp** files, however, are simple stubs that allow the package to compile correctly but don't actually do anything. Finishing the **Program** class is one of the two big pieces of this assignment. The process for doing so is described in the section entitled "Storing the program" that begins later on this page.

statement.h
statement.cpp

These files define the **statementT** type, which is analogous to the **expressionT** type exported by **exp.h**. The **expressionT** type is defined as a pointer to the abstract **ExpNode** class, which is then extended by defining subclasses for the various types of expression node. In much the same way, the **statementT** type is defined as a pointer to an abstract **StmtNode** class, which is then extended to create subclasses for each of the statement types. Defining the **StmtNode** class hierarchy is the other big piece of this assignment, and is described in the section "The statement class hierarchy" on page 11.

error.h
error.cpp

These files reimplement the **Error** function so that errors that occur in the parsing and evaluation phases don't terminate the operation of the interpreter but instead allow it to fail gracefully back to the top level. You probably don't need to look at these files at all; the only place where the extended **Error** facility is used is part of the code you're given in the **basic.cpp** module. There are, however, some notes explaining this facility in the section "Exception handling" on page 14.

When you download the starter project, you already have a program that runs. To complete the assignment, you need to proceed strategically toward your goal by making carefully staged edits. To whatever extent you can, you should make sure that the BASIC project continues to run at the completion of each stage in the implementation.

Storing the program

The first task you need to undertake is making it so your BASIC interpreter can store programs. Whenever you type in a line that begins with a line number, such as

```
100 REM Program to print the Fibonacci sequence
```

your interpreter has to store that line in its internal data structure so that it becomes part of the current program. As you type the rest of the lines from the program in Figure 3, the data structure inside your implementation must add the new lines and keep track of the sequence. In particular, when you correct the program by typing

```
145 PRINT n1
```

your data structure must know that this line goes between lines 140 and 150 in the existing program.

The job of keeping track of lines in the program is the job of the **Program** class, which is exported by the **program.h** interface shown in Figure 4. In some ways, the **Program** class is like a vector; you can insert new elements in the middle and delete elements by

Figure 4. The `program.h` interface

```
/*
 * File: program.h
 * ---------------
 * This interface exports a Program class for storing a BASIC
 * program.
 */

#ifndef _program_h
#define _program_h

#include "genlib.h"
#include "statement.h"

/*
 * This class stores the lines in a BASIC program.  Each line
 * in the program is stored in order according to its line number.
 * Moreover, each line in the program is associated with two
 * components:
 *
 * 1. The source line, which is the complete line (including the
 *    line number) that was entered by the user.
 *
 * 2. The parsed representation of that statement, which is a
 *    statementT.
 */

class Program {

public:

/*
 * Constructor: Program
 * Usage: Program program;
 * -----------------------
 * Constructs an empty BASIC program.
 */

   Program();

/*
 * Destructor: ~Program
 * Usage: usually implicit
 * -----------------------
 * Frees any heap storage associated with the program.
 */

   ~Program();

/*
 * Method: clear
 * Usage: program.clear();
 * -----------------------
 * Removes all lines from the program.
 */

   void clear();
```

```
/*
 * Method: addSourceLine
 * Usage: program.addSourceLine(lineNumber, line);
 * -----------------------------------------------
 * Adds a source line to the program with the specified line number.
 * If that line already exists, the text of the line replaces
 * the text of any existing line and the parsed representation
 * (if any) is deleted.  If the line is new, it is added to the
 * program in the correct sequence.
 */

   void addSourceLine(int lineNumber, string line);

/*
 * Method: removeSourceLine
 * Usage: program.removeSourceLine(lineNumber);
 * --------------------------------------------
 * Removes the line with the specified number from the program,
 * freeing the memory associated with any parsed representation.
 * If no such line exists, this method simply returns without
 * performing any action.
 */

   void removeSourceLine(int lineNumber);

/*
 * Method: getSourceLine
 * Usage: string line = program.getSourceLine(lineNumber);
 * -------------------------------------------------------
 * Returns the text for the program line with the specified
 * line number.  If no such program line exists, this method
 * returns the empty string.
 */

   string getSourceLine(int lineNumber);

/*
 * Method: setParsedStatement
 * Usage: program.setParsedStatement(lineNumber, stmt);
 * ----------------------------------------------------
 * Adds the parsed representation of the statement to the
 * statement at the specified line number.  If no such
 * line number exists, this method raises an error.  If
 * a previous parsed representation exists, the memory
 * for that statementT is reclaimed.
 */

   void setParsedStatement(int lineNumber, statementT stmt);

/*
 * Method: getParsedStatement
 * Usage: statementT stmt = program.getParsedStatement(lineNumber);
 * ----------------------------------------------------------------
 * Retrieves the parsed representation of the statement at the
 * specified line number.  If no value has been set, this method
 * returns NULL.
 */

   statementT getParsedStatement(int lineNumber);
```

```
/*
 * Method: getFirstLineNumber
 * Usage: int lineNumber = program.getFirstLineNumber();
 * ------------------------------------------------------
 * Returns the line number of the first line in the program.
 * If the program has no lines, this method returns -1.
 */

   int getFirstLineNumber();

/*
 * Method: getNextLineNumber
 * Usage: int nextLine = program.getNextLineNumber(lineNumber);
 * ------------------------------------------------------------
 * Returns the line number of the first line in the program whose
 * number is larger than the specified one, which must already exist
 * in the program.  If no more lines remain, this method returns -1.
 */

   int getNextLineNumber(int lineNumber);

private:

#include "programpriv.h"

};

#endif
```

specifying their line number. There are, however, a number of differences, including the following:

- The line numbers in a program need not be consecutive, which means that there can be holes in the sequence. The **Program** class makes it possible to step through the statements in a program by exporting the pair of methods **getFirstLineNumber** and **getNextLineNumber**. The first returns the line number of the first line in the program, and the second returns the next line number given an existing one. These methods suggest that the internal representation might have something that looked like a linked list as well.
- The elements of the **Program** data structure are not simply strings. The data structure needs to keep track of the strings to implement the **LIST** command, but the interpreter also needs to store an executable representation of the statement so that it can run the program. The methods **getSourceLine**, **addSourceLine**, and **removeSourceLine** update the string part of the data structure; the methods **setParsedStatement** and **getParsedStatement** make it possible to store the executable representation as well.

You should read through the interface carefully and make sure you understand what each of the methods is supposed to be doing for you as a client. If you're confused, come to office hours or the question sessions we'll be running for this assignment and get those questions answered. You can write the implementation only if you know what you are supposed to produce.

What makes this part of the assignment challenging is that we are not going to tell you what internal data structures to use. You can use any of the structures we've developed so far or that you worked with in Chapter 4. The only think to keep in mind is that your

representation must make it possible to execute programs efficiently. The Altair 8800, after all, was very slow. It mattered a lot in those days how quickly the program could run, and it didn't make much sense to waste a lot of cycles finding out that the next statement after line 100 in a particular program was line 999. Your data structure should make that easy.

By contrast, it is not quite as important to ensure that typing in the statements is quite so efficient. Human typing is even slower than the Altair, so if it took a second to add a new line, users probably wouldn't notice. It takes much longer than that to type the line in the first place. Thus, it may be appropriate to spend more time in **`addSourceLine`** and **`removeSourceLine`** if by doing so you can make **`getNextLineNumber`** fast.

As you will come to understand when you implement the **`RUN`** command, the methods from the **`Program`** class that get called by the interpreter during the execution phase are **`getFirstLineNumber`**, **`getNextLineNumber`**, and **`getParsedStatement`**. Your design must make each of these methods run in constant time. You should try to make the other methods operate as efficiently as possible, but you shouldn't worry if some of the other methods end up taking linear time.

Once you've figured out how you want to store the data for the **`Program`** class, your next task is to complete the **`programpriv.h`** and **`program.cpp`** files with the data structures and code necessary to implement your design.

Testing the program structure

Before you continue on to work on the other phases of the assignment, it is essential to implement enough of the **`basic.cpp`** module to test your code for the **`Program`** class. The initial version of that module includes a function called **`ProcessLine`** that evaluates arithmetic expressions exactly as the code in Chapter 14 does. That's not useful for testing whether you can store program statements. You need to replace it with a version that can handle the following cases:

- If you type in a line beginning with a number, the code needs to store the entire line (including the number) in the **`Program`** object by calling **`addSourceLine`**.
- If you type in a line that contains *only* a line number and no statement body, you need to delete that line by calling **`removeSourceLine`**.
- If you type in the command **`LIST`**, the interpreter should list the source text for every line in the program in order.
- If you type in the command **`CLEAR`**, the interpreter should discard all the lines in the program, freeing any associated heap memory that was allocated along the way.

The statement class hierarchy

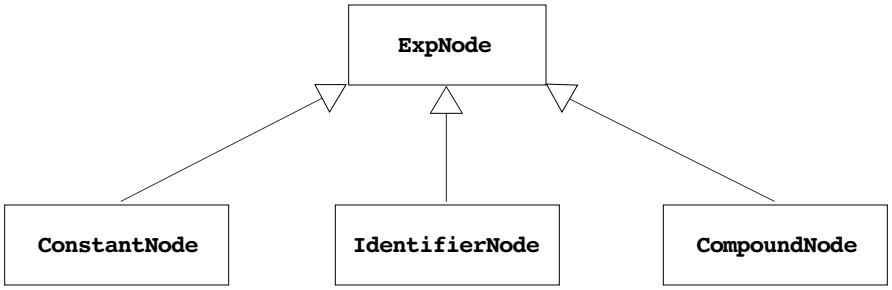
At this point, you can enter and delete lines from the program and execute the **`LIST`** and **`CLEAR`** commands. But you're not really closer to being able to run that

```
PRINT 2 + 2
```

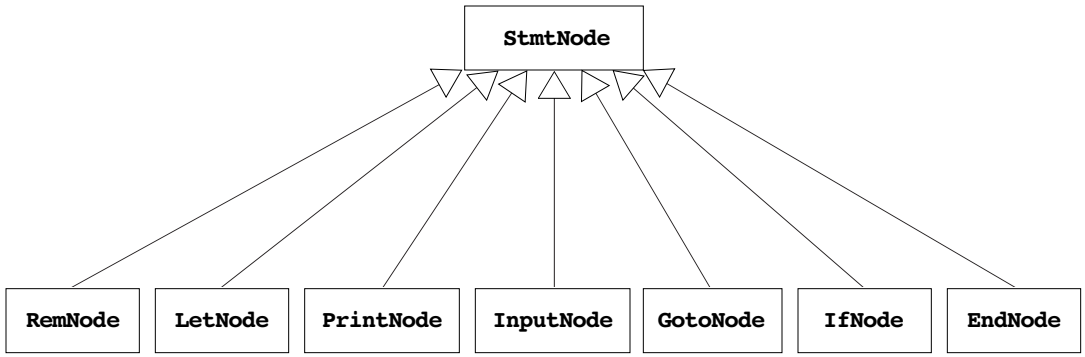
program that launched the Microsoft corporation back in 1975. To do that, you need to be able to parse and execute statements, in much the same way that the interpreter program from Chapter 14 parses and executes expressions.

Chapter 14 covers the design of the **`exp.h`** interface in considerable detail. From the client's perspective, the primary type exported by the **`exp.h`** interface is **`expressionT`**, which is a pointer to an **`ExpNode`**. The **`ExpNode`** class is the abstract superclass for a

hierarchy that includes three concrete subclasses for the three different expression types, as follows:



The structure of the **statement.h** interface is quite similar. The primary class exported by the interface is **statementT**, which is a pointer to an **StmtNode**. The **StmtNode** class is the abstract superclass for a set of subclasses corresponding to each of the statement types, as illustrated in the following diagram:



Even though there are more subclasses in the **StmtNode** hierarchy, it is still somewhat easier to implement than the **ExpNode** hierarchy for two reasons. First, you have the expression code as a model and can copy most of it, making only straightforward changes. Second, one of the things that makes the **ExpNode** hierarchy complex—but also powerful—is that it is recursive. Compound expressions contain other expressions, which makes it possible to create expression trees of arbitrary complexity. Although the same is not true for modern languages like C++, statements in BASIC are not recursive.

The starter project contains the code for the **StmtNode** class itself. What you need to do is add definitions to **statement.h** and **statement.cpp** to implement the concrete classes in the hierarchy. For each one, you need to implement the following methods:

1. A constructor that creates an instance of the subclass by parsing input from a scanner. For example, the constructor for the **LetNode** class reads a identifier name, an equal sign, and an expression from the scanner and then stores that information in instance variables. Similarly, the **GotoNode** constructor reads a line number and stores that in a different collection of instance variables, since each of the subclasses defines its own structure.
2. An **execute** method, which allows the interpreter to simulate the operation of that statement when the program runs. For the **LetNode** class, the **execute** method has to evaluate the expression that appeared on the right side of the equal sign and then store that value in the variable that appears on the left side.
3. If the instance variables for a subclass include pointers that are allocated in the heap—as all **expressionT** values are, for example—that subclass must also include a new definition for the destructor method to free that memory.

Parsing statement forms

As noted earlier in the tour of the starter project, one of your tasks is to add a method called **ParseStatement** to the **parser.h** file. The strategy for parsing a statement in BASIC begins by reading the first token on the line. If that token is the name of one of the seven legal statement forms, all you have to do is call the constructor for the appropriate **StmtNode** subclass. For example, if the first token you read from the scanner is **PRINT**, you can create the appropriate **PrintNode** statement form by calling

```
new PrintNode(scanner)
```

which will then go through and read the remaining tokens on the line and assemble them up into a **statementT**.

As a reward for reading this far in the handout, I'm going to give you the code for the **PrintNode** class, even though it is not in the starter files, so you can use it as model for the other statement forms. The **PRINT** statement in BASIC is relatively simple:

PRINT *exp*

The job of the parser is to make sure that the rest of the line after the **PRINT** keyword consists of a valid expression, and then to create a **PrintNode** object that stores that the parsed representation of the expression in an instance variable. The definition of the **PrintNode** class that you need to add to **statement.h** therefore looks like this:

```
class PrintNode: public StmtNode {
  public:
    PrintNode(Scanner & scanner);
    virtual ~PrintNode();
    virtual void execute(EvalState & state);
  private:
    expressionT exp;
};
```

The corresponding implementation in **statement.cpp** looks like this:

```
PrintNode::PrintNode(Scanner & scanner) {
    exp = ReadE(scanner);
    if (scanner.hasMoreTokens()) {
        Error("Extraneous token " + scanner.nextToken());
    }
}

PrintNode::~PrintNode() {
    delete exp;
}

void PrintNode::execute(EvalState & state) {
    cout << exp->eval(state) << endl;
};
```

The code for the constructor calls **ReadE** to read the expression and then checks to make sure that there are no more tokens on the line, since any such extraneous tokens represent a syntax error. The destructor definition is required in this class because the private data contains an **expressionT**, which is allocated on the heap. The code for the destructor, however, is extremely simple; all it does is free the memory for the expression. The **execute** method is equally simple; all the code does is evaluate the expression in the context of the **EvalState** object and send the result to **cout**.

Getting error messages right in an interpreter is more of an art than a science. As you write the code to parse each of the statement forms, try to think about all the things that might go wrong and what messages you would have like to see as a programmer if you made those mistakes.

Keeping track of the program sequence

By the time you reach this point in the handout, you are ready to try implementing the various **StmtNode** subclasses other than the **PrintNode** implementation you were given in the preceding section. Three of those statement forms—**LET**, **REM**, and **INPUT**—are at least as easy as **PRINT** was. The problematic ones are **GOTO**, **IF**, and **END**. These statements don't change the values of variables like **LET** or **INPUT** or display information like **PRINT**. What they do instead is change the execution state of the program itself.

To make this work, the **EvalState** object needs to store more than just a map of variable names and values. To implement the control statements, the **EvalState** object must also keep track of the current line being executed. Ordinarily, each time you execute a statement, you move on to the line that follows the current one. The **GOTO**, **IF**, and **END** statements can change that assumption, and you need to make sure that the **EvalState** class has the methods you need to make the control structures work.

The simplest approach to solving this problem is to add the methods **setCurrentLine** and **getCurrentLine** to the definition of the **EvalState** class. The **Run** command can then grab the statement for the current line and then call **setCurrentLine** with the line number of the next line (which you can get by calling **getNextLineNumber** in the **Program** class). If nothing else happens, the next statement will be that line number. If the statement is a control statement, however, the execution of that statement could designate a different line number as the next line in sequence, at which point the program would continue its operation from there.

Exception handling

The code for the main program in **basic.cpp** contains a new feature that you have not yet seen. The code to process the line from the user is embedded in the following loop:

```
while (true) {
    try {
        ProcessLine(GetLine(), program, state);
    } catch (ErrorException & ex) {
        cerr << "Error: " << ex.getMessage() << endl;
    }
}
```

The new feature in this code is the **try** statement. The **try** statement operates by executing the statements in its body. If everything proceeds normally, the **ProcessLine** function returns and the execution goes back to the **while** loop to read another command. If, however, the **Error** function 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 **catch** clause, which displays the error message. Because the error exception has been handled by the **try** statement, however, the program does not 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. Crashing the whole interpreter because your BASIC program has a syntax error would be incredibly frustrating, since you would lose everything you'd typed in up to that point. Including the **try**/**catch** statement means that your interpreter responds to errors much more gracefully.