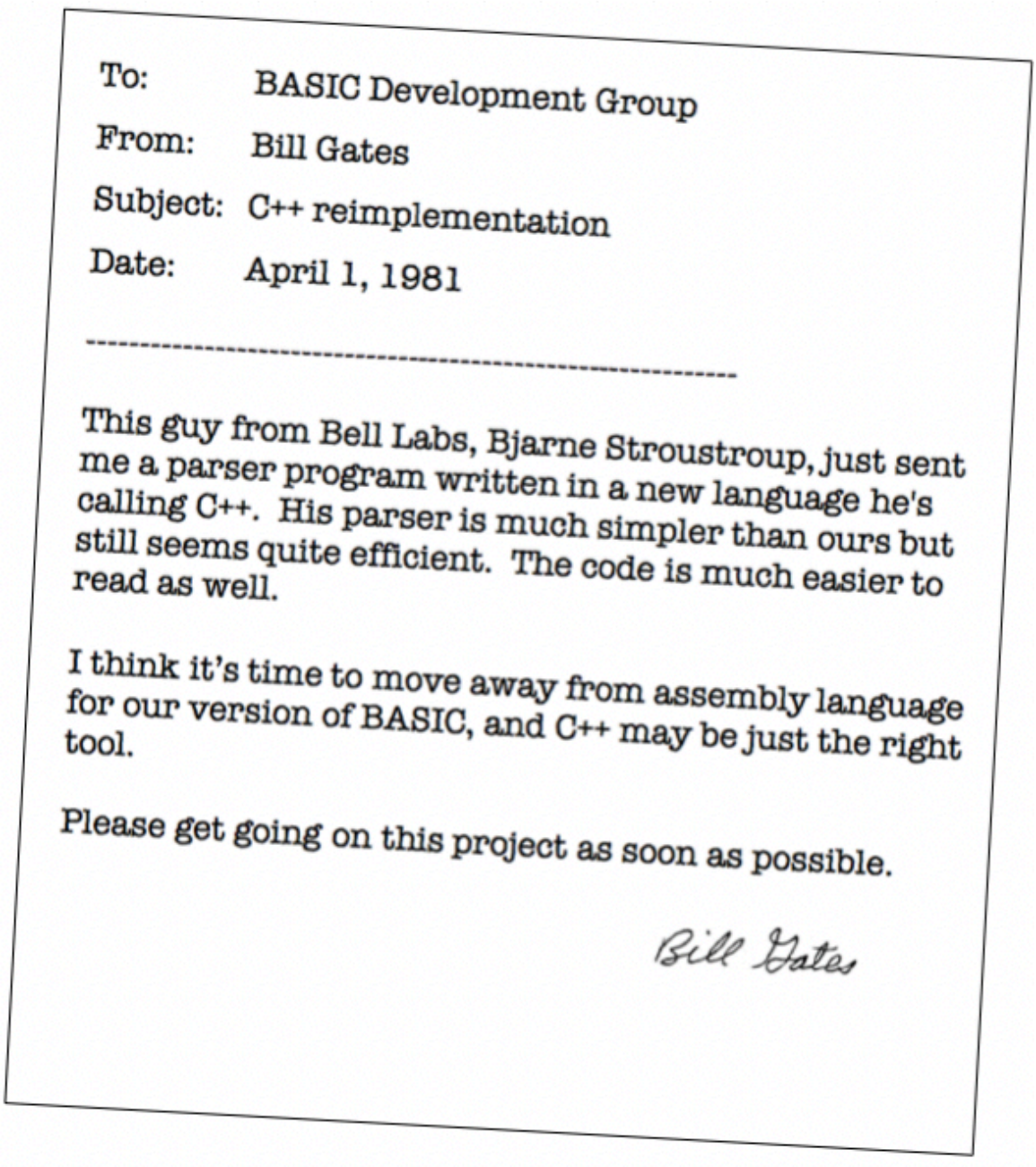
Project 1: Minimal Basic

Please compress your source code using 7z and upload the 7z file to oc.sjtu.edu.cn before the final deadline.



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 project, your mission is to build a minimal BASIC interpreter. You need 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:

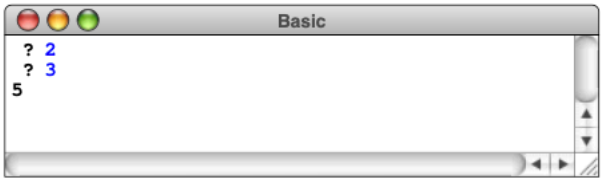


Figure 1

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
```

2. This statement is inserted into the program between line 30 and line 40. Replace the old line 40 with an update version by typing

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

In classical implementations of BASIC, **the standard mechanism for deleting lines was to type 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, **the assignment operator is no longer part of the expression structure**. 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 `/`. **You only need to support +, -, *, /, (,) operators with signed integers (at least 32-bit) in expressions. (Be aware of negative integers.)**

Additionally, you need to support the exponentiation operator in expressions:

```
exp1 ** exp2
```

The exponentiation operator returns the result of $exp1^{exp2}$, where exp1 and exp2 are expressions. The exponentiation operator is right associative, i.e., a ** b ** c is equal to a ** (b ** c). The operator has higher precedence than * and /.

For all expressions and statements, you need to handle extra spaces. For example, LET a = b + 4 * (-5 + 4).

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, allowing programs to represent 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 Table 1. 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 respond immediately 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.

Commands recognized by the BASIC interpreter

In addition to the statements listed in Table 1, BASIC accepts the commands shown in Table 2. These commands cannot be part of a program and must therefore be entered without a line number.

Table 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 minimal BASIC, 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 print a newline character so that the output from the next **PRINT** statement begins on a new 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. (The string " ? " should display in the command input edit box in GUI.)
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. Your program should report an error if line *n* does not exist.
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 program should continue from line *n* just as in the **GOTO** statement. If not, the program continues on to the next line. Note that the conditional operators (=, <, >) are not parts of expressions.
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.

Table 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.
LOAD	This command loads a file containing statements and commands. Statements and commands should be stored (also displayed in GUI) and executed respectively, as if they were entered into input box in order. A prompt window should be displayed when this command is entered. The window asks users to choose the file to load.
LIST	This command lists the steps in the program in numerical sequence. It has been required to be implemented in the previous version of this project. In the new version, your interpreter should be able to display all the codes that have been entered in real time, so there is no need to implement this command.
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.

Example of use

Figure 2 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 three output windows are used to the current program, the standard output (and errors) of program, and the syntax tree of each line of statements. User can enter statements or commands into command input box, or load a file to be executed through LOAD button. The syntax tree is displayed only when RUN is called. CLEAR will clears the content of all three windows. The

RUN and CLEAR buttons are used to execute and clear statements entered respectively.

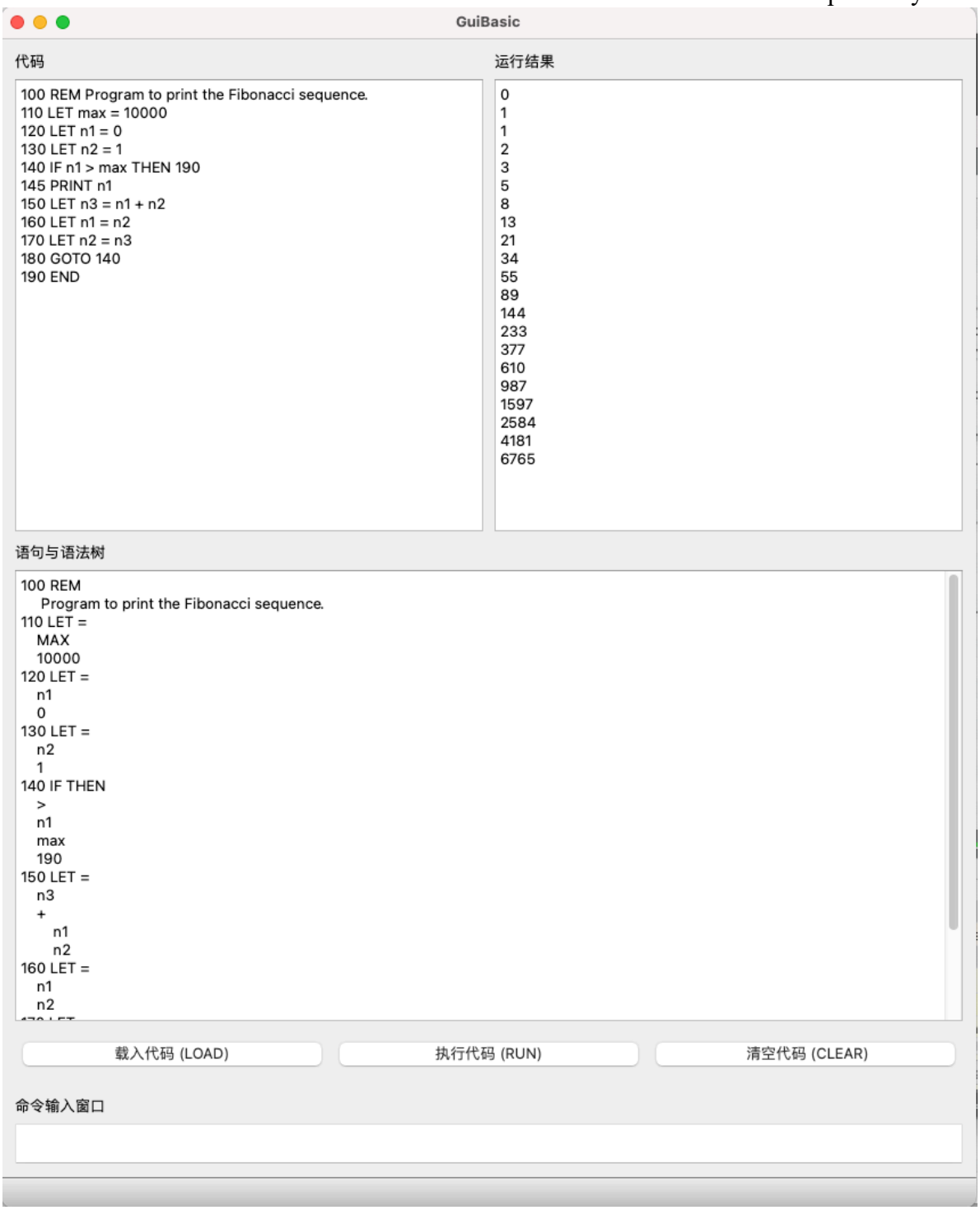


Figure 2

Syntax tree display

Conceptually, syntax tree is one abstract representation of program. More specifically, every **statement** in the program can be represented as a tree. The structure of the syntax tree can be seen as the steps of the computation of the **expression** in the statement.

As you have learnt from previous programming course, some statements have side effects, in our mini basic language, the side effects include assignment and branch. And these side effects should also be displayed in the syntax tree.

The node of the tree can be identifier definition, assignment, function call (You need not to implement function call in this project), expression computation and conditional or unconditional branch. In your interpreter implementation, you can make the computation along the syntax tree from leaf node to root. Because the connection structures of the tree are determined by the computation rules, e.g. operator priority and association.

In your implementation, you should construct the syntax tree in **infix notation**.

For displaying the syntax tree structure in your GUI window easily, you don't need to plot the real tree. Instead, you should use indentation to display the syntax tree of each statement.

Followings are some concrete examples. Structure in the red border is expression of that statement.

1. LET m = p + q*t

   The syntax tree of this statement:

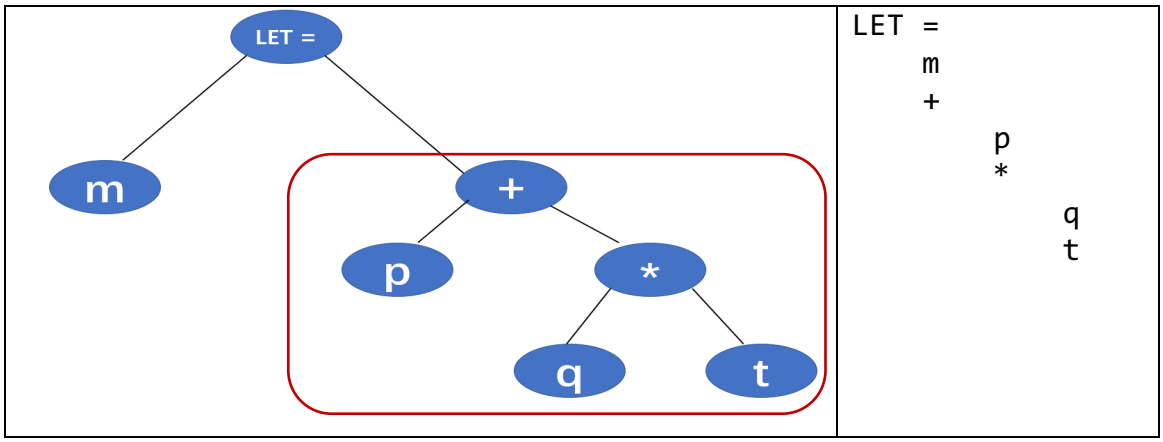


Figure 3

2. IF m > max THEN n

   The syntax tree of this statement:

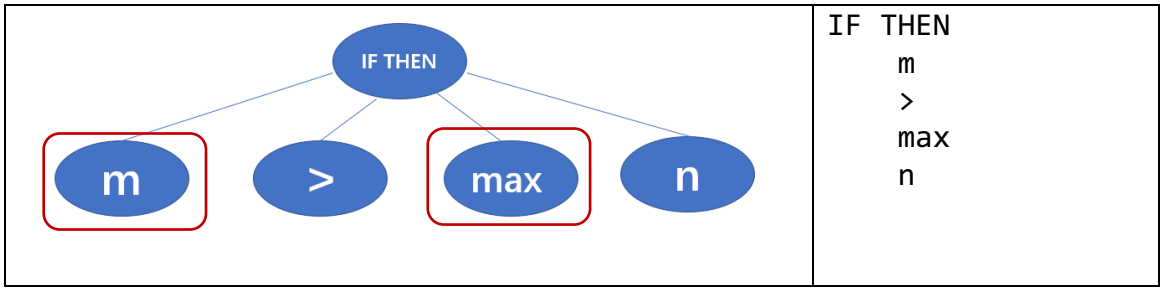


Figure 4

3. GOTO n

   The syntax tree of this statement:

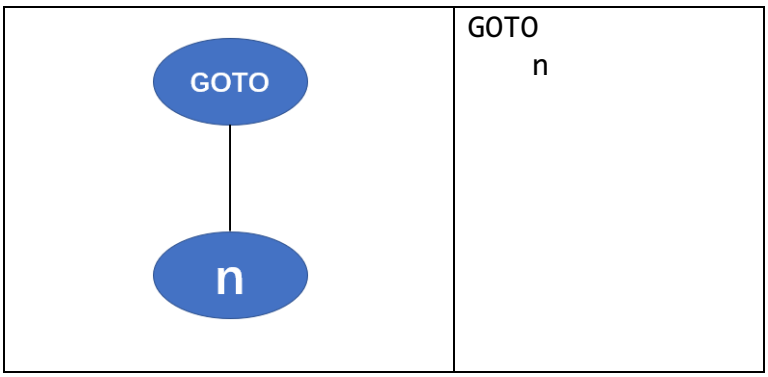


Figure 5

4. PRINT p + q*t

   The syntax tree of this statement:

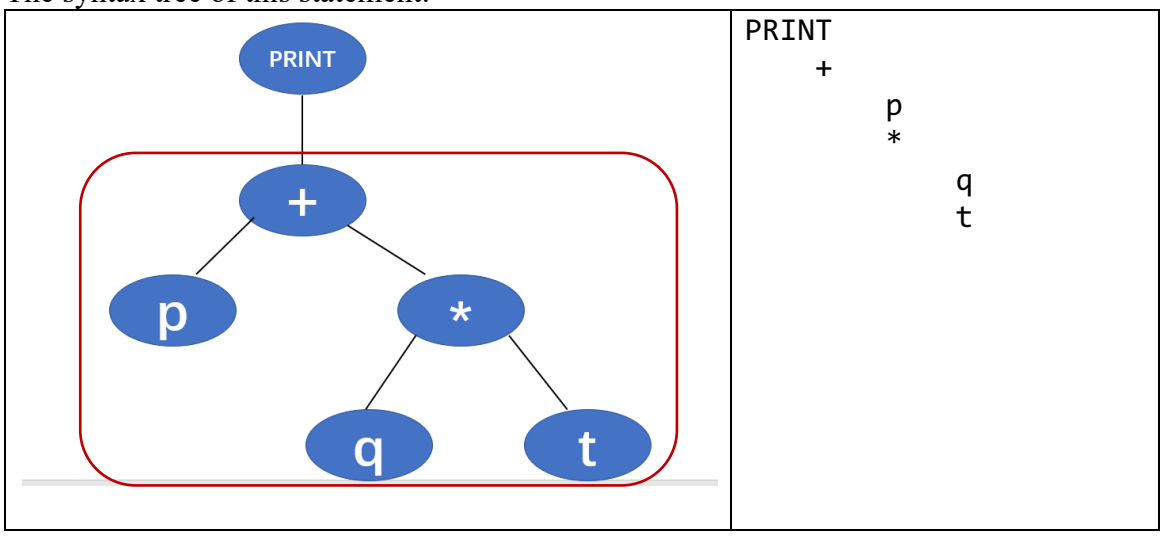


Figure 6

As what you can observe, the nodes at the same level in the tree are at the same vertical line in the indentation notation.

Each indent contains 4 spaces to make the structure of the syntax tree clear enough.

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
```

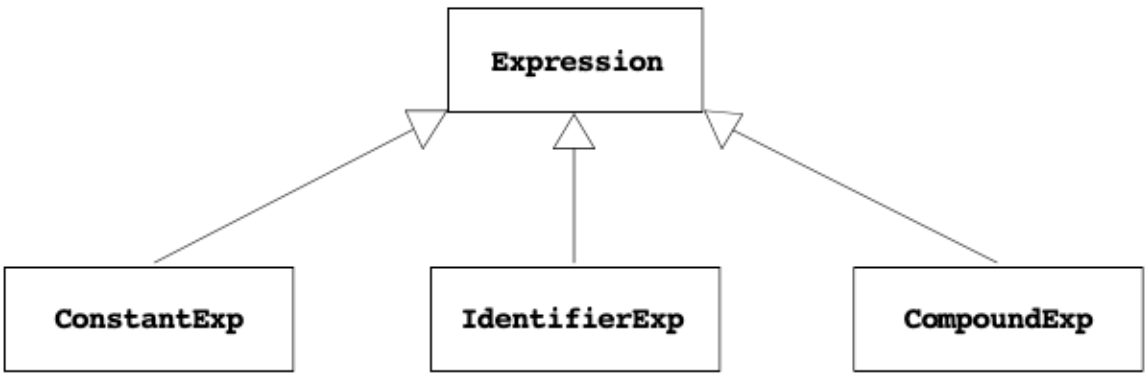
your interpreter has to store that line in its internal data structure so that it becomes part of the current program. As you type the rest of the lines from the program in Figure 2, the data structure inside your implementation must add the new lines and keep track of the sequence. In particular, when you correct the program by typing

```
145 PRINT n1
```

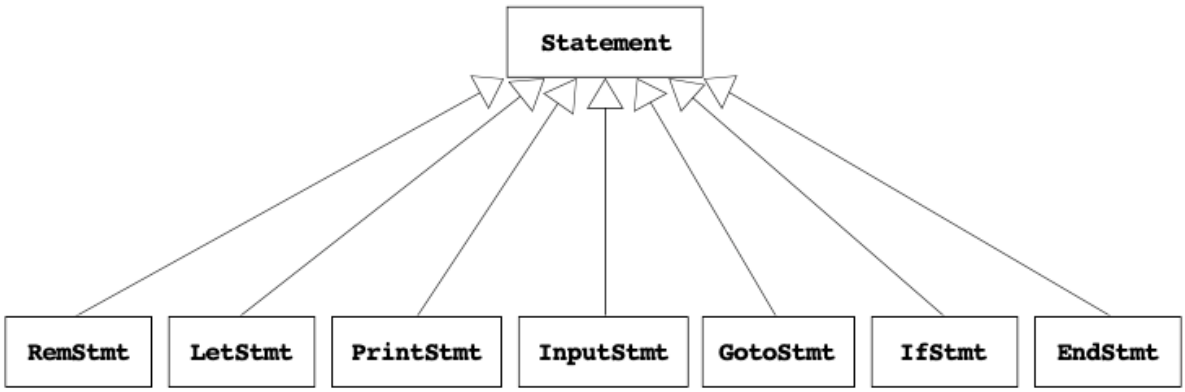
your data structure must know that this line goes between lines 140 and 150 in the existing program.

Hints on the statement class hierarchy

The primary class for expressions should be **`Expression`**, which is the abstract superclass for a hierarchy that includes three concrete subclasses for the three different expression types, as follows:



The structure of the statements is quite similar. The primary class should be **`Statement`**, which 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 `Statement` hierarchy, it is still somewhat easier to implement than the `Expression` hierarchy. One of the things that makes the `Expression` 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 statements in modern languages like C++ *are* recursive, statements in BASIC are not.

Exception handling

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. Thus, you need to use `try`/`catch` so that your interpreter responds to errors much more gracefully.

Strategy and tactics

As you work through this project, you might want to keep in mind the following bits of advice:

- The last line of the fictional memo from Bill Gates encourages his team to "get going on this project as soon as possible." I encourage you to adopt that same strategy.
- You need to first create a project and initialize the GUI. You can use the UI editor of Qt Creator or place the UI widgets by C++ code. To finish the Minimal BASIC project, you need to proceed strategically 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.
- Make sure you get the project working before you embark on extensions. It's easy to get too ambitious at the beginning and end up with a mass of code that proves impossible to debug.

Grading

- **(10')** Your interpreter should be able to present a GUI and interact with user input.
  - GUI should contain the input and output interfaces shown in Figure 2.
- **(20')** Your interpreter should be able to load and edit basic programs.
  - Users can add, update or delete statements through input box or LOAD button.
  - The statements entered by user can be stored and displayed in the correct order.
- **(50')** Your interpreter should be able to interpret basic programs correctly.

  - Expression parsing (display the syntax tree, although this should be done when you store the programs);
  - Expression evaluation and statement execution (display the result of print if exists);
  - Runtime context maintenance (e.g., the current line to be executed, all variables and their values).
- **(10')** Your interpreter should be robust and correctly handle errors in the input.
- **(10')** You should finish the project with object-oriented design and implementation; your code should be clear and easy to read with appropriate comments.

Hints:

You could learn something from this article (https://doc.qt.io/archives/qq/qq27-responsive-guis.html). But there are always other ways to achieve the same goal, and you are not restricted by the article.

Side Quests

We have several optional tasks available to you so that you can extend your minibasic program if you like. Implementing these tasks **won't** get you additional scores. But they do make your minibasic look better.

1. Add support for string data types. String literals are presents within double quotation marks:

```
"1.234str_123xxx"
```

   Strings should at least support two operations:

```
"string" + exp1
"string" * exp2
```

   The "+" operator turns the second expression to strings and append it to the first string. The "*" operator repeats the first string by the times indicated by the second parameter. The parameter should be an integer, otherwise errors should be reported. Here are some examples:

```
"string" + 123 => "string123"
"string" + "str" => "stringstr"
"string" * 2 => "stringstring"
```

2. Add support for functions. Users can define a function as follows:

```
Sub function_name
```

```
    Statements...
End Sub
```

The parameters should be at fixed times.

There is only a global variable namespace. Thus, the function can use variables defined outside the function, and variables defined inside the function can also be accessed in other parts of the code. As a result, you can use variables to pass parameters.

The function can be invoked as follows:

```
CALL function_name
```

For simplicity, you don't need treat a function invocation as an expression. The following is an example that should print 3 after execution.

```
Sub myadd
    LET C = A + B
END Sub
LET A = 1
LET B = 2
CALL myadd
PRINT C
```

3. Add support for calling functions in C libraries. You need to add a new command NATIVE_C_LIB to load the C library indicated by the path. There can be multiple libraries to be loaded at the same time.

```
NATIVE_C_LIB "path-to-c-library-file"
```

Since C functions have different form from our function, e.g., C functions have explicit parameters and return values, we add special variables `c_para1`, `c_para2`, …, to hold values that should be passed to the C function. We also add a special variable `c_retval` as the variable to hold return value of C functions. Below is an example which should print 3 after execution.

```
/* C source code file that compiles to mycode.a */
int myadd2(int a, int b)
{
    return a + b;
}
```

```
NATIVE_C_LIB "/path/to/mycode.a"
LET c_para1 = 1
LET c_para2 = 2
CALL myadd2
PRINT c_retval
```

4. Add support for interactive debugging that contains at least the following features:

    1) Step-in/Continue execution, and show current line to be executed via GUI
    2) Set/Show/Delete breakpoints
    3) Show real-time variable names and values via GUI

    You need to add customized commands for the interpreter such as DEBUG and STEP.