6 Example: BASIC Compiler

Program file for this chapter: `basic`

The BASIC programming language was designed by John Kemeny and Thomas Kurtz in the late 1960s. (The name is an acronym for Beginner's All-purpose Symbolic Instruction Code.) It was first implemented on a large, central computer facility at Dartmouth; the designers' goal was to have a language that all students could use for simple problems, in contrast to the arcane programming languages used by most experts at that time.

A decade later, when the microcomputer was invented, BASIC took on a new importance. Kemeny and Kurtz designed a simple language for the sake of the users, but that simplicity also made the language easy for the *computer!* Every programming language requires a computer program to translate it into instructions that the computer can carry out. For example, the Logo programs you write are translated by a Logo interpreter. But Logo is a relatively complex language, and a Logo interpreter is a pretty big program. The first microcomputers had only a few thousand bytes of memory. (Today's home computers, by contrast, have several million bytes.) Those early personal computers couldn't handle Logo, but it was possible to write a BASIC interpreter that would fit them. As a result, BASIC became the near-universal language for amateur computer enthusiasts in the late 1970s and early 1980s.

Today's personal computers come with translators for a wide variety of programming languages, and also with software packages that enable many people to accomplish their computing tasks without writing programs of their own at all. BASIC is much less widely used today, although it has served as the core for Microsoft's "Visual Basic" language.

In this chapter, I want to show how Logo's `define` command can be used in a program-writing program. My program will translate BASIC programs into Logo programs. I chose BASIC for the same reason the early microcomputers used it: It's a small language and the translator is relatively easy to write. (Kemeny and Kurtz, the designers of BASIC, have criticized the microcomputer implementations as *too* simple

and as unfaithful to their original goals. My implementation will share that defect, to make the project easier. Don't use this version as a basis on which to judge the language! For that you should investigate True Basic, the version that Kemeny and Kurtz wrote themselves for personal computers.)

Here's a typical short BASIC program:

```
10 print "Table of Squares"
20 print
30 print "How many values would you like?"
40 input num
50 for i=1 to num
60 print i, i*i
70 next i
80 end
```

And here's what happens when we run it:

```
Table of Squares

How many values would you like?
5
1       1
2       4
3       9
4       16
5       25
```

A Short Course in BASIC

Each line in the sample BASIC program begins with a *line number*. These numbers are used for program editing. Instead of the modern screen editors with which you're familiar, the early versions of BASIC had a very primitive editing facility; you could replace a line by typing a new line with the same number. There was no way to replace less than an entire line. To delete a line completely, you'd enter a line containing just the number. The reason the line numbers in this program are multiples of ten is to leave room for inserting new lines. For example, I could say

```
75 print "Have a nice day."
```

to insert a new line between lines 70 and 80. (By the way, the earliest versions of Logo used a similar line numbering system, except that each Logo procedure was separately

numbered. The editing technique isn't really part of the language design; early systems used "line editors" because they had typewriter-like paper terminals instead of today's display screens. I'm using a line editor in this project because it's easy to implement!)

The BASIC language consists of one or two dozen commands, depending on the version used. My BASIC dialect understands only these ten commands:

```
LET variable = value
PRINT values
INPUT variables
FOR variable = value TO value
NEXT variable
IF value THEN command
GOTO linenumber
GOSUB linenumber
RETURN
END
```

Unlike Logo procedure calls, which consist of the procedure name followed by inputs in a uniform format, each BASIC command has its own format, sometimes including internal separators such as the equal sign and the word `to` in the `for` command format, or the word `then` in the `if` command format.

In some versions of BASIC, including this one, a single line can contain more than one command, if the commands are separated with colons. Thus the same program shown earlier could also be written this way:

```
10 print "Table of Squares":print
30 print "How many values would you like?":input num
50 for i=1 to num : print i, i*i : next i
80 end
```

The `let` command assigns a value to a variable, like Logo's `make` procedure. Unlike Logo, BASIC does not have the rule that all inputs are evaluated before applying the command. In particular, the word after `let` must be the name of the variable, not an expression whose value is the name. Therefore the name is not quoted. Also, a variable can't have the same name as a procedure, so there is no need for anything like Logo's use of the colon to indicate a variable value. (This restricted version of BASIC doesn't have named procedures at all, like some early microcomputer versions.)

```
make "x :y + 3        (Logo)
let x = y + 3         (BASIC)
```

In my subset of BASIC, the value of a variable must be a number. More complete BASIC dialects include string variables (like words in Logo) and arrays (like Logo's arrays).

The value to be assigned to a variable can be computed using an arithmetic expression made up of variables, numbers, the arithmetic operators `+`, `-`, `*`, and `/`, and parentheses for grouping.

The `print` command is similar to Logo's print procedure in that it prints a line on the screen. That line can include any number of values. Here is an example `print` command:

```
print "x = "; x, "y = "; y, "sum = "; x+y
```

In this example two kinds of values are printed: arithmetic values (as in the `let` command) and strings. A *string* is any sequence of characters surrounded by quotation marks.

Notice that the values in this example are separated by punctuation marks, either commas or semicolons. When a semicolon is used, the two values are printed right next to each other, with no space between them. (That's why each of the strings in this example ends with a space.) When a comma is used, BASIC prints a tab character between the two values, so that values on different lines will line up to form columns. (Look again at the table of squares example at the beginning of this chapter.)

The `input` command is the opposite of `print`; it reads values from the keyboard and assigns them to variables. There is nothing in Logo exactly like `input`. Instead, Logo has *operations* `readword` and `readlist` that output the contents of a line; those values can be assigned to variables using `make` or can be used in some other way. The Logo approach is more flexible, but the early versions of BASIC didn't have anything like Logo's operations. The `input` command will also accept a string in quotation marks before its list of variables; that string is printed as a prompt before BASIC reads from the keyboard. (BASIC does not start a new line after printing the prompt, so the effect is like Logo's `type` command rather than like `print`.) Here's an example:

```
input "Please enter x and y: " x,y
```

The user can type the values for x and y on the same line, separated by spaces, or on separate lines. BASIC keeps reading lines until it has collected enough numbers for the listed variables. Notice that the variable names in the `input` command must be separated by commas, not by semicolons.

The `for` and `next` commands work together to provide a numeric iteration capability like Berkeley Logo's `for` procedure. The `for` command format includes a

variable name, a starting value, and an ending value. (The step value is always 1.) The named variable is given the specified starting value. If that value is less than the ending value, then all of the commands between the `for` command and the matching `next` command (the one with the same named variable) are carried out. Then the variable is increased by 1, and the process continues until the ending value is reached. `For` and `next` pairs with different variables can be nested:

```
10 input "Input size: " num
20 for i = 1 to num
30 for j = i to num
40 print i;"  ";j
50 next j:next i
60 end

Input size: 4
1  1
1  2
1  3
1  4
2  2
2  3
2  4
3  3
3  4
4  4
```

Notice that the `next j` must come before the `next i` so that the `for/next` pairs are properly nested.

The `if` command allows conditional execution, much like Logo's `if` command, but with a different notation. Instead of taking an instruction list as an input, BASIC's `if` uses the keyword `then` to introduce a single conditional command. (If you want to make more than one command conditional, you must combine `if` with `goto`, described next.) The value that controls the `if` must be computed using one of the operators `=`, `<`, or `>` for numeric comparison.*

* Notice that the equal sign has two meanings in BASIC. In the `let` command, it's like Logo's `make`; in the `if` command, it's like Logo's `equalp`. In the early 1980s, Logo enthusiasts had fierce arguments with BASIC fans, and this sort of notational inconsistency was one of the things that drove us crazy! (More serious concerns were the lack of operations and of recursion in the microcomputer versions of BASIC.)

The `goto` command transfers control to the beginning of a command line specified by its line number. It can be used with `if` to make a sequence of commands conditional:

```
10 input x
20 if x > 0 then goto 100
30 print "x is negative."
40 print "x = "; x
50 goto 200
100 print "x is positive."
200 end
```

The `gosub` and `return` commands provide a rudimentary procedure calling mechanism. I call it "rudimentary" because the procedures have no inputs, and can only be commands, not operations. Also, the command lines that make up the procedure are also part of the main program, so you generally need a `goto` in the main program to skip over them:

```
10 let x=7
20 gosub 100
30 let x=9
40 gosub 100
50 goto 200
100 print x, x*x
110 return
200 end
```

Finally, the `end` command ends the program. There must be an `end` at the end of a BASIC program, and there should not be one anywhere else. (In this implementation of BASIC, an `end` stops the BASIC program even if there are more lines after it. It's roughly equivalent to a `throw` to `toplevel` in Logo.)

Using the BASIC Translator

To start the translator, run the Logo procedure `basic` with no inputs. You will then see the BASIC prompt, which is the word `READY` on a line by itself.

At the prompt you can do either of two things. If you type a line starting with a line number, that line will be entered into your BASIC program. It is inserted in order by line number. Any previous line with the same number will be deleted. If the line you type contains *only* a line number, then the line in the program with that number will be deleted.

If your line does not start with a number, then it is taken as an *immediate* command, not as part of the program. This version of BASIC recognizes only three immediate commands: The word `run` means to run your program, starting from the smallest line number. The word `list` means to print out a listing of the program's lines, in numeric order. The word `exit` returns to the Logo prompt.

Overview of the Implementation

There are two kinds of translators for programming languages: compilers and interpreters. The difference is that a compiler translates one language (the *source* language) into another (the *target* language), leaving the result around so that it can be run repeatedly without being translated again. An interpreter translates each little piece of source language into one action in the target language and runs the result, but does not preserve a complete translated program in the target language.

Ordinarily, the target language for both compilers and interpreters is the "native" language of the particular computer you're using, the language that is wired into the computer hardware. This *machine language* is the only form in which a program can actually be run. The BASIC compiler in this chapter is quite unrealistic in that it uses Logo as the target language, which means that the program must go through *another* translation, from Logo to machine language, before it can actually be run. For our purposes, there are three advantages to using Logo as the target language. First, every kind of computer has its own machine language, so I'd have to write several versions of the compiler to satisfy everyone if I compiled BASIC into machine language. Second, I know you know Logo, so you can understand the resulting program, whereas you might not be familiar with any machine language. Third, this approach allows me to cheat by leaving out a lot of the complexity of a real compiler. Logo is a "high level" language, which means that it takes care of many details for us, such as the allocation of specific locations in the computer's memory to hold each piece of information used by the program. In order to compile into machine language, I'd have to pay attention to those details.

Why would anyone want an interpreter, if the compiler translates the program once and for all, while the interpreter requires retranslation every time a command is carried out? One reason is that an interpreter is easier to write, because (just as in the case of a compiler with Logo as the target language) many of the details can be left out. Another reason is that traditional compilers work using a *batch* method, which means that you must first write the entire program with a text editor, then run the compiler to translate the program into machine language, and finally run the program. This is okay

for a working program that is used often, but not recompiled often. But when you're creating a program in the first place, there is a debugging process that requires frequent modifications to the source language program. If each modification requires a complete recompilation, the debugging is slow and frustrating. That's why interpreted languages are often used for teaching—when you're learning to program, you spend much more time debugging a program than running the final version.

The best of both worlds is an *incremental compiler,* a compiler that can recompile only the changed part when a small change is made to a large program. For example, Object Logo is a commercial version of Logo for the Macintosh in which each procedure is compiled when it is defined. Modifying a procedure requires recompiling that procedure, but not recompiling the others. Object Logo behaves like an interpreter, because the user doesn't have to ask explicitly for a procedure to be compiled, but programs run faster in Object Logo than in most other versions because each procedure is translated only once, rather than on every invocation.

The BASIC translator in this chapter is an incremental compiler. Each numbered line is compiled into a Logo procedure as soon as it is typed in. If the line number is `40` then the resulting procedure will be named `basic%40`. The last step in each of these procedures is to invoke the procedure for the next line. The compiler maintains a list of all the currently existing line numbers, in order, so the `run` command is implemented by saying

```
run (list (word "basic% first :linenumbers))
```

Actually, what I just said about each procedure ending with an invocation of the next one is slightly simplified. Suppose the BASIC program starts

```
10 let x=3
20 let y=9
30 ...
```

and we translate that into

```
to basic%10
make "%x 3
basic%20
end
```

```
to basic%20
make "%y 9
basic%30
end
```

Then what happens if the user adds a new line numbered 15? We would have to recompile line 10 to invoke `basic%15` instead of `basic%20`. To avoid that, each line is compiled in a way that defers the choice of the next line until the program is actually run:

```
to basic%10
make "%x 3
nextline 10
end

to basic%20
make "%y 9
nextline 20
end
```

This solution depends on a procedure `nextline` that finds the next available line number after its argument:

```
to nextline :num
make "target member :num :linenumbers
if not emptyp :target [make "target butfirst :target]
if not emptyp :target [run (list (word "basic% first :target))]
end
```

`Nextline` uses the Berkeley Logo primitive `member`, which is like the predicate `memberp` except that if the first input is found as a member of the second, instead of giving `true` as its output, it gives the portion of the second input starting with the first input:

```
? show member "the [when in the course of human events]
[the course of human events]
```

If the first input is not a member of the second, `member` outputs an empty word or list, depending on the type of the second input.

The two separate `emptyp` tests are used instead of a single `if` because the desired line number might not be in the list at all, or it might be the last one in the list, in which case the `butfirst` invocation will output an empty list. (Neither of these cases should arise. The first means that we're running a line that doesn't exist, and the second means

that the BASIC program doesn't end with an `end` line. But the procedure tries to avoid disaster even in these cases.)

Look again at the definition of `basic%10`. You'll see that the variable named `x` in the BASIC program is named `%x` in the Logo translation. The compiler uses this renaming technique to ensure that the names of variables and procedures in the compiled program don't conflict with names used in the compiler itself. For example, the compiler uses a variable named `linenumbers` whose value is the list of line numbers. What if someone writes a BASIC program that says

```
10 let linenumbers = 100
```

This won't be a problem because in the Logo translation, that variable will be named `%linenumbers`.

The compiler can be divided conceptually into four parts:

- The *reader* divides the characters that the user types into meaningful units. For example, it recognizes that `let` is a single word, but `x+1` should be understood as three separate words.
- The *parser* recognizes the form of each of the ten BASIC commands that this dialect understands. For example, if a command starts with `if`, the parser expects an expression followed by the word `then` and another command.
- The *code generator* constructs the actual translation of each BASIC command into one or more Logo instructions.
- The *runtime library* contains procedures that are used while the translated program is running, rather than during the compilation process. The `nextline` procedure discussed earlier is an example.

Real compilers have the same structure, except of course that the code generator produces machine language instructions rather than Logo instructions. Also, a professional compiler will include an *optimizer* that looks for ways to make the compiled program as efficient as possible.

The Reader

A *reader* is a program that reads a bunch of characters (typically one line, although not in every language) and divides those characters into meaningful units. For example, every

Logo implementation includes a reader that interprets square brackets as indications of list grouping. But some of the rules followed by the Logo reader differ among implementations. For example, can the hyphen character (`-`) be part of a larger word, or is it always a word by itself? In a context in which it means subtraction, we'd like it to be a word by itself. For example, when you say

```
print :x-3
```

as a Logo instruction, you mean to print three less than the value of the variable named `x`, not to print the value of a variable whose name is the three-letter word `x-3`! On the other hand, if you have a list of telephone numbers like this:

```
make "phones [555-2368 555-9827 555-8311]
```

you'd like the `first` of that list to be an entire phone number, the word `555-2368`, not just `555`. Some Logo implementations treat every hyphen as a word by itself; some treat every hyphen just like a letter, and require that you put spaces around a minus sign if you mean subtraction. Other implementations, including Berkeley Logo, use a more complicated rule in which the status of the hyphen depends on the context in which it appears, so that both of the examples in this paragraph work as desired.

In any case, Logo's reader follows rules that are not appropriate for BASIC. For example, the colon (`:`) is a delimiter in BASIC, so it should be treated as a word by itself; in Logo, the colon is paired with the variable name that follows it. In both languages, the quotation mark (`"`) is used to mark quoted text, but in Logo it comes only at the beginning of a word, and the quoted text ends at the next space character, whereas in BASIC the quoted text continues until a second, matching quotation mark. For these and other reasons, it's desirable to have a BASIC-specific reader for use in this project.

The rules of the BASIC reader are pretty simple. Each invocation of `basicread` reads one line from the keyboard, ending with the Return or Enter character. Within that line, space characters separate words but are not part of any word. A quotation mark begins a quoted word that includes everything up to and including the next matching quotation mark. Certain characters form words by themselves:

```
+ - * / = < > ( ) , ; :
```

All other characters are treated like letters; that is, they can be part of multi-character words.

```
? show basicread
30 print x;y;"foo,baz",z:print hello+4
[30 print x ; y ; "foo,baz" , z : print hello + 4]
```

Notice that the comma inside the quotation marks is not made into a separate word by `basicread`. The other punctuation characters, however, appear in the output sentence as one-character words.

`Basicread` uses the Logo primitive `readword` to read a line. `Readword` can be thought of as a reader with one trivial rule: The only special character is the one that ends a line. Everything else is considered as part of a single long word. `Basicread` examines that long word character by character, looking for delimiters, and accumulating a sentence of words separated according to the BASIC rules. The implementation of `basicread` is straightforward; you can read the procedures at the end of this chapter if you're interested. For now, I'll just take it for granted and go on to discuss the more interesting parts of the BASIC compiler.

The Parser

The *parser* is the part of a compiler that figures out the structure of each piece of the source program. For example, if the BASIC compiler sees the command

```
let x = ( 3 * y ) + 7
```

it must recognize that this is a `let` command, which must follow the pattern

```
LET variable = value
```

and therefore `x` must be the name of a variable, while `( 3 * y ) + 7` must be an expression representing a value. The expression must be further parsed into its component pieces. Both the variable name and the expression must be translated into the form they will take in the compiled (Logo) program, but that's the job of the code generator.

In practice, the parser and the code generator are combined into one step; as each piece of the source program is recognized, it is translated into a corresponding piece of the object program. So we'll see that most of the procedures in the BASIC compiler include parsing instructions and code generation instructions. For example, here is the procedure that compiles a `let` command:

```
to compile.let :command
make "command butfirst :command
make "var pop "command
make "delimiter pop "command
if not equalp :delimiter "= [(throw "error [Need = in let.])]
make "exp expression
queue "definition (sentence "make (word ""% :var) :exp)
end
```

In this procedure, all but the last instruction (the line starting with `queue`) are parsing the source command. The last line, which we'll come back to later, is generating a Logo `make` instruction, the translation of the BASIC `let` in the object program.

BASIC was designed to be very easy to parse. The parser can read a command from left to right, one word at a time; at every moment, it knows exactly what to expect. The command must begin with one of the small number of command names that make up the BASIC language. What comes next depends on that command name; in the case of `let`, what comes next is one word (the variable name), then an equal sign, then an expression. Each instruction in the `compile.let` procedure handles one of these pieces. First we skip over the word `let` by removing it from the front of the command:

```
make "command butfirst :command
```

Then we read and remember one word, the variable name:

```
make "var pop "command
```

(Remember that the `pop` operation removes one member from the beginning of a list, returning that member. In this case we are removing the variable name from the entire `let` command.) Then we make sure there's an equal sign:

```
make "delimiter pop "command
if not equalp :delimiter "= [(throw "error [Need = in let.])]
```

And finally we call a subprocedure to read the expression; as we'll see later, that procedure also translates the expression to the form it will take in the object program:

```
make "exp expression
```

The parsers for other BASIC commands have essentially the same structure as this example. They repeatedly invoke `pop` to read one word from the command or `expression` to read and translate an expression. (The `if` command is a little more

complicated because it contains another command as a component, but that inner command is just compiled as if it occurred by itself. We'll look at that process in more detail when we get to the code generation part of the compiler.)

Each compilation procedure expects a single BASIC command as its input. Remember that a line in a BASIC program can include more than one command. The compiler uses a procedure named `split` to break up each line into a list of commands:

```
? show split [30 print x ; y ; "foo,baz" , z : print hello + 4]
[30 [print x ; y ; "foo,baz" , z] [print hello + 4]]
```

`Split` outputs a list whose first member is a line number; the remaining members are lists, each containing one BASIC command. `Split` works by looking for colons within the command line.

Here is the overall structure of the compiler, but with only the instructions related to parsing included:

```
to basic
forever [basicprompt]
end

to basicprompt
print "READY
make "line basicread
if emptyp :line [stop]
ifelse numberp first :line [compile split :line] [immediate :line]
end

to compile :commands
make "number first :commands
ifelse emptyp butfirst :commands ~
       [eraseline :number] ~
       [makedef (word "basic% :number) butfirst :commands]
end

to makedef :name :commands
...
foreach :commands [run list (word "compile. first ?) ?]
...
end
```

`Basic` does some initialization (not shown) and then invokes `basicprompt` repeatedly. `Basicprompt` calls the BASIC reader to read a line; if that line starts with a number, then `split` is used to transform the line into a list of commands, and `compile` is invoked with that list as input. `Compile` remembers the line number for later use, and then invokes `makedef` with the list of commands as an input. I've left out most of the instructions in `makedef` because they're concerned with code generation, but the important part right now is that for each command in the list, it invokes a procedure named `compile.`something based on the first word of the command, which must be one of the command names in the BASIC language.

The Code Generator

Each line of the BASIC source program is going to be compiled into one Logo procedure. (We'll see shortly that the BASIC `if` and `for` commands are exceptions.) For example, the line

```
10 let x = 3 : let y = 4 : print x,y+6
```

will be compiled into the Logo procedure

```
to basic%10
make "%x 3
make "%y 4
type :%x
type char 9
type :%y + 6
print []
nextline 10
end
```

Each of the three BASIC commands within the source line contributes one or more instructions to the object procedure. Each `let` command is translated into a `make` instruction; the `print` command is translated into three `type` instructions and a `print` instruction. (The last instruction line in the procedure, the invocation of `nextline`, does not come from any of the BASIC commands, but is automatically part of the translation of every BASIC command line.)

To generate this object procedure, the BASIC compiler is going to have to invoke Logo's `define` primitive, this way:

```
define "basic%10 [[] [make "%x 3] [make "%y 4] ... [nextline 10]]
```

Of course, these actual inputs do not appear explicitly in the compiler! Rather, the inputs to `define` are variables that have the desired values:

```
define :name :definition
```

The variable `name` is an input to `makedef`, as we've seen earlier. The variable `definition` is created within `makedef`. It starts out as a list containing just the empty list, because the first sublist of the input to `define` is the list of the names of the desired inputs to `basic%10`, but it has no inputs. The procedures within the compiler that parse each of the commands on the source line will also generate object code (that is, Logo instructions) by appending those instructions to the value of `definition` using Logo's `queue` command. `Queue` takes two inputs: the name of a variable whose value is a list, and a new member to be added at the end of the list. Its effect is to change the value of the variable to be the extended list.

Look back at the definition of `compile.let` above. Earlier we considered the parsing instructions within that procedure, but deferred discussion of the last instruction:

```
queue "definition (sentence "make (word ""% :var) :exp)
```

Now we can understand what this does: It generates a Logo `make` instruction and appends that instruction to the object procedure definition in progress.

We can now also think about the output from the `expression` procedure. Its job is to parse a BASIC expression and to translate it into the corresponding Logo expression. This part of the compiler is one of the least realistic. A real compiler would have to think about such issues as the precedence of arithmetic operations; for example, an expression like `3+x*4` must be translated into two machine language instructions, first one that multiplies `x` by 4, and then one that adds the result of that multiplication to 3. But the Logo interpreter already handles that aspect of arithmetic for us, so all `expression` has to do is to translate variable references like `x` into the Logo form `:%x`.

```
? show expression [3 + x * 4]
[3 + :%x * 4]
```

(We'll take a closer look at translating arithmetic expressions in the Pascal compiler found in the third volume of this series, *Beyond Programming*.)

We are now ready to look at the complete version of `makedef`:

```
to makedef :name :commands
make "definition [[]]
foreach :commands [run list (word "compile. first ?) ?]
queue "definition (list "nextline :number)
define :name :definition
make "linenumbers insert :number :linenumbers
end
```

I hope you'll find this straightforward. First we create an empty definition. Then, for each BASIC command on the line, we append to that definition whatever instructions are generated by the code generating instructions for that command. After all the BASIC commands have been compiled, we add an invocation of `nextline` to the definition. Now we can actually define the Logo procedure whose text we've been accumulating. The last instruction updates the list of line numbers that `nextline` uses to find the next BASIC command line when the compiled program is running.

In a sense, this is the end of the story. My purpose in this chapter was to illustrate how `define` can be used in a significant project, and I've done that. But there are a few more points I should explain about the code generation for some specific BASIC commands, to complete your understanding of the compiler.

One such point is about the difference between `goto` and `gosub`. Logo doesn't have anything like a `goto` mechanism; both `goto` and `gosub` must be implemented by invoking the procedure corresponding to the given line number. The difference is that in the case of `goto`, we want to invoke that procedure and not come back! The solution is to compile the BASIC command

```
goto 40
```

into the Logo instructions

```
basic%40 stop
```

In effect, we are calling line 40 as a subprocedure, but when it returns, we're finished. Any additional Logo instructions generated for the same line after the `goto` (including the invocation of `nextline` that's generated automatically for every source line) will be ignored because of the `stop`.*

* In fact, the Berkeley Logo interpreter is clever enough to notice that there is a `stop` instruction after the invocation of `basic%40`, and it arranges things so that there is no "return" from that procedure. This makes things a little more efficient, but doesn't change the meaning of the program.

The next tricky part of the compiler has to do with the `for` and `next` commands. Think first about `next`. It must increment the value of the given variable, test that value against a remembered limit, and, if the limit has not been reached, go to... where? The `for` loop continues with the BASIC command just after the `for` command itself. That might be in the middle of a line, so `next` can't just remember a line number and invoke `basic%N` for line number N. To solve this problem, the line containing the `for` command is split into two Logo procedures, one containing everything up to and including the `for`, and one for the rest of the line. For example, the line

```
30 let x = 3 : for i = 1 to 5 : print i,x : next i
```

is translated into

```
to basic%30
make "%x 3
make "%i 1
make "let%i 5
make "next%i [%g1]
%g1
end

to %g1
type :%i
type char 9
type :%x
print []
make "%i :%i + 1
if not greaterp :%i :let%i [run :next%i stop]
nextline 30
end
```

The first `make` instruction in `basic%30` is the translation of the `let` command. The remaining four lines are the translation of the `for` command; it must give an initial value to the variable `i`, remember the limit value 5, and remember the Logo procedure to be used for looping. That latter procedure is named `%g1` in this example. The percent sign is used for the usual reason, to ensure that the names created by the compiler don't conflict with names in the compiler itself. The `g1` part is a *generated symbol*, created by invoking the Berkeley Logo primitive operation `gensym`. Each invocation of `gensym` outputs a new symbol, first `g1`, then `g2`, and so on.

The first four instructions in procedure `%g1` (three `type`s and a `print`) are the translation of the BASIC `print` command. The next two instructions are the translation

of the `next` command; the `make` instruction increments `i`, and the `if` instruction tests whether the limit has been passed, and if not, invokes the looping procedure `%g1` again. (Why does this say `run :next%i` instead of just `%g1`? Remember that the name `%g1` was created during the compilation of the `for` command. When we get around to compiling the `next` command, the code generator has no way to remember which generated symbol was used by the corresponding `for`. Instead it makes reference to a variable `next%i`, named after the variable given in the `next` command itself, whose value is the name of the procedure to run. Why not just call that procedure itself `next%i` instead of using a generated symbol? The trouble is that there might be more than one pair of `for` and `next` commands in the same BASIC program using the same variable, and each of them must have its own looping procedure name.)

There is a slight complication in the `print` and `input` commands to deal with quoted character strings. The trouble is that Logo's idea of a word ends with a space, so it's not easy to translate

```
20 print "hi there"
```

into a Logo instruction in which the string is explicitly present in the instruction. Instead, the BASIC compiler creates a Logo global variable with a generated name, and uses that variable in the compiled Logo instructions.

The trickiest compilation problem comes from the `if` command, because it includes another command as part of itself. That included command might be translated into several Logo instructions, all of which should be made to depend on the condition that the `if` is testing. The solution is to put the translation of the inner command into a separate procedure, so that the BASIC command line

```
50 if x<6 then print x, x*x
```

is translated into the two Logo procedures

```
to basic%50
if :%x < 6 [%g2]
nextline 50
end

to %g2
type :%x
type char 9
type :%x * :%x
print []
end
```

Unfortunately, this doesn't quite work if the inner command is a `goto`. If we were to translate

```
60 if :foo < 10 then goto 200
```

into

```
to basic%60
if :%foo < 10 [%g3]
nextline 60
end

to %g3
basic%200 stop
end
```

then the `stop` inside `%g3` would stop only `%g3` itself, not `basic%60` as desired. So the code generator for `if` checks to see whether the result of compiling the inner command is a single Logo instruction line; if so, that line is used directly in the compiled Logo `if` rather than diverted into a subprocedure:

```
to basic%60
if :%foo < 10 [basic%200 stop]
nextline 60
end
```

How does the code generator for `if` divert the result of compiling the inner command away from the definition of the overall BASIC command line? Here is the relevant part of the compiler:

```
to compile.if :command
make "command butfirst :command
make "exp expression
make "delimiter pop "command
if not equalp :delimiter "then [(throw "error [Need then after if.])]
queue "definition (sentence "if :exp (list c.if1))
end
```

```
to c.if1
local "definition
make "definition [[]]
run list (word "compile. first :command) :command
ifelse (count :definition) = 2 ~
       [output last :definition] ~
       [make "newname word "% gensym
        define :newname :definition
        output (list :newname)]
end
```

The first few lines of this are straightforwardly parsing the part of the BASIC `if` command up to the word `then`. What happens next is a little tricky; a subprocedure `c.if1` is invoked to parse and translate the inner command. It has to be a subprocedure because it creates a local variable named `definition`; when the inner command is compiled, this local variable "steals" the generated code. If there is only one line of generated code, then `c.if1` outputs that line; if more than one, then `c.if1` creates a subprocedure and outputs an instruction to invoke that subprocedure. This technique depends on Logo's dynamic scope, so that references to the variable named `definition` in other parts of the compiler (such as, for example, `compile.print` or `compile.goto`) will refer to this local version.

The Runtime Library

We've already seen the most important part of the runtime library: the procedure `nextline` that gets the compiled program from one line to the next.

There is only one more procedure needed as runtime support; it's called `readvalue` and it's used by the BASIC `input` command. In `BASIC`, data input is independent of lines. If a single `input` command includes two variables, the user can type the two desired values on separate lines or on a single line. Furthermore, two *separate* `input` commands can read values from a single line, if there are still values left on the line after the first `input` has been satisfied. `Readvalue` uses a global variable `readline` whose value is whatever's still available from the last data input line, if any. If there is nothing available, it reads a new line of input.

A more realistic BASIC implementation would include runtime library procedures to compute built-in functions (the equivalent to Logo's primitive operations) such as absolute value or the trigonometric functions.

Further Explorations

This BASIC compiler leaves out many features of a complete implementation. In a real BASIC, a string can be the value of a variable, and there are string operations such as concatenation and substring extraction analogous to the arithmetic operations for numbers. The BASIC programmer can create an array of numbers, or an array of strings. In some versions of BASIC, the programmer can define named subprocedures, just as in Logo. For the purposes of this chapter, I wanted to make the compiler as simple as possible and still have a usable language. If you want to extend the compiler, get a BASIC textbook and start implementing features.

It's also possible to expand the immediate command capabilities of the compiler. In most BASIC implementations, for example, you can say `list 100-200` to list only a specified range of lines within the source program.

A much harder project would be to replace the code generator in this compiler with one that generates machine language for your computer. Instead of using `define` to create Logo procedures, your compiler would then write machine language instructions into a data file. To do this, you must learn quite a lot about how machine language programs are run on your computer!

Program Listing

I haven't discussed every detail of the program. For example, you may want to trace through what happens when you ask to delete a line from the BASIC source program. Here is the complete compiler.

```
to basic
make "linenumbers []
make "readline []
forever [basicprompt]
end

to basicprompt
print []
print "READY
print []
make "line basicread
if emptyp :line [stop]
ifelse numberp first :line [compile split :line] [immediate :line]
end
```

```
to compile :commands
make "number first :commands
make :number :line
ifelse emptyp butfirst :commands ~
       [eraseline :number] ~
       [makedef (word "basic% :number) butfirst :commands]
end

to makedef :name :commands
make "definition [[]]
foreach :commands [run list (word "compile. first ?) ?]
queue "definition (list "nextline :number)
define :name :definition
make "linenumbers insert :number :linenumbers
end

to insert :num :list
if emptyp :list [output (list :num)]
if :num = first :list [output :list]
if :num < first :list [output fput :num :list]
output fput first :list (insert :num butfirst :list)
end

to eraseline :num
make "linenumbers remove :num :linenumbers
end

to immediate :line
if equalp :line [list] [foreach :linenumbers [print thing ?] stop]
if equalp :line [run] [run (list (word "basic% first :linenumbers))
                       stop]
if equalp :line [exit] [throw "toplevel]
print sentence [Invalid command:] :line
end

;; Compiling each BASIC command

to compile.end :command
queue "definition [stop]
end

to compile.goto :command
queue "definition (list (word "basic% last :command) "stop)
end
```

```
to compile.gosub :command
queue "definition (list (word "basic% last :command))
end

to compile.return :command
queue "definition [stop]
end

to compile.print :command
make "command butfirst :command
while [not emptyp :command] [c.print1]
queue "definition [print []]
end

to c.print1
make "exp expression
ifelse equalp first first :exp "" ~
       [make "sym gensym
        make word "%% :sym butfirst butlast first :exp
        queue "definition list "type word ":%% :sym] ~
       [queue "definition fput "type :exp]
if emptyp :command [stop]
make "delimiter pop "command
if equalp :delimiter ", [queue "definition [type char 9] stop]
if equalp :delimiter "\; [stop]
(throw "error [Comma or semicolon needed in print.])
end

to compile.input :command
make "command butfirst :command
if equalp first first :command "" ~
   [make "sym gensym
    make "prompt pop "command
    make word "%% :sym butfirst butlast :prompt
    queue "definition list "type word ":%% :sym]
while [not emptyp :command] [c.input1]
end

to c.input1
make "var pop "command
queue "definition (list "make (word ""% :var) "readvalue)
if emptyp :command [stop]
make "delimiter pop "command
if not equalp :delimiter ", (throw "error [Comma needed in input.])
end
```

```
to compile.let :command
make "command butfirst :command
make "var pop "command
make "delimiter pop "command
if not equalp :delimiter "= [(throw "error [Need = in let.])]
make "exp expression
queue "definition (sentence "make (word ""% :var) :exp)
end

to compile.for :command
make "command butfirst :command
make "var pop "command
make "delimiter pop "command
if not equalp :delimiter "= [(throw "error [Need = after for.])]
make "start expression
make "delimiter pop "command
if not equalp :delimiter "to [(throw "error [Need to after for.])]
make "end expression
queue "definition (sentence "make (word ""% :var) :start)
queue "definition (sentence "make (word ""let% :var) :end)
make "newname word "% gensym
queue "definition (sentence "make (word ""next% :var)
                                  (list (list :newname)))
queue "definition (list :newname)
define :name :definition
make "name :newname
make "definition [[]]
end

to compile.next :command
make "command butfirst :command
make "var pop "command
queue "definition (sentence "make (word ""% :var) (word ":% :var) [+ 1])
queue "definition (sentence [if not greaterp]
                            (word ":% :var) (word ":let% :var)
                            (list (list "run (word ":next% :var)
                                        "stop)))
end
```

```
to compile.if :command
make "command butfirst :command
make "exp expression
make "delimiter pop "command
if not equalp :delimiter "then [(throw "error [Need then after if.])]
queue "definition (sentence "if :exp (list c.if1))
end

to c.if1
local "definition
make "definition [[]]
run list (word "compile. first :command) :command
ifelse (count :definition) = 2 ~
       [output last :definition] ~
       [make "newname word "% gensym
        define :newname :definition
        output (list :newname)]
end

;; Compile an expression for LET, IF, PRINT, or FOR

to expression
make "expr []
make "token expr1
while [not emptyp :token] [queue "expr :token
                           make "token expr1]
output :expr
end

to expr1
if emptyp :command [output []]
make "token pop "command
if memberp :token [+ - * / = < > ( )] [output :token]
if memberp :token [, \; : then to] [push "command :token  output []]
if numberp :token [output :token]
if equalp first :token "" [output :token]
output word ":% :token
end
```

```
;; reading input

to basicread
output basicread1 readword [] "
end

to basicread1 :input :output :token
if emptyp :input [if not emptyp :token [push "output :token]
                  output reverse :output]
if equalp first :input "| | [if not emptyp :token [push "output :token]
                             output basicread1 (butfirst :input)
                                               :output "]
if equalp first :input "" [if not emptyp :token [push "output :token]
                             output breadstring butfirst :input
                                                 :output "]
if memberp first :input [+ - * / = < > ( ) , \; :] ~
   [if not emptyp :token [push "output :token]
    output basicread1 (butfirst :input) (fput first :input :output) "]
output basicread1 (butfirst :input) :output (word :token first :input)
end

to breadstring :input :output :string
if emptyp :input [(throw "error [String needs ending quote.])]
if equalp first :input "" ~
   [output basicread1 (butfirst :input)
                      (fput (word "" :string "") :output)
                      "]
output breadstring (butfirst :input) :output (word :string first :input)
end

to split :line
output fput first :line split1 (butfirst :line) [] []
end

to split1 :input :output :command
if emptyp :input [if not emptyp :command [push "output reverse :command]
                  output reverse :output]
if equalp first :input ": [if not emptyp :command
                              [push "output reverse :command]
                           output split1 (butfirst :input) :output []]
output split1 (butfirst :input) :output (fput first :input :command)
end
```

```
;; Runtime library

to nextline :num
make "target member :num :linenumbers
if not emptyp :target [make "target butfirst :target]
if not emptyp :target [run (list (word "basic% first :target))]
end

to readvalue
while [emptyp :readline] [make "readline basicread]
output pop "readline
end
```