ORCA/Integer BASIC™

An Integer BASIC Compiler,
with Source Code,
for the Apple® IIGS

Mike Westerfield

Byte Works, Inc.
4700 Irving Blvd. NW, Suite 207
Albuquerque, NM 87114
(505) 898-8183

Table of Contents

Chapter 1
Introducing Integer BASIC

Why Integer BASIC?

This program is a complete Integer BASIC compiler for the ORCA environment. It supports such advanced features as the source-level debugger, use of the linker so you can use libraries and call assembly language subroutines, and true 65816 native code generation. It's fair to ask, though, why we are releasing an Integer BASIC compiler at all. One is that you might actually want an Integer BASIC compiler. That's a long shot, I admit, but there may be a few folks who fit the bill. Another is that you will see how a compiler actually works. Personally, I think that is a very important thing for any serious programmer to know. Until you know how a compiler works, you will not know why compilers do some of the things they do. Without knowing how compilers work, you won't be able to get the most from the tools that are available to you. There are also a lot of different kinds of languages, each with its own strengths and weaknesses. Once you know how compilers work, you'll be more aware of the advantages and disadvantages of a compiled language as compared to assembly language, an interpreted language like AppleSoft, or a crossbreed, like Apple Pascal (which compiles to an interpreted language). Another reason for learning about compilers is that compilers use a number of very useful programming techniques that appear in many other programs. Adventure games, text editors, spelling checkers, spread sheets, and even operating systems all make use of the techniques you will see in a compiler. In fact, almost any program that processes text can be made better, smaller, faster, or written quicker if the ideas used in writing compilers are applied to the program. Finally, a compiler – even a small one – is a large program. This is one of the few chances you will get to study a complete program of this size. I always learn something new from reading and studying a large program written by another programmer, and I think you will learn some things from this program, too.

But why Integer BASIC? After all, a full BASIC compiler is sorely needed on the Apple IIGS, as well as FORTRAN, and possibly even Modula or Cobol. Why not use one of these other languages as a sample compiler?

First, none of the other languages are really appropriate for a sample compiler. A full compiler for a language like FORTRAN would be about 20,000 lines of code, with another 15,000 or so lines for the run-time library. Sorry, folks, but that's just too long to document in a short book intended to introduce compiler concepts. Even an Integer BASIC compiler is pretty long: this compiler is over 3,000 lines of Pascal, plus another 1,000 lines or so of assembly language for the run-time library. That's pushing it for a program presented in a book this size. So whatever language I picked, it had to be a small one.

Many compiler books will invent their own language. I've read a lot of those, and I always come away feeling a little cheated. The languages are often defined so they avoid the really interesting parts of a language. Real languages have warts – little inconsistencies and problems for the compiler writer that are not there in these little model languages. For that reason, I wanted a real language instead of something I invented on my own.

Finally, I didn't want you to have to learn a new language to read and understand this book. Nearly everyone who programs the Apple II knows BASIC, and there are still a lot of folks around who remember Integer BASIC. You might even find some real programs that you could move over to the GS.

A Quick History of Integer BASIC

Those of you who didn't have the pleasure of owning one of the original Apple][computers, even before the Apple][+, may never have heard of Integer BASIC. Back when the Apple][was introduced in 1976, it came with 12K worth of ROM. In this tiny ROM you could find such gems as a miniassembler, which is still the fastest assembler ever written for the Apple][; the F8 ROM, which was chocked full of miscellaneous subroutines; and an interpreter for Integer BASIC. Naturally, to squeeze all of this into such a tiny space, Integer BASIC couldn't have been all that big of a language, and it wasn't. On the other hand, Integer BASIC was used for years after Apple][computers came equipped with AppleSoft, for two reasons: first, Integer BASIC is faster than AppleSoft, since it does its work using two byte integer numbers while AppleSoft does all calculations using five byte floating-point numbers, and second, a lot of folks had programs written in Integer BASIC, and we all know how hard it is to get a programmer to port a working program to another language, even if it is a better language!

Integer BASIC gradually became more and more rare, though, and had pretty much died out by the mid 1980's. Still, you can find a large number of old Integer BASIC programs in the dusty back corners of Apple][public domain libraries. This package even comes with a conversion program, so you can move some of those old programs to the Apple IIGS.

What You Need and What You Get

This package comes with a complete, ready-to-use Integer BASIC compiler; a subroutine library for the compiler; source code for the compiler and the subroutine library; a set of programs used to test the compiler; and a conversion utility that can be used to convert old Integer BASIC programs to GS/OS SRC files, ready for the compiler to compile. You also get this manual and a product registration card. Be sure you send back the registration card – it secures your place in our database, so you will get notified of updates in this product, as well as get special offers on new products as they are released.

The disk is not copy protected, and we highly recommend that you make a copy of it and work only from the copy.

To use Integer BASIC, you must have one of the ORCA languages or some other development environment that is compatible with the ORCA languages. You will need the normal 1.25M of RAM (1.125 with ROM 03) and at least one 3.5" floppy drive. If you want to make changes to the compiler and libraries, you will need ORCA/Pascal to recompile the compiler, and ORCA/M to recompile the libraries. While it is possible to use Pascal, assembly language, and Integer BASIC from floppy disks to do all of this, life will be a lot easier of you have a hard disk, so we highly recommend one.

Installation

Before we actually start writing the compiler, it would be a real good idea to install the finished compiler and play around with it a bit to get familiar with the language.

You need to have some ORCA language installed to install Integer BASIC. I will assume that you already have a working ORCA system, and are reasonably familiar with it. Here's the steps you need to follow to install Integer BASIC:

1. Copy the file IBASIC into your LANGUAGES folder. You can find IBASIC in the LANGUAGES folder of your Integer BASIC program disk.
2. Copy the file BASICLIB into your libraries folder. BASICLIB is in the LIBRARIES folder of your Integer BASIC program disk. BASICLIB must come before the other libraries in the libraries folder; the easiest way to make sure it is is to move all of the libraries that are already in the LIBRARIES folder to some other folder, then copy all of the libraries back in, one at a time, in the proper order.
3. Edit the SYSCMND file in your ORCA SYSTEM folder, and add the lines

   ```
   IBASIC  *L      10      Integer BASIC Compiler
   CONVERT         *U              Integer BASIC Converter
   ```

   This defines IBASIC as a new language with a language number of 10, and tells the system about the convert utility.
4. Edit the SYSTABS file in your ORCA SYSTEM folder. Add these lines to the SYSTABS file in the appropriate spot. (You can tell where the "appropriate" spot is by looking at the format of the tab lines that are already there.)

   ```
   4
   10010
   00000000100000001000000010000000100000001000000010000000100000001000
   00001000000001000000010000000000000000100000001000000010000000100000000
   10000000100000001000000010000000100000001000000010000000100000001000
   0000100000000100000001000000010000000100000001000002
   ```

   While I can't show it well in this manual, this is really just three lines. The last group of four lines should be typed as a single line in the SYSTABS file. An easier way to get the line into the SYSTABS file is to copy one of the tab lines from another language, like Pascal (language 5) and paste it down into the correct spot.

Once these steps are performed, you should either reboot or use the COMMANDS shell command to reread the SYSCMNDs file. I usually reboot – it takes a little longer, but conserves brain power.

Using the Compiler

The next step is to write your first Integer BASIC program. From the text environment, start by typing

```
IBASIC
```

then edit a new file, HELLO.BAS. Typing the language name before you create the new file tells the editor to create an Integer BASIC program. Things are done a little differently from PRIZM, where you create a new window first, then select IBASIC from the languages menu to set the language, and save the window as HELLO.BAS. After these preliminary steps, type in this program:

```
100 print "Hello, world."
```

Compile and execute the program in the normal way.

You can find a number of sample programs on the disk with the compiler. These sample programs are actually used later to test the compiler; they also give examples of all of the statements in the Integer BASIC language.

Features of the Compiler

For the most part, the features of this compiler are pretty similar to the features of any ORCA compiler. The compiler generates object module format files (OMF), so it does use the linker. It supports the normal flags, like +w to wait when an error is flagged, +l to list the source program, and +d to generate debug code for use with PRIZM's source-level debugger. It has one additional flag, +s, that we will use later when we look at the code generator. The +s flag tells the compiler to list the machine language generated by the compiler as assembly language source code.

From the text shell, you can press a key to pause, or use ⌘. to abort a compile. If you specify the +t flag, the compiler will stop a compilation when an error occurs and enter the editor, placing the cursor on the offending line and displaying the error message at the bottom of the edit screen.

There is one major addition to the original Integer BASIC language. For those of you who have been around for a while, you will find the familiar split screen, with low resolution graphics at the top and four lines of text at the bottom of the display. You can't use this mechanism with PRIZM, though, so a new graphics mode has also been added. This compiler has full support for 320 mode Apple IIGS graphics.

With the exception of programs that use low resolution graphics or 40 column text, programs created by the compiler can be executed from PRIZM or the ORCA shell environment.

In other words, the only surprise is that all of the normal features of a commercial compiler are actually supported!

Chapter 2
The Integer BASIC Language

How to Use This Chapter

This chapter gives a technical description of the Integer BASIC language as implemented by this compiler. The chapter is laid out for easy reference, rather than as a teaching tool.

The statements, listed in alphabetical order, start with a one-line model that shows how the statement is written in a BASIC program. This model line is basically a simplified form of BNF, one of many notations used by compiler writers to specify the grammar of a language. As an example, the COLOR statement looks like this:

```
COLOR = expression
```

Reserved words are shown in capitol letters, and must be typed exactly as shown, with no intervening spaces. The word COLOR is an example of a reserved word. Special characters, like the = character in this statement, are also shown. Like reserved words, these special characters must appear in the program if they are shown in the model statement. While spaces will be used in the statement model lines, they are not actually required. Finally, portions of the statement which must be filled in by some other characters in the program are shown in lowercase, like "expression" in this example. Expressions are covered in detail later in this chapter, but could include numbers, like 4, or computed values, like LASTCOLOR/2+1.

Some statements have optional tokens, like the LET statement:

```
[ LET ] l-value = expression
```

The optional tokens are surrounded by square brackets, which are not typed in the program. This statement model says that you can write a LET statement with or without the first reserved word. For example, these two statements are equivalent:

```
100 LET A = I*4
100 A = I*4
```

Sometimes, the optional field can be repeated zero or more times, in which case the closing] character is followed by an asterisk, as in the NEXT statement.

```
NEXT identifier [ , identifier ]*
```

This shows that you can place a second, optional loop variable after the first one, separating the two with a comma. Since the asterisk is used, you are allowed to repeat this process, placing another comma and a third loop variable after the second, and so forth.

The last case is when there are more than one optional possibility, as in the PRINT statement:

```
PRINT [ , | ; | expression ]*
```

This statement model shows that the PRINT statement can be followed by any number of expressions, comma characters, or semicolon characters. The three possibilities are shown inside brackets, which means that they are all optional, and separated by | characters, which mean that any of the three options can be used.

There are often other rules associated with the statements that cannot be shown as part of the statement model. For example, while the statement model for the FOR loop shows you can put as many identifiers in the NEXT statement as you want to, the compiler does insist that each of the values appear before the NEXT statement on a FOR statement, and places some additional requirements on the order in which the variables appear.

The following table lists the various optional fields, like expression, which are used in the statement models.

field	use
expression	A numeric or string expression. Expressions are covered later in this chapter.
identifier	The name of a variable.
integer	An integer constant value, like 10 or 32000.
l-value	Some variable that can be assigned a value, like I, ARR(2), or STR$.
statement	Any one of the BASIC statements.
string	A string constant or string expression, like "Prompt: " or STR$.

Integer BASIC Programs

Tokens

BASIC programs are made up of a series of words and punctuation marks, much the same as a sentence written in English. These words and punctuation marks are called tokens by compiler writers. In this section, we'll look at the various tokens from the viewpoint of the BASIC programmer.

Identifiers are the first class of tokens in BASIC. These are the words used for variables names, like I and ARRAY. Integer basic identifiers must start with an alphabetic character; this character can be followed by up to 99 other alphabetic or numeric characters. The last character can be a $, in which case the identifier is used for a string value; all other identifiers refer to integer values. Unlike the original interpreted version of integer BASIC, this compiled version is case insensitive, so you can use uppercase or lowercase letters in identifiers. The case is not significant, so the variable Array is the same as the variable ARRAY. While the compiler is case insensitive, this manual will show BASIC programs in uppercase, since that seems to be traditional.

Here are some examples of legal identifiers:

```
i       Array   myName  NAME$   N3
```

Reserved words look a great deal like identifiers, but are used for special purposes by the BASIC language. Reserved words include words like PRINT, used to start a print statement. Reserved words can only be used where the BASIC language expects them. In particular, a reserved word can never be used as the name of a variable. You can, of course, use reserved words inside of a string, since the entire string is treated as a single token. Unlike the original interpreted version of Integer BASIC, you can also use a reserved word as a smaller part of a larger variable name. PrintIt, for example, is a perfectly legal variable name. Like identifiers, reserved words are case insensitive, but this manual shows them in uppercase.

Here's a list of the reserved words in Integer BASIC:

```
ABS     AND     ASC     AT      CALL    COLOR   DIM
END     FOR     GOSUB   GOTO    GR      HGR     HLIN
IF      INPUT   LEN     LET     MOD     NEXT    NOT
OR      PDL     PLOT    POP     PRINT   REM     RETURN
RND     SCRN    SGN     STEP    TAB     TEXT    TEXT80
THEN    TO      VLIN    VTAB
```

There are two kinds of constants in Integer BASIC, and each is a new kind of token. Integer constants are numbers; they are made up of a series of numeric characters. There is no limit to the number of characters that can make up an integer constant, since you can put as many leading zeros on the number as you like. No matter how many characters you use, though, the value of the integer must be in the range 0 to 32767. Negative numbers are allowed in Integer BASIC, but one of the little known facts about compilers is that there is no such thing as a negative constant. The compiler treats -45 as two tokens, not one – a minus sign, followed by the integer constant 45.

Here are some examples of legal integer constants:

```
0       4       32767   000003
```

The second kind of constant is the string constant. String constants are formed by placing any typeable character except a quote mark between two quote marks. In Integer BASIC, there is no way to place a quote mark inside of a string. String constants can contain zero to 255 characters.

Here are some examples of string constants:

```
"Input: "
"Hello, world."
```

The last kind of token is called a reserved symbol; these are the punctuation marks of BASIC. The table below lists the reserved symbols used by Integer BASIC. Some of these use more than one character, unlike punctuation marks in English.

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

Internally, the compiler also treats end of line characters and the end of the file as a token, but these are not tokens in the traditional sense and are not usually listed in reference manuals. Since this is a book about writing compilers, though, it seems appropriate to be complete.

Lines

An Integer BASIC program is made up of a series of lines; each of these lines corresponds to a line you type in the editor. Integer BASIC lines have the following format:

```
integer statement [ : statement ]
```

The integer at the start of the line is the old, familiar line number from interpreted BASICs like AppleSoft. Line numbers must appear on each line, and each new line number must be larger than the one that came before it. The compiler uses line numbers as destination points for GOTO and GOSUB statements.

More than one statement can appear on the same line; if so, they are separated with colon characters.

Expressions

Expressions come in two types; integer expressions and string expressions. Integer expressions are also used for two purposes: to compute a value, as you would do in a LET statement, or to decide if a condition is met, as in an IF statement.

Integer expressions are made up of one or more integer constants, integer variables, or functions, separated by the various operators.

From the perspective of the compiler writer, the easiest way to explain an expression is with a series of definitions, each of which contains other definitions. Since our purpose here is to learn a little about how compilers are put together, that's how we'll describe expressions.

At the top level, an expression is made up of a simpler kind of expression. This simpler expression can be followed by the OR operator and another of the simpler expressions. For reasons that will become clear in a moment, we'll call this simpler expression an andop expression, and write the definition of an expression like this:

```
expression ::= andop [ OR andop ]*
```

Ignoring for a moment exactly what an andop expression is, take it for granted that an integer constant satisfies the requirements. Then this definition says that the following are all legal expressions. The OR operator, incidentally, is generally called a boolean operator, since it returns a logical result. If either of the andop expressions is not zero, OR returns a value of 1 (true). If both of the andop expressions are zero, the result is also zero.

expression	resulting value
4	4
0 or 0	0
45 or 16	1
3 or 0	1
1 or 2 or 0	1

The andop expression looks a great deal like an expression. In fact, the andop expression is an expression that may contain an AND operator, which is where the cleverly chosen name comes from.

```
andop ::= cmpop [ AND cmpop ]*
```

The AND operator is another logical operator, and like the OR operator, it returns either 0 or 1. The AND operator returns a 1 if both of the surrounding cmpop expressions are non-zero (true), and returns 0 if either or both of these operators are zero.

At this point, we have enough of the description of the expression to start to see the power of this method of description. After all, this cumbersome syntax is more complicated than the way most books describe expressions, so there must be some point to all of this. There are two, actually: this method of describing expressions, known as BNF, tells you a great deal about the structure of an expression once you know how to read the statements. It also makes writing the part of the compiler called the parser almost trivial, as we'll see later in Chapter 3.

As you know, operators in most languages have different precedence, which means they are evaluated in a specific order, even if they appear in a different order in the line. For example, in most languages, 1+2*3 is 7, not 9, since multiplication has a higher precedence than addition. In Integer BASIC, the AND operator has a higher precedence than the OR operator. The BNF for the language shows this, in a backwards sort of way. When you look at the two statements we have so far,

```
expression ::= andop [ OR andop ]*
andop ::= cmpop [ AND cmpop ]*
```

you see that an OR operator has an andop expression on either side. This andop expression is evaluated completely, before the OR operation is performed – in other words, the AND operation is always done first, and has a higher precedence than the OR operator.

We can flesh out the description of the expression quickly, now, showing precedence and exactly how the expression is put together:

```
expression  ::= andop [ OR andop ]*
andop       ::= cmpop [ AND cmpop ]*
cmpop       ::= plusminus [ = | < | > | <= | >= | # | <>
                plusminus ]*
plusminus   ::= mulop [ + | -  mulop]*
mulop       ::= exponent [ * | / | MOD  exponent ]*
exponent    ::= term [ ^ term ]*
```

```
term        ::= NOT term
term        ::= - term
term        ::= + term
term        ::= integer
term        ::= string
term        ::= identifier
term        ::= identifier ( expression [ , expression ] )
term        ::= ( expression )
term        ::= ASC ( expression )
term        ::= LEN ( expression )
term        ::= ABS ( expression )
term        ::= SGN ( expression )
term        ::= PDL ( expression )
term        ::= RND ( expression )
term        ::= SCRN ( expression , expression )
```

Some of these statements are a lot more complicated than the simple AND and OR operations. For both plusminus and mulop, there is more than one operator that can be used; in this case, the operators have equal precedence. When operators have equal precedence, they are generally evaluated from left to right, although this actually varies from language to language, and sometimes from operator to operator within the same language. Integer BASIC does evaluate operations of equal precedence from left to right, though.

This seems trivial, but it is actually a very important point. After all, one common trick using integer math is to lop off part of a number like this:

```
RESULT = A/512*512
```

For you clever types, you know that this gives us the value of A with the least significant 9 bits cleared, so we've done a clever binary operation using integer math – or have we? An optimizing compiler, like ORCA/Pascal, which is not restricted to doing operations of equal precedence in any particular order is going to "help" you by converting this internally to

```
RESULT = A/262144
```

Even if you stuff the value 512 in to a variable and try again, a really clever optimizing compiler would do the same thing. (ORCA/Pascal is merely fairly clever, not really clever, so it wouldn't do this wonderful optimization.)

The most complicated of all of the parts of the expression, though, is the term, which is basically the part that comes between operators, and results in a number. To keep things as simple as possible, the description of term has been broken down into several lines, one for each major variation. The first three are easy enough: they implement the boolean NOT operator, which returns 0 of the value of the following term is non-zero, and 1 if the following term is 0; the negation operator, which returns the negative of a number; and the unary plus operator, which actually doesn't do anything. (Unary plus lets you do comforting things like I = +6, but of course, that means the same thing as I = 6, so the program the compiler creates is the same either way.) The next few variations are also fairly straight-forward, showing integer and string constants, simple variables, and subscripted variables. The subscripted variable with either one or two expressions shows one of the weaknesses of BNF, though. The single subscript is used for integer arrays, while the double subscript is used to select a substring from a string value.

The next form of the term is really where the fun starts, and it shows why all compilers are either recursive (as this one is) or use some form of stack to effectively do the same thing as recursion. A term can consist of another expression, enclosed in parenthesis – but a term is itself a part of an expression. Rather than getting tied up in the philosophy of recursion, just keep in mind that all this is really saying is that you can enclose part of an expression in parenthesis to make sure that part gets evaluated first. Another way of thinking about it is that, since the term is at the lowest level of the expression hierarchy, it is always evaluated first, so enclosing part of an expression in parenthesis gives that part a higher precedence than the surrounding parts. We put this to use all the time. For example, if you really want 1+2*3 to come out to be 9, you can use parenthesis, as in (1+2)*3, so the addition will be performed first.

The last kind of term is the function call. There are seven examples in Integer BASIC, all with parameters. You could create a function that didn't have a parameter, but there don't happen to be any examples in Integer BASIC. It just turned out that in Integer BASIC, all of the functions that were needed happened to have one or two parameters.

The table below lists the various operators and functions, telling you what they do.

operator	use
a OR b	Returns 0 if a and b are zero, and 1 otherwise.
a AND b	Returns 0 if a or b is zero, and 1 otherwise.
a = b	Returns 1 of a and b are the same value, and 0 if not.
a < b	Returns 1 if a is numerically less than b, and 0 if not.
a > b	Returns 1 if a is numerically greater than b, and 0 if not.
a <= b	Returns 1 if a is numerically less than or equal to b, and 0 if not.
a >= b	Returns 1 if a is numerically greater than or equal to b, and 0 if not.
a # b	Returns 0 of a and b are the same value, and 1 if not.
a <> b	Same as #.
a + b	Returns the sum of a and b.
a - b	Returns a minus b.
a * b	Returns the product of a and b.
a / b	Returns the integer part of a divided by b.
a MOD b	Returns the integer remainder of a divided by b.
a ^ b	Returns a raised to the power b.
- a	Returns 0 - a.
+ a	Returns 0 + a.
NOT a	Returns 1 if a is 0, and 0 if not.
ASC(a)	Returns the numeric ASCII value associated with the first character in the string a.
LEN(a)	Returns the number of characters in the string a.
ABS(a)	Returns the value of a if a is positive, or - a if a is negative.
SGN(a)	Returns 0 if a is zero, -1 if a is less than zero, and 1 if a is greater than zero.
PDL(a)	Returns information about the mouse. PDL(0) returns the horizontal (X) coordinate, which ranges from 0 to 319. PDL(1) returns the vertical (Y) coordinate, which ranges from 0 at the top of the screen to 199 at the bottom. PDL(2) returns 1 if the button is down, and 0 if not. Any other argument returns the mouse status word.
RND(a)	Returns a pseudo-random number in the range 0 to a-1.
SCRN(a,b)	Returns the color of the screen pixel at a,b, where a is the horizontal coordinate and b is the vertical coordinate.

Graphics

The original Integer BASIC interpreter on the Apple][used low resolution graphics. For those of you fortunate enough to have missed that graphics mode, it consisted of a 40 block wide by 48 block high graphics screen, where each block could be any of 16 different colors. The memory for the graphics screen occupied the same space as the 40 column by 24 line text display, and with a judicious combination of pokes, you could switch between text, graphics, or even a mixed mode with 4 lines of text at the bottom and 40 lines of blocks at the top. You can get the mixed text and graphics mode using the GR statement.

As nostalgic as I sometimes am, the 320 by 200 pixel graphics screen of the Apple IIGS is so much nicer than the original low resolution graphics screen that this compiled Integer BASIC also supports the new 320 mode graphics, rather than the original low resolution graphics. The graphics commands, however, have not changed. You can use any of the graphics commands in either graphics mode.

Integer BASIC Statements

CALL identifier

The CALL statement does a JSL (the assembly language Jump to Subroutine Long instruction) to the named label. This lets you create assembly language subroutines that can be called from BASIC. The assembly language subroutines can access any of BASICs variables. Here's a sample, showing a BASIC program calling an assembly language subroutine that uses QuickDraw to draw a rectangle.

```
100 HGR
110 COLOR = 7
120 H1 = 10
130 H2 = 100
140 V1 = 10
150 V2 = 100
160 CALL DRAWRECT
170 FOR i = 1 to 30000: NEXT I

DrawRect start

         lda    h1                      get the rectangle size from
         sta    r+2                      BASIC's variables
         lda    h2
         sta    r+6
         lda    v1
         sta    r
         lda    v2
         sta    r+4
         ph4    #r                      draw the rectangle
         _DrawRect
         rtl                            return to BASIC

r        ds     8                       rectangle
         end
```

COLOR = expression

The COLOR statement changes the color that will be used the next time you use one of the graphics commands to draw to the graphics screen. Color numbers range from 0 to 15; if the value of the expression is outside of that range, it is reduced to the range 0 to 15 using a mod operation.

Here are the various pen colors, listed by color number. The colors are different for low resolution and high resolution graphics.

Number	HGR Color	GR Color
0	black	black
1	dark gray	magenta
2	brown	dark blue
3	purple	purple
4	blue	dark green
5	dark green	gray
6	orange	medium blue
7	red	light blue
8	beige	brown
9	yellow	orange
10	green	gray
11	light blue	pink
12	lilac	green
13	periwinkle blue	yellow
14	light gray	aqua
15	white	white

Here's an example of the COLOR statement, used in a short program that draws a small flag.

```
100 HGR
110 STRIPE = 7
120 FOR I = 10 TO 16
130    COLOR = 4
140    HLIN 10, 18 AT I
150    COLOR = STRIPE
160    HLIN 18, 30 AT I
170    GOSUB 500
180 NEXT I
190 FOR I = 17 to 22
200    COLOR = STRIPE
210    HLIN 10, 30 AT I
220    GOSUB 500
230 NEXT I
240 REM Pause for a moment
250 FOR I = 1 to 30000: FOR J = 1 TO 10: NEXT J: NEXT I
260 END
500 REM Change the color
510 IF STRIPE = 7 THEN 540
520 STRIPE = 7
530 RETURN
540 STRIPE = 15
550 RETURN
```

DIM identifier (integer) [, identifier (integer)]*

The DIM statement is used to create strings and arrays of integers.

In Integer BASIC, integer arrays always have a single subscript. In this compiled version of Integer BASIC, these arrays are always of a fixed size, and the subscript must be specified as an integer constant in the range 1 to 32767. The subscripts are numbered from 0 to the number given, so there is one more element in the array than the subscript given. For example,

```
100 DIM A(10),B(20)
```

declares two arrays of integer, the first with eleven elements numbered from 0 to 10, and the second with 21 elements numbered from 0 to 20.

If an array is used in the program before it is declared in a DIM statement, the array is declared by the compiler with eleven subscripts, numbered 0 to 10. It is illegal to dimension an array twice, or to dimension an array after the compiler has created this default range.

Strings are arrays of characters, implemented internally with a null terminator to mark the end of the string, although this is not something that is apparent when using the compiler. Strings with a null terminator are often referred to as c-strings. Strings can be declared with a subscript from 1 to 255, and take up one more byte in memory than the subscript size. The subscript size becomes the maximum number of characters the string can hold. The compiler does not double check string sizes when you assign a value to a string, so it is up to the programmer to make sure that the characters assigned to a string will fit.

When you use an integer array, whether you are using it as an l-value or as a term in an expression, you select an element from the array by following the array with an expression in parenthesis. For example, here's a short section of code to double the size of a three element vector – just the sort of thing you might do in a graphics program:

```
100 FOR I = 1 TO 3
110    A[I] = A[I]*2
120 NEXT I
```

You can use subscripts on string names, too. Integer BASIC more or less lets you treat a string as an array of characters. We'll use a few quick examples to see how this works. First, you can assign strings. Here's how you would assign a string constant to a variable:

```
100 A$ = "This is a test."
```

In this case, we've placed the string constant in the variable, replacing any characters that were already there. Using a subscript, we can tack another string onto the end of this one, or replace a part of the string with a new string.

```
110 A$(9) = "not a test."
```

After this statement executes, A$ contains the string "This is not a test."

Subscripts can also be used to pull a piece out of a string, like this:

```
120 B$ = A$(6,7)
```

This takes two characters from the string A$, assigning them to B$. At this point, B$ contains the word "is".

END

The END statement stops execution of the program. While you can put an END statement in your program as the last statement in the program, this is not required. That's different from the original interpreted Integer BASIC, which did require the END statement. The END statement is generally used to stop the program based on some condition, or just before the start of the subroutines in a program. For one example of the end statement, check out the flag program listed for the COLOR statement.

FOR identifier = expression TO expression [STEP [-] integer]

The FOR statement, together with the NEXT statement, is used for looping. Two values are used, one specifying the starting value, and the other the ending value. Either of these can be expressions.

For example, the following program uses a loop to write the numbers from 1 to 10.

```
100 FOR I = 1 TO 10
110    PRINT I
120 NEXT I
```

An optional step size can be used to change the loop so it counts by some number other than 1. This step size must be specified as a constant, possibly with a leading minus sign. For example, this loop counts to 100 by tens.

```
100 FOR I = 10 TO 100 STEP 10
110    PRINT I
120 NEXT I
```

The step size can also be used to count down, rather than up.

```
100 FOR I = 10 TO 1 STEP -1
110    PRINT I
120 NEXT I
```

The body of the FOR loop is always executed at least one time, even if the loop's stop condition is met right away. For example, the program

```
100 FOR I = 10 TO 1
110    PRINT I
120 NEXT I
```

does print the value 10.

The NEXT statement is always used to end a FOR statement, and you must code the same variable on the NEXT statement that was used in the FOR statement. For you AppleSoft buffs, that's a difference: in AppleSoft, you can leave off the loop variable on the NEXT statement. The NEXT statement must appear after the FOR statement, and there can be no open FOR statements before the NEXT statement is found. For example, this program is illegal, since the NEXT statement for the loop variable I appears after another FOR statement has been started, but before it has been finished.

```
100 FOR I = 1 TO 10
110 FOR J = 1 TO 10
120 NEXT I: REM This is not legal!  The J for loop must be closed first.
130 NEXT J
```

For loops may be nested up to 16 levels deep. When FOR loops are nested, the NEXT statements can be separate, or a single NEXT statement can be used, separating the loop variables with a comma. In either case, though, the loop variable for the innermost for loop must be specified first.

```
100 GR
110 COLOR = 4
120 FOR I = 1 TO 10
130    FOR J = 10 TO 1 STEP -1
140       PLOT I,J
150    NEXT J
160 NEXT I
```

```
100 GR
110 COLOR = 4
120 FOR I = 1 TO 10
130    FOR J = 10 TO 1 STEP -1
140       PLOT I,J
150 NEXT J, I
```

GR

The GR starts the old low resolution graphics mode with a 40 by 40 graphics screen at the top of the display, and four lines of 40 column text at the bottom of the display. This is generally used when you are porting old Integer BASIC programs to the Apple IIGS, since this version of Integer BASIC supports super high resolution graphics using the HGR command. Either GR or HGR must be used at least one time before any of the other graphics commands are used. You should not use the GR command from PRIZM, which needs to use the standard super high resolution graphics mode to function.

To switch to 40 column text mode you can use the TEXT statement. You can also switch back to 80 column text output using TEXT80, or even switch between low resolution graphics and super high resolution graphics by alternating GR and HGR commands. If you happen to exit the program while in graphics mode, the program will clean up after itself, essentially going a TEXT80 call.

```
100 GR
110 HLIN 0, 39 AT 20
120 VLIN 0, 39 AT 20
130 FOR I = 1 to 30000: NEXT I
```

HGR

The HGR statement starts QuickDraw II, the Apple IIGS graphics tool, in 320 graphics mode with the standard graphics palette. The screen will be white, and the graphics drawing mode will be COPY. The HGR statement or GR statement must be used at least one time before any of the other graphics commands are used.

To switch back to the text screen, you can use the TEXT80 statement. Once the TEXT80 statement has been used, you can switch back to the graphics screen without clearing the screen by using another HGR statement; anything that was drawn while the graphics screen was invisible will be on the screen when you switch back. For example, the following program draws a large + on the graphics screen while the text screen is visible, then switches back to the graphics screen to show it.

```
100 HGR
110 TEXT80
120 HLIN 0, 320 AT 100
130 VLIN 0, 200 AT 160
140 HGR
150 FOR I = 1 to 30000: NEXT I
```

If you happen to exit the program while in graphics mode, the program will clean up after itself, essentially going a TEXT80 call.

HLIN expression , expression AT expression

The HLIN statement draws a horizontal line on the graphics screen. The first two values give the left and right coordinates for the line, while the last value specifies the vertical position of the line. This statement can be used with coordinates that do not fall on the graphics screen, in which case the portion of the line that is off of the screen will not show up.

For example, the following group of HLIN and VLIN statements draw a rectangle that is about twice as wide as it is high.

```
100 HGR
110 HLIN 10, 210 AT 10
120 VLIN 10, 110 AT 210
130 HLIN 10, 210 AT 110
140 VLIN 10, 110 AT 10
```

You must use the GR statement or HGR statement to start one of the graphics modes before using HLIN.

IF expression THEN integer

IF expression THEN statement

There are two kinds of IF statements. In both cases, the expression is evaluated, and the result is treated as true if it is non-zero, and false if it is zero. If the expression is false, the statement after the IF statement is the next one executed. For the first form of the IF statement, if the expression is true, a GOTO is performed to the statement number given as the integer value. For the second form of the IF statement, if the expression is true, the statement after the THEN is executed.

```
100 FOR I = 1 TO 100
110    PRIME = 1
120    FOR J = 2 TO I-1
130       IF (I/J*J) != I THEN 160
140          PRIME = 0
150          GOTO 180
160    NEXT J
170    IF PRIME THEN PRINT I
180 NEXT I
```

The next statement can be a statement on the same line as the IF! This is a little different from AppleSoft. For example, try this program:

```
100 IF 0 THEN GOTO 110: PRINT "Test me"
110 END
```

The expression is 0, which is false, so the GOTO 100 is not executed. Integer BASIC trudges right along, executing the PRINT statement.

INPUT [string ,] l-value [, l-value]*

The INPUT statement reads values from the keyboard. An optional string may be used as a prompt. The INPUT statement can read either numbers or strings, basing the type of the read on the type of variable listed as the input variable. More than one value can be read with a single INPUT statement; if the user does not enter enough values, or if an incorrect type of value is entered, the INPUT statement responds by printing a ? prompt. The ? prompt is also used if you don't code a prompt in the program.

```
100 DIM NAME$(255)
110 INPUT "What is your name? ", NAME$
120 INPUT "How old are you? ", AGE
130 PRINT "So, "; NAME$; " you will be "; AGE+1; " next year."
```

[LET] l-value = expression

The expression to the right of the equal sign is evaluated, and the result is assigned to the l-value to the left of the equal sign.

Rules for evaluating expressions were given earlier in this chapter.

The l-value to the left of the equal sign must be some variable that can have a value assigned. This could be the name of an integer variable, as in

```
100 LET I = 4
```

You can also assign a value to an element of an array, like this:

```
100 A(I+1) = A(I)
```

Finally, you can assign strings, like this:

```
100 S$ = "This is a string"
```

If the l-value is a string, the result of the expression must also be a string; if the l-value is an integer, the result of the expression must be an integer.

By using array subscripts, you can use the LET statement to select a substring from a longer string, or to add one string to another string, starting from any position in the string. These topics are discussed in the section describing the DIM statement.

NEXT identifier [, identifier]*

The NEXT statement is used to close a for loop. For a description, refer to the description of the FOR statement.

GOSUB integer

The GOSUB statement calls the subroutine that starts on the line specified by the integer. When a RETURN is found, control returns to the statement following the GOSUB statement. Here's a short example of a subroutine call using the GOSUB statement:

```
100 PRINT "I'm going to call a subroutine..."
110 GOSUB 200
120 PRINT "I did it!"
130 END
200 PRINT "I'm doing it..."
210 RETURN
```

This program prints the following to the text screen:

```
I'm going to call a subroutine...
I'm doing it...
I did it!
```

Subroutines can call other subroutines, and if you are careful, they can even call themselves. The only restrictions are that there must be exactly one RETURN statement executed for each GOSUB statement that is executed, and while there is no limit on the total number of GOSUB statements, you do have to limit the nesting level. Unlike the original Integer BASIC, which had a maximum nesting level of about 100 calls, though, ORCA/Integer BASIC has a maximum nesting level of about 4000 calls.

See the POP statement for a way to break that first rule.

GOTO integer

The GOTO statement is used to jump to another location in the program. The next statement executed is the one with the line number specified by the GOTO statement.

Here's an example of the GOTO statement, used with an IF statement to create what would be a repeat loop in Pascal, or a while statement in C.

```
100 IF I > 100 GOTO 130
110 REM Place the statements in the loop here
120 GOTO 100
130 REM This is the end of the loop
```

POP

The POP statement removes one GOSUB return address from the stack. This is sort of like doing the RETURN without going back.

Here's an example of the POP statement:

```
100 GOSUB 200
110 END
200 GOSUB 300
210 PRINT "I got back."
220 RETURN
300 POP
310 RETURN
```

Following this example through, the main program calls a subroutine at line 200, which immediately calls a subroutine at line 300. At this point, there are two return addresses available, so we could do two RETURN statements. Instead, on line 300, the POP statement removes the return location from the most recent subroutine call – the one made on line 200. The RETURN statement on line 310 returns to line 110 and the program stops, printing nothing.

PLOT expression , expression

The PLOT statement draws a single point on the graphics screen. The first value is the X coordinate (horizontal position), while the second value is the Y coordinate (vertical position). Unlike mathematical coordinates, though, the Y coordinate starts at zero at the top of the screen, and increases toward the bottom of the screen.

The super high resolution graphics screen extends from 0 to 319 horizontally, and from 0 to 199 vertically. The low resolution graphics screen extends from 0 to 39 in both directions. If you specify a point that lies off of the screen, nothing happens – it isn't an error, but nothing will appear on the screen, either.

For example, the following program uses a line drawing algorithm which isn't limited to horizontal and vertical lines, like HLIN and VLIN. The program makes hundreds of calls to the line drawing subroutine, alternating colors to create a rather pretty picture.

```
10 HGR
20 X1 = 160: Y1 = 100: LINECOLOR = 9: COLOR = LINECOLOR
30 Y2 = 0: FOR X2 = 0 TO 320: GOSUB 100: GOSUB 400: NEXT X2
40 X2 = 320: FOR Y2 = 0 TO 200: GOSUB 100: GOSUB 400: NEXT Y2
50 Y2 = 200: FOR X2 = 320 TO 0 STEP -1: GOSUB 100: GOSUB 400: NEXT X2
60 X2 = 0: FOR Y2 = 200 TO 0 STEP -1: GOSUB 100: GOSUB 400: NEXT Y2
70 CALL KEYBOARD: IF KEY < 128 THEN 70: CALL CLEARSTROBE
80 END
100 REM
110 REM Line drawing subroutine.
120 REM
130 REM Adapted from "Create Your Own Games Computers Play,"
140 REM Keith Reid-Green, Digital Press, 1984
150 REM
160 X = X1
170 Y = Y1
180 PLOT X,Y
190 F = 0
200 F1 = F + ABS(X1-X2)
210 F2 = F - ABS(Y1-Y2)
220 IF ABS(F1) < ABS(F2) THEN 260
230    F = F2
240    X = X + SGN(X2-X1)
250 GOTO 280
260    F = F1
270    Y = Y + SGN(Y2-Y1)
280 PLOT X,Y
290 IF (X # X2) OR (Y # Y2) THEN 200
300 RETURN
```

```
400 REM
410 REM Swap colors
420 REM
430 IF LINECOLOR = 9 THEN 460
440    LINECOLOR = 9
450 GOTO 470
460    LINECOLOR = 13
470 COLOR = LINECOLOR
480 RETURN
```

You must use the GR statement or HGR statement to start one of the graphics modes before using PLOT.

PRINT [, | ; | expression]*

The PRINT statement writes strings or integers to the console. Each expression can be a string constant, an integer expression, or a string expression. In the case of the string constant, the string value is printed to the screen exactly as it is typed into the program. String expressions are first evaluated (the result of using a string variable is just the value of the string variable), then the resulting string is written to the screen. Integer expressions are evaluated and written to the screen as a number with no leading zeros or spaces.

Comma and semicolon characters can be intermixed with the expressions, with each expression separated by one or more of these characters in any order. These characters can also appear before the first expression and after the last expression. The comma character serves as a tab, writing enough spaces to get to the next screen column that is evenly divisible by 15, counting from 1. This gives a quick and easy way to write columns of numbers, as in this example.

```
100 PRINT "Number","Square"
110 FOR I = 1 TO 10
120    PRINT I, I*I
130 NEXT I
```

The semicolon character serves as a place holder, generally doing nothing. The one exception is when the semicolon is the last character on the PRINT statement, when it blocks the carriage return normally written after a PRINT statement writes the values of the expressions. A comma at the end of a PRINT statement also suppresses the carriage return, but it writes some spaces first.

```
100 PRINT "These strings ";
110 PRINT "appear on the same line."
```

A PRINT statement can also be used with nothing after it, in which case a blank line is written to the screen, or a line started with a PRINT statement that ended with a ; character is finished.

REM anything

The REM statement is used to place comments in your program. The compiler ignores any characters that appear after a REM statement, up to the end of the line or a : character, whichever comes first.

Unlike interpreted BASIC, a REM statement in a compiled BASIC does not take up any room in the finished program, so there is no penalty for doing a good job of commenting your program.

```
100 REM This program is legal, but since it contains only comments,
110 REM it doesn't do anything.
```

RETURN

Returns from the previous GOSUB statement. The statement that will be executed after this one is the statement that follows the most recently used GOSUB statement. If a RETURN statement is executed when no GOSUB statement has been executed, many things are possible, all of them bad. The most common result would be for your program to crash or hang.

See the description of GOSUB for an example.

TAB expression

The TAB command is used to change where text will be printed by the PRINT and INPUT statements. The TAB statement moves the cursor to the location specified without changing the current line. (You can use VTAB to change the line.) Columns are normally numbered from 0 to 79, although you only get 40 columns in low resolution graphics mode or 40 column text mode. If the value of the expression is less than zero, zero is used; if it is greater than the width of the screen, the width of the screen is used.

There's an example of the TAB statement being used to format a table of squares and cubes.

```
100 FOR X = 1 TO 10
110    PRINT X;
120    TAB 15
130    PRINT X*X
140    TAB 30
150    PRINT X*X*X
160 NEXT X
```

TEXT

The TEXT statement switches to a 40 column text display. You can use this display when porting old Integer BASIC programs that were formatted for a 40 column display, or to switch between a full text display and the mixed text and graphics display created by the GR command. Do not use 40 column text from PRIZM – the two environments are not compatible.

TEXT80

The TEXT80 statement switches back to the normal 80 column text screen from and other display mode.

VLIN expression , expression AT expression

The VLIN statement draws a vertical line on the graphics screen. The first two values give the top and bottom coordinates for the line, while the last value specifies the horizontal position of the line. The super high resolution graphics screen extends from 0 to 319 horizontally, and from 0 to 199 vertically, while the low resolution graphics screen goes from 0 to 39 in each direction, but this statement can be used with coordinates that do not fall on the graphics screen, in which case the portion of the line that is off of the screen will not show up.

For example, the following group of HLIN and VLIN statements draw a rectangle 10 pixels from the edge of the graphics screen, framing the screen.

```
100 HGR
110 HLIN 10, 310 AT 10
120 VLIN 10, 190 AT 310
130 HLIN 10, 310 AT 190
140 VLIN 10, 190 AT 10
```

You must use the GR statement or HGR statement to start one of the graphics modes before using VLIN.

VTAB expression

.ib.VTAB statement;

The VTAB command is used to change the line on which text will be printed by the PRINT and INPUT statements. (You can use TAB to change the column where text will be printed.) Lines are numbered from 0 to 23. If the value of the expression is less than zero, zero is used; if it is greater than 23, 23 is used.

There's an example of the VTAB statement being used along with the TAB statement to print a large X on the screen:

```
100 FOR X = 1 TO 10
120    VTAB X
130    TAB X
140    PRINT "@";
150    TAB 11-X
160    PRINT "@";
170 NEXT X
180 PRINT
```

Chapter 3
How a Compiler is Constructed

Overview

Compilers are Simple

One of the interesting things about compilers is that they really aren't very complicated. Over the space of several decades, some very smart people have worked on the problem of writing a compiler, and the result is several techniques that break a compiler up into a relatively simple, step-by-step approach. The basics of writing a compiler can easily be covered in a short book like this one.

"Sure," you mumble. I heard that. Yes, I know undergraduate computer science students often spend one to two grueling semesters in compiler writing courses. Yes, I know that the good books on compiler writing are hundreds of pages thick. I didn't say writing a compiler was easy, I said it was simple. Walking from Los Angeles to Chicago is simple: you just put one foot in front of the other for a long time, and eventually, with the aid of a map, several pairs of shoes, and some food, you get there. It's simple – but not easy.

You see, the reason compilers are hard programs, and the reason they take so long, is that they are very *long* programs, not because the individual steps take a world-class mind to figure out. Writing a large program – any large program – is a skill that takes a little talent and a lot of hard work to perfect. A compiler is a large program; a typical commercial-quality compiler can involve tens of thousands of lines of code, and that doesn't even count the development environment that supports the compiler. And, as you add nice little features like optimization, good run-time libraries, and complete support for an established standard, you can easily double the size of the compiler compared to a 95% implementation that cuts some corners.

All of that sounds real impressive, but it generally takes an example to bring a point like this home, so let's look at some real compilers. ORCA/Pascal is a complete implementation of the ISO Pascal standard (which is, for all practical purposes, the same thing as the ANSI Pascal standard), along with several extensions popularized by various non-standard compilers and a few that are needed to deal with the Apple IIGS. It is written in a mix of Pascal and assembly language. The compiler makes use of an extensive run-time library, all coded in assembly language. No compiler is complete without a test suite to make sure it works reasonably well; this test suite was purchased from a company that develops Pascal test suits. Of course, we added some new tests to check out extensions we added to Pascal. The compiler itself consists of 14707 lines of Pascal and 6121 of assembly language. The run-time libraries are 13892 lines long. The test suite that ORCA/Pascal must pass before you see it consists of 813 different Pascal programs. The source code archive for the Pascal compiler, libraries, and test suite spans 4 800K floppy disks – and remember, that's just the source code. ORCA/C is also written in Pascal and assembly language. It consists of 25642 lines of Pascal, and 2438 lines of assembly language, with 22193 lines of assembly language in the run-time libraries, and 656 programs in the test suite. There is some overlap in the libraries of the two compilers, but taken together, the libraries for the two compilers are still 28692 lines of code.

No individual piece of either of these compilers is particularly hard to understand, but the sheer size of a commercial-quality compiler does make it a hard program to write. By the end of this series of articles, you will have a good feel for the basic ideas used in all compilers, but we will use a pretty small program to explore these ideas. Our Integer BASIC compiler is a mere toddler of a program, coming in at 3537 lines of Pascal for the compiler itself, 2099 lines of assembly language for the run-time libraries, and a scant 42 programs in the test suite. A very small program, I'm sure you will agree – at least, small by comparison to a complete compiler for an industrial strength language.

Parts of a Compiler

As anyone who understands structured programming knows very well, the way to make a big problem manageable is to break the big problem up into a series of smaller problems. That, in essence, is the secret to understanding how modern compilers are constructed.

A compiler project consists of three major components: the compiler itself, the run-time library, and the test suite. The compiler is the part most people think about, and it is the one we will spend the most time on. Later, after you understand more about the compiler itself, we will come back to the other aspects of a compiler.

Inside the compiler, we divide things up even more. To see how, and to get a first glimpse at why, we'll follow a short program through the compilation process. Here's the program:

```
100 PRINT "Hello, world."
```

When you read this program, this book, or anything else, you don't really worry about the individual letters that make up a word. Instead, you concentrate on forming sentences, ideas, and finally understanding from the symbols you see. Compilers do the same thing: the controlling part of a compiler is called the parser, and it is responsible for forming "sentences" from various "words" that make up a program. In compiler parlance, these "words" are called tokens. Of course, this implies that the parser somehow knows what tokens are – and that's where the scanner comes in. The scanner reads your program one character at a time and returns tokens to the parser. The scanner is sometimes called a lexical analyzer by people who like to try and impress others with big words, or a lexer by people who like to sound like they are "in the know;" the terms all mean the same thing. In our sample program, the scanner would return five tokens: an integer value of 100, the reserved word PRINT, the string constant "Hello, world.", an end of line marker, and an end of file marker. The last two are not words in the traditional sense, but to understand a program, the compiler obviously needs to know when a line ends and when the program is finished; the scanner passes this information on in the form of two more tokens.

As the parser forms sentences from the words in the program, it uses the structure of the program itself to tell the semantic analyzer what to do. As the compiler sees the words returned by the scanner in our sample program, it tells the semantic analyzer that it has found a line number of 100. The semantic analyzer remembers this fact, but doesn't do anything with it just yet. Next, the parser sees the word PRINT, and tells the semantic analyzer that it will have to process a print statement: a few variables are set up, and the parser continues. Next, the parser sees a string, and passes this on to the semantic analyzer. This time, the semantic analyzer calls the fourth part of the compiler, the code generator, and tells it to create the machine language instructions necessary to print the string to the screen. Finally, the parser sees an end-of-line token, and tells the semantic analyzer that the statement is complete. When the parser sees the end-of-file token, the compiler stops.

To summarize, the compiler has four major stages: scanning, parsing, semantic analysis, and code generation. Scanning is the process of collecting individual characters and forming tokens. Parsing is the process of collecting tokens and deciding how (or if!) they fit together to form statements in a program. Semantic analysis is the process of deciding what the statements mean, and code generation is the process of creating a machine-language program that caries out that meaning.

Actually, there is a fifth part of the compiler, too, but it is hidden below the surface a bit. Compilers also use well-formed symbol tables to remember things about the program being compiled, like what variables exist, how large arrays are, and so forth. This isn't a separate task, though, so we will deal with the symbol table as the need for it comes up. Some compiler books will talk about the symbol table as a separate part of the compiler, like the scanner and parser, while others treat the symbol table as the glue that holds the other parts together. It really doesn't matter which way you think about the symbol table; I sort of like the last method, so that's how we'll treat it here.

Choosing a Language

One of the things that people often don't think about is that a compiler is really just a translator. Technically, an assembler is also a compiler: it translates assembly language, which is easy for us to read, into machine language, which is pretty tough to deal with. There are at least three languages involved in any compiler project: the source language, the target language, and the language the compiler itself is written in.

The source language for our compiler is Integer BASIC. The source language for a Pascal compiler is Pascal, while the source language for a German to English translator is German.

The destination language for our sample compiler is machine language – which is the classic, but not the only possible, destination language. The destination language for Apple Pascal is an interpreted language called p-code. The destination language for a C++ pre-compiler is C. The destination language for a German to English translator is English.

Finally, compilers are programs, and programs are written in some language. Before we move on to the actual compiler, I want to take a little time to tell you why I picked Pascal to write the compiler.

First, with the exception of a few special-purpose interpreters, absolutely any computer language I have ever encountered could be used to write a compiler. I have heard of compilers written in Pascal, C, assembly language, BASIC, LISP, Ada, FORTRAN, and Algol, and I am sure that other languages have been used that I haven't personally heard of. To be a good choice for writing a compiler, though, the language you pick has to fit a few requirements. First, it must be available. That seems obvious, but this simple requirement eliminates more languages than any other, since there are only four languages with good implementations on the Apple IIGS: Assembly language, BASIC, C and Pascal. This cuts out both Ada and Algol, either of which is a great language for implementing a compiler. The next requirement is that the language must be able to read text files and write either object files that are later linked, or executable files. All of the remaining languages satisfy this requirement. As you will see later, to handle symbol tables and parsing effectively, the language needs to be able to deal with dynamically allocated memory and recursion gracefully. This eliminates BASIC as an acceptable choice, and hinders assembly language quite a bit – you can handle dynamically allocated memory and recursion in assembly language, but very few people would consider it easy. Finally, a compiler is a large program. To finish one in the span of a year or so, you need a language that you can write quickly in, which eliminates assembly language. This is also the reason for choosing Pascal over C: Pascal is a language that puts restraints on the way you program so it can check your work for stupid mistakes, and you will make a lot of them in the process of churning out 30,000 lines or so of code. C is a language that assumes you know exactly what you are doing, so it stays out of the way. If your program crashes – well, you should have known better. We don't need the minimal added flexibility of C for a compiler, and we certainly need all the help we can get in writing a 30,000 line program, so Pascal is the only rational choice. If Ada or Algol were available, another choice might be possible, but of the languages that we actually have on the Apple IIGS, Pascal is clearly superior to any of the other contenders.

The Scanner

Tokens

The first step in writing a scanner is to decide what the various tokens are for the source language. These tokens can be divided up into categories according to how they are recognized and what information we need to remember about the tokens. The various tokens are known as reserved words, identifiers, integer constants, strings and reserved symbols. We'll look at each of these classes of tokens one at a time.

As we look at these various topics, I want to show you how they are handled in the compiler itself. You will find various pieces of the compiler interspersed throughout the text so you can see firm examples of what is being discussed. The complete source for the compiler is on disk; you can print it out to follow along, or even load it into your computer as you read.

It is the parser that is in control of the overall process of compiling a program, so it is the parser that decides when it needs a new token. When the parser is ready for a new token, it calls NextToken, which is the top-level subroutine in the scanner. NextToken calls NextCh, also in the scanner, to get characters from the file. NextCh hides all of the ugly little details of opening a file, reading from the file, and printing lines when they are finished. This is a simple, straight-forward task, but it helps the design of the scanner to hide these details in NextCh.

Getting back to NextToken, it starts by skipping spaces and illegal characters, then it looks at the character it has found, and uses a big case statement to branch to the section of code that knows how to build a particular type of token. As it turns out, we can decide what type of token we are compiling right away, just from looking at the first character that we find. This is no accident, of course – it makes the compiler a lot smaller and faster that it would be if this were not true, so people who design computer languages design this idea right into the language.

With all of the workhorse routines removed, NextToken becomes a short loop to skip the spaces and illegal characters, followed by a case statement, and it looks like this:

```
begin {NextToken}
while chTypes[ch] in [chSpace,chDollar,chIllegal] do begin
   if chTypes[ch] in [chIllegal,chDollar] then begin
      tpos := cpos;
      tlineNumber := lineNumber;
      FlagError(1);
      end; {if}
   NextCh;
   end; {while}
tpos := cpos;
tlineNumber := lineNumber;
case chTypes[ch] of
```

```
      chLetter: begin
         <<<handle a reserved word or identifier>>>
         end;

      chNumber: begin
         <<<handle an integer constant>>>
         end;

      chQuote: begin
         <<<handle a string constant>>>
         end;

      otherwise: <<<handle a reserved symbol>>>;

      end; {case}
   end; {NextToken}
```

Reserved Words and Identifiers

Going back to our sample program,

```
   100 PRINT "Hello, world."
```

you can see an example of a reserved word, PRINT. Reserved words and identifiers make up all of the parts of the program that are more or less normal looking English words and that appear outside of comments or string constants. They look pretty much the same; in fact, reserved words are just identifiers that the compiler attaches special meaning to. PRINT is one of them: the only place you can use the token PRINT in a BASIC program is at the start of a print statement. You cannot, for example, use PRINT as the name of a variable. This makes PRINT a reserved word, while PRINTER is just an identifier.

All reserved words and identifiers start with an alphabetic character. NextToken uses this fact to decide when it is starting to process a reserved word or token, branching to this section of code when it finds a letter:

```
         token.kind := ident;              {assume it is an identifier}
         len := 0;                         {read the identifier}
         while chTypes[ch] in [chLetter, chDollar, chNumber] do begin
            if len = nameLength then begin
               FlagError(4);
               len := 0;
               end; {if}
            len := len+1;
            if ch in ['a'..'z'] then
               ch := chr(ord(ch)-ord('a')+ord('A'));
            token.name[len] := ch;
            NextCh;
            end; {while}
         token.name[0] := chr(len);
         for s := tokenIndex[token.name[1]] to {check for reserved words}
            pred(tokenIndex[succ(token.name[1])]) do
            if token.name = tokenNames[s] then begin
               token.kind := s;
               goto 1;
               end; {if}
   1:
```

The scanner starts by assuming that it has an identifier, setting the kind of the token to ident to record this fact. The first thing that the compiler has to do is to collect the rest of the letters in the token, flagging an error if the number of characters exceeds the limit. The limit, which is 100 characters, is implemented as a constant called nameLength, and this same constant is used to define arrays that hold identifiers, as well as in the error message that tells you if you have created an identifier that is too long. This is a common technique in large programs, and has many advantages, not the least of which is that you can change one constant in one location, then recompile the program, and it will properly handle some other maximum length of variables – smaller to use less memory, or larger for more flexibility.

After collecting the characters in the token and setting the length of the token string, the compiler steps through its list of reserved words to see if the characters are really a reserved word. This means doing a lot of string compares, and string compares take a lot of time. As a result, it is important to use any tricks that are reasonable to cut down on the number of string compares. One simple technique that works very well is a sort of poor-man's hash table, where the reserved words are placed in an array in alphabetical order, and a second array is used to record the first and last word in the array that starts with any given letter. The array tokenIndex is the array of starting indices for each letter. It's initialized in InitArrays in the file Scanner.Pas, and the first few lines look like this:

```
      tokenIndex['A'] := abssy;
      tokenIndex['B'] := callsy;
      tokenIndex['C'] := callsy;
      tokenIndex['D'] := dimsy;
      tokenIndex['E'] := endsy;
```

So, for example, if you are scanning the list for a token called Apple, you only need to loop from tokenIndex['A'] to tokenIndex['B']-1. While there is some fancy indexing, that's basically what the for loop that looks for reserved words is doing. Using this technique, the worst case is five string compares, something that happens for the letter T. For many letters, like J, there are no reserved words, and the scanner will not do a single string compare, since Pascal's for loop doesn't execute at all if the loop condition is false at the start of the loop, as it would be for the letter J. If you're not into Pascal enough for that to make sense, try computing the start and stop values for a symbol that starts with a J, and see for yourself why there are no string compares.

Scanning is actually one of the most time-consuming parts of the compilation process, so any effort made in making the scanner faster pays off well. One of the tricks you can do to speed up this particular loop is to use something other than alphabetical order for the tokens. For example, the entries for the tokenNames array for the letter P are in this order (see InitArrays again for the complete list):

```
    tokenNames[pdlsy    ] := 'PDL';
    tokenNames[plotsy   ] := 'PLOT';
    tokenNames[popsy    ] := 'POP';
    tokenNames[printsy  ] := 'PRINT';
```

Now, it doesn't take a rocket scientist to realize that on the average there will be more PRINT statements in a program than PDL function calls, POP function calls, or even PLOT statements. There may be a program here or there where this isn't true, but making a program faster is often a process of playing the averages, and on the average, PRINT will occur more often. Putting PRINT first in the array, then, will result in an overall reduction in the number of string compares that have to be done to compile the average program. Especially in a larger language, it pays to collect several hundred programs and do a statistical analysis of the number of times each symbol is used, choosing the order of the tokenNames array based on the result. This was done for ORCA/Pascal and ORCA/C, where statistical results like this are available in the published literature, and even for the ORCA/M assembler, where the 70,000 lines of the ORCA/M package were analyzed to figure out how to order the list.

Assuming we find a match in the token table, the token kind is changed to the correct value. There is a separate token in the enumerated list; you can find the list of token names in BASICCom.Pas:

```
    tokens = (eofsy, eolnsy,             {special symbols}
                                         {constants and identifiers}
          ident, intconst, stringconst,
                                         {operators}
          colon, comma, semicolon, lparen, rparen,
          eq, lt, gt, le, ge,
          ne, plus, minus, mult, divd,
          exp,
                                         {reserved words}
          abssy, andsy, ascsy, atsy, callsy,
          colorsy, dimsy, endsy, forsy, gosubsy,
          gotosy, grsy, hgrsy, hlinsy, ifsy,
          inputsy, lensy, letsy, modsy, nextsy,
          notsy, orsy, pdlsy, plotsy, popsy,
          printsy, remsy, returnsy, rndsy, scrnsy,
          sgnsy, stepsy, tabsy, textsy, text80sy,
          thensy, tosy, vlinsy, vtabsy);
```

The reserved words are the names ending with sy, at the end of the list. The loop and array index variable from our loop that looks for reserved words is actually of type tokens, putting Pascal's strong typing to great use, and showing one more tiny place where Pascal excels as a compiler writing language.

Integers

Evaluating integers is a lot easier than dealing with reserved words and identifiers. The only trick is making sure that you don't get a numeric overflow if the number is too big to handle.

```
      chNumber: begin
         token.kind := intconst;
         token.ival := 0;
         while chTypes[ch] = chNumber do begin
            digit := ord(ch) - ord('0');
            if ((token.ival = (maxint div 10)) and (digit > (maxint mod 10)))
               or (token.ival > (maxint div 10)) then begin
               FlagError(5);
               token.ival := 0;
               end;
            token.ival := token.ival*10 + digit;
            NextCh;
            end; {while}
         end;
```

Strings

Strings are pretty easy, too: since BASIC doesn't allow a quote mark inside of a string, we don't even have to worry about that small detail – we just scan from the opening quote to the ending quote, stopping to flag an error if we hit an end of line or end of file before the end of the string is found.

```
      chQuote: begin
         token.kind := stringconst;
         len := 0;
         NextCh;
         while not (chTypes[ch] in [chQuote, chEoln, chEof]) do begin
            if len = stringLength then begin
               FlagError(6);
               len := 0;
               end; {if}
            len := len+1;
            token.str[len] := ch;
            NextCh;
            end; {while}
         if chTypes[ch] = chQuote then
            NextCh
         else
            FlagError(7);
         token.str[0] := chr(len);
         end;
```

Reserved Symbols

Reserved symbols are the punctuations marks of the BASIC language, like : to separate statements, = for assignment statements, or <= for comparisons. Most of these are a single character, so the original case statement that branches to the various scanner routines does a great job of splitting the reserved symbols apart quickly and efficiently. For example, the = character is a single token, and cannot start any multi-character token, so the entire section for this token is just

```
      chEq: begin
         token.kind := eq;
         NextCh;
         end;
```

In a few cases, things get just a little more complicated. For example, > can be a token by itself, but it can also be the first character of the token >=. This still isn't hard to handle, though:

```
      chGt: begin
         NextCh;
         if ch = '=' then begin
            token.kind := ge;
            NextCh;
            end {if}
         else
            token.kind := gt;
         end;
```

All of the other reserved symbols are handled like one of these cases.

Reporting Errors

The scanner is the part of the compiler most closely connected to input and output, although the code generator writes to the output file. The scanner is the only part of a compiler that has direct knowledge of things like character positions for the tokens, the exact contents of the source file, or the name of the source file. Since this is the only place this information is available, and since all of this information is needed for accurate, useful error reporting, error handling is a chore that gets stuffed into the scanner.

Throughout the compiler, when an error is found, a call is made to the subroutine FlagError. The caller only has to pass a single parameter, the number of the error, yet when errors are printed, you get a lot more information. To see where all of this information comes from, let's go back to the beginning, in the NextToken subroutine that collects characters to build tokens.

If you examined the main loop for the NextToken subroutine carefully, you may recall these statements:

```
   tpos := cpos;                          {remember where this token starts}
   tlineNumber := lineNumber;
```

NextToken is using tpos and tlineNumber to keep track of the character position at the start of the token. If an error is found while the token is being processed, FlagError knows the position.

FlagError basically just records the errors that are reported in an array:

```
begin {FlagError}
if numErr < maxErrors then begin        {prevent an array overflow}
   numErr := numErr+1;                  {record the error}
   numErrors := numErrors+1;
   if numErr = maxErrors then
      errNum := 2;
   with errors[numErr] do begin
      err := errNum;
      pos := tpos;
      lineNumber := tlineNumber;
      end; {with}
   end; {if}
end; {FlagError}
```

The work of actually printing error messages falls on PrintLine, which is called by NextCh each time an end of line character is encountered. If any errors have been found, or if the user asked the compiler to print a source listing, PrintLine prints the line to standard out. It then steps through the error array, using the carefully recorded error position to point right at the offending token, and the error number to print the appropriate message. Sometimes, we don't find an error until after we've scanned past the line where it really occurred. In that case, the line number is available and PrintLine prints the line number and character position of the actual error.

Here's PrintLine with most of the error messages left out (you can view them in Scanner.Pas, if you would like to see the full list):

```
begin {PrintLine}
if listSource or (numErr <> 0) then begin
   write(lineNumber:4, ' ');         {print the line number}
   ptr := lineStart;                 {print the line}
   while (ptr^ <> ch_eoln) do begin
      write(chr(ptr^));
      ptr := bytePtr(ord4(ptr)+1);
      end; {while}
   writeln;
   for i := 1 to numErr do begin     {print any errors}
      case errors[i].err of
         1:  msg := 'illegal character';
         2:  msg := 'further errors supressed';
         3:  msg := '''=''' expected';
               .
               .
               .
         32: msg := 'type conflict';
         33: msg := 'expression type must be string';
         34: msg := 'string compares for =, # only';
         end; {case}
      if lineNumber = errors[i].lineNumber then
         writeln('^ ':6+errors[i].pos, msg)
      else
         writeln('Error at character ', errors[i].pos:1, ' of line ',
            errors[i].lineNumber:1, ': ', msg);
      if terminal then
         TermError(0);
      end; {for}
   if KeyPress or (wait and (numErr <> 0)) then
      while not KeyPress do ;
   end; {if}
lineNumber := lineNumber+1;          {start a new line}
lineStart := cp;
cpos := 0;
numErr := 0;
end; {PrintLine}
```

The error array is fixed in size, which presents a minor problem if there are too many errors on a single line, but in practice this doesn't happen often enough to worry about, and on the rare occasions it does happen, it is almost always because some statement was so malformed that the compiler became very confused and started spitting out all sorts of errors as it tried to get back on track. This phenomenon is called error cascading, and it's why you sometimes see several dozen errors that all go away when the first error is fixed. Integer BASIC doesn't have much problem with error cascading, since it generally skips to the end of a line when it finds any error in the structure of the program. Languages like C and Pascal don't have that advantage, though, since the end of a line in those languages literally has no more meaning than a space. Error cascading can be a nasty problem in those languages.

Shell Interface

The ORCA development environment is fairly unique in that it handles multiple languages with a single set of commands. The RUN command, for example, compiles or assembles a program, links the program, and executes the program in one step, whether the program is written in C, Pascal, assembly language, BASIC, or whatever. In fact, the RUN command can even work this way if an interpreter is used.

A lot of this flexibility is due to the way the ORCA environment interfaces with compilers. When a compiler starts, it is responsible for making a GetLInfo call, a call to the ORCA shell that returns information telling the compiler what to do. In Integer BASIC, the record that controls this call looks like this:

```
lInfoDCB = record
    sFile:      pathPtr;
    dFile:      pathPtr;
    namesList:  cStringPtr;
    iString:    cStringPtr;
    merr:       byte;
    merrf:      byte;
    opFlags:    byte;
    keepFlag:   byte;
    mFlags:     longint;
    pFlags:     longint;
    origin:     longint;
    end;
```

Let's start by looking at the mechanics for the GetLInfo call, then move on to what it all means. The first four parameters to the GetLInfo call must be set by the compiler before making the call to GetLInfo. The first two parameters point to 65 byte input buffers, and are set to ProDOS path names by the shell. The next two parameters are 256 byte buffers, and are set to null terminated strings. The remaining parameters are filled in by the shell just as would happen with a GS/OS call. Here's the first few lines of InitShell, a subroutine in Scanner.Pas that handles the GetLInfo call:

```
with lInfo do begin                  {set up the pointers}
   sFile := @sourceFile;
   dFile := @keepFile;
   namesList := @names;
   iString := @parameters;
   end; {with}
Get_LInfo(lInfo);                    {read the command line stuff}
if ToolError <> 0 then
   TermError(1);
```

Some of the parameters to the GetLInfo call have fairly esoteric uses, and we won't deal with them here. I'll run through them in a moment, but for now, I'll concentrate on the parameters used by Integer BASIC.

Those first two parameters, sFile and dFile, are the source file and destination file. The source file is the full path name of the file the compiler should compile, so it is the file that will be opened in a moment. The scanner takes care of some housekeeping first, though, deleting any old object files that have been left around from previous compiles:

```
destroyRec.pathName := @name;        {delete any old obj files}
name := concat(keepFile, '.root');
P16Destroy(destroyRec);
suffix := '.a';
repeat
   name := concat(keepFile, suffix);
   P16Destroy(destroyRec);
   suffix[2] := succ(suffix[2]);
until ToolError <> 0;
```

To understand what this code is doing, stop and recall the naming convention used by languages that run under the ORCA environment. The output file name is not used as is, but instead, the compiler or assembler appends a .root to the name, and places the first subroutine in this file. The remaining subroutines are placed in a file with .a appended to the output name. Using multi-lingual compiles or partial assemblies, it is also possible to have files that end in .b, .c, and so forth. With this in mind, the purpose of the code becomes clear: it is deleting first the .root file, then the .a file, and then continues to delete files with higher alphabetic suffixes until it doesn't find any more files.

The namesList and iString fields are not used by Integer BASIC. NamesList is used by compilers that can have more than one subroutine; it is a blank delimited list of the subroutines that should be compiled on a partial compile. The iString field is used by compilers that support language-specific parameters; to date, only C compilers have used this feature.

The language information record continues with four one-byte fields, only two of which are used by Integer BASIC. The first is merrf, which will be set by Integer BASIC before it returns to the shell; we'll look at that process later. The other is kFlag, which tells the compiler where it is in the keep process. If an output file name has been specified, kFlag will be 1. If kFlag is not set to 1, the shell hasn't provided an output name, and in that case, the compiler should only create an output file if the source contains something like a keep directive. Throughout the ORCA environment, the assumption is made that no file will be created unless the user has given a file name, so it would not be appropriate to create one automatically from the source file name. After all, if the person using the compiler wanted a name to be formed automatically from the source file name, there is a shell variable he could use to tell the shell to form the name, and that name would have been passed on to the compiler. If there is no output name, the compiler should simply check for errors and return.

The compiler keeps track of the keep flag, saving it for later use by the code generator, which will handle the output file:

```
keep := lInfo.keepFlag <> 0;       {set the command line flag variables}
```

The mFlags and pFlags fields are the flags typed by the person using the compiler when the compiler was called. These are bit fields, which are compact, but hard to use. For that reason, the compiler sets up boolean variables, plucking out the appropriate bits to set the variables:

```
debug := (lInfo.pFlags & flag_d) <> 0;
edit := (lInfo.pFlags & flag_e) <> 0;
listSource := (lInfo.pFlags & flag_l) <> 0;
memory := (lInfo.pFlags & flag_m) <> 0;
listSymbols := (lInfo.pFlags & flag_s) <> 0;
terminal := (lInfo.pFlags & flag_t) <> 0;
wait := (lInfo.pFlags & flag_w) <> 0;
```

These boolean variables are used through the compiler to handle such diverse tasks as generating debug code, listing the machine code generated by the compiler, and deciding if the editor should be called when an error is encountered.

Right after making the GetLInfo call, the scanner calls ReadFile to actually read in the source file. Here's the ReadFile subroutine:

```
begin {ReadFile}
with ffRec do begin                  {read the file from disk}
   action := 0;
   index := 0;
   flags := $C000;
   name := @sourceFile;
   end; {with}
FastFile(ffRec);
if ToolError <> 0 then
   TermError(2);
fileLength := ffRec.file_length;     {record the file length}
cp := bytePtr(ffRec.file_handle^);   {set the file pointer}
end; {ReadFile}
```

All compilers that want to handle PRIZM gracefully must read source files the way Integer BASIC does, by using a FastFile call. The reason is simple: if the user has made changes to the file on the desktop, but has not saved these changes to disk, making calls to ProDOS or GS/OS will get the old, disk version of the file, not the most recent version. Using FastFile also has the advantage of speed, since the file won't have to be reloaded from disk if it is already in memory.

So far, we have looked at how the shell tells the compiler what to do, and how the compiler finds out using the GetLInfo call. The next step is to look at the SetLInfo call, where the compiler tells the shell what it did just before returning to the shell. This process is carried out in ShutDownScanner, where the first task is to call the FastFile system again, releasing the source file:

```
begin {ShutDownScanner}
with ffRec do begin                      {mark the source file as purgeable}
   action := 7;
   index := 0;
   flags := $C000;
   name := @sourceFile;
   end; {with}
FastFile(ffRec);
```

There are two basic conditions that we have to deal with: either the program compiled correctly, and it is time to move on to the link step; or the program contained errors, and the shell must be informed so the link step is not performed. The last part of the subroutine handles these conditions.

```
if numErrors <> 0 then begin            {set the correct error codes}
   lInfo.opFlags := 0;
   if terminal then
      lInfo.merrf := 128
   else
      lInfo.merrf := maxErrorLevel;
   end {if}
else begin
   lInfo.opFlags := lInfo.opFlags & $FFFE; {set the opflags field}
   if lInfo.keepFlag = 0 then
      lInfo.opFlags := 0;
   lInfo.sFile := @keepFile;            {set the link file name}
   end; {else}
Set_LInfo(lInfo);                       {pass the info back to the shell}
end; {ShutDownScanner}
```

The scanner's error handler keeps track of the total number of errors found by the compiler in the variable numErrors. If this number is not zero, something when wrong. The first thing we do is clear the opFlags field of the language information record we are about to use with the SetLInfo call, which passes information back to the shell. The opFlags byte uses three bits to tell languages what action to take: the least significant bit, $0001, tells the language to compile the program; the next bit, bit $0002, tells the linker to link the program; and the $0004 bit is a message back to the shell telling it to run the finished program. Stop and think about the complement of commands you can use from the shell to compile a program. The COMPILE command compiles the program, but does not link or execute it, so it sets the opFlags field to $0001. The CMPL command compiles and links, but does not run the program, so it sets the opFlags field to $0003. The LINK command links a program, but does not execute it; the opFlags field gets set to $0002. Finally, the RUN command does it all, and sets opFlags to $0007. If we have found an error, though, we need to make sure the linker is not called, and that the program is not executed, so we set opFlags to 0.

When the compiler exists with an error, the shell can take one of two actions: it can stop, returning to the command line prompt, or it can call the editor right away, displaying the offending line and an error message. The shell makes this choice based on the merrf field. A value if 128 or greater tells the shell to enter the editor, while a smaller value tells the shell not to enter the editor. One of the flags we recorded back when GetLInfo was called was called terminal, and was set to true if the +t flag was used. Here, the compiler checks to see if terminal is true, and if so, sets merrf to 128.

Assuming things went well, the least significant bit of the opFlags field is cleared before returning to the shell. This tells the shell that the compilation process is finished, and that no other languages have to be called, as would happen if we used the append command in Pascal to append an assembly language file, for example. If there was no output file, all of the opFlags are cleared, since there is no point in doing the link if there is nothing to link. Finally, we set the source file to the same name the shell passed to us as the destination file. After all, the next program that will be called is the linker, and the input to the linker is the output file created by the compiler.

After setting up the parameters for the SetLInfo call, the ShutDownScanner subroutine makes the call itself and returns.

The whole process of calling SetLInfo occurs in one other place, too. There are some errors, such as disk I/O errors or out of memory errors, which are so sever that the compiler doesn't even try to go on. These are called terminal errors, and are handled by TermError. TermError handles things just like we described here, so I won't go over it in detail, but you should know that SetLInfo can be called from two different places.

Other Uses for Scanners

The scanner we've just dissected can be pulled out of the compiler and used for a variety of different purposes. I think it might be fun to stop for a moment and realize just how much you've learned to do before plowing on into the parser.

This scanner can be quickly adapted to handle almost any language you will ever use, even English. By keeping track of the various identifiers that are found, you could generate a great set of utilities. For example, if you track the identifiers and the line numbers, you could crank out a cross reference utility for BASIC. Changing the list of reserved words and adding a short section of code to handle comments and the differences in the way strings are used, and you could have a cross reference generator for C or Pascal.

A spelling checker works a lot like a scanner, too. You could collect the words, then look them up in a separate dictionary file to create a simple, but very useful, spelling checker.

Back when we looked at reserved words, I mentioned that it would be useful to know how often a particular reserved word is used to optimize the scanner. Knowing how often English words are used is one of the tidbits of information you need to know to optimize a spelling checker. This scanner can be quickly adapted to count how often reserved words or identifiers are used by just tacking a counter onto the array of reserved words, or creating a linked list of the identifiers you find in English and adding a count field.

There are a lot of other uses for just a scanner, and quite a few uses for a scanner combined with a parser. We'll look at a few more after learning about the parser in the next section.

The Parser

There are many ways to write a compiler, but the two most popular are undoubtedly the table driven parser and the recursive descent parser. Table driven parsers are favored at universities, where the emphasis is often on the design of a language and where the design is frequently changed as more is learned about the language. Table driver parsers are also used frequently in commercial compilers, but in situations where a compiler is being written for a well defined language (in other words, one that already exists!) my feeling is that table driven parsers have no particular advantage over a recursive descent parser. Both have advantages and disadvantages, but on balance, they both are reasonable choices. Two things that make a recursive descent parser very attractive for this compiler is that no special tools are required, as they are for table driven parsers, and nothing is hidden from you, as it would be if you fed a formal grammar into a parser generator, and got a program out the other end. As Houdini might have put it, "Look, ma, nothing up my sleeve!"

Integer BASIC uses a recursive descent parser, which is actually very easy to understand once you learn a bit about how they are built. The main program for Integer BASIC looks like this:

```
begin {BASIC_Compiler}
writeln('Integer BASIC 1.0');           {write the header}
writeln('Copyright 1991, Byte Works, Inc.');
writeln;
InitCommon;                             {initialize the globals area}
InitScanner;                            {initialize the scanner}
InitParser;                             {initialize the parser}

Compile;                                {compile the program}

writeln;                                {write the trailer}
writeln(numErrors:1, ' errors found.');
if maxErrorLevel <> 0 then
   writeln(maxErrorLevel:1, ' was the highest error level.');
ShutDownScanner;                        {shut down the scanner}
end. {BASIC_Compiler}
```

The compiler performs some initialization, then calls Compile. The Compile procedure is actually the top level of the parser, and you can find it in Parser.Pas. Once the parser decides that the program is complete (or so buggy nothing else can be done), it returns control to the top level of the program, which shuts things down and quits.

Compile: At the Top

The parser's job is to make sure that the tokens read by the scanner form a legal program. In technical terms, the parser verifies that the program's syntax is correct. In the process, of course, the parser learns something about the meaning of the program, at least to the extent that it sees which statements are being compiled and verifies that statements have the correct form. Deciding what the program means is the job of semantic analysis, which we will cover separately, but parsing and semantic analysis are closely linked. For now, I'm going to leave out the details of semantic analysis and look at some of the parsing subroutines after stripping out the semantic analysis code. Later, we will look at the same subroutines again, and see how semantic analysis flows smoothly from parsing, but I want to keep things as simple as possible while we look at the parser.

In Chapter 2, when we defined what a program was, we used a loose form of a design language called BNF. While we only did this for expressions and statements, let's take a moment now to do the same thing for an entire BASIC program. Using BNF, our definition of a BASIC program looks like this:

```
program    ::= [ line ]*
line       ::= integer statement [ : statement ]*
```

Since BNF probably isn't your most natural way of thinking about a language (yet), let's take just a moment to see what those lines tell us, and how they fit into the Integer BASIC language described in Chapter 2. These lines tell us that an Integer BASIC program consists of zero or more lines. Implied in this, as far as the parser is concerned, is that there is some way to tell when there are no more lines, and some way to tell when we get to the end of a line. As you saw when we looked at the scanner, this is done through two tokens, one of which is reported at the end of a line, and the other of which is reported at the end of the BASIC program.

The second line tells us that a line from a BASIC program must start with a line number. This is followed by a statement, which can be followed by any number of other statements, separated from one another by colons.

Now let's look at the procedure Compile, from Parser.Pas. Remember, this isn't exactly what you will see if you look in the program, since this version of the subroutine has been stripped of all semantic routines; it's just the bare parser.

```
begin {Compile}
while token.kind <> eofsy do begin
   if token.kind = intconst then        {handle a line number}
      NextToken
   else
      FlagError(9);
   Statement;                           {compile a statement}
   while token.kind = colon do begin    {compile any other statements}
      NextToken;
      Statement;
      end; {while}
   if token.kind <> eolnsy then begin   {check for garbage on the line}
      FlagError(10);
      while not (token.kind in [eolnsy, eofsy]) do
         NextToken;
      end; {if}
   NextToken;                           {skip the eoln}
   end; {while}
end; {Compile}
```

Looking at the subroutine we see that the parser is going to process something until it gets to eofsy, which is the token returned by the scanner when it gets to the end of the file. In other words, it's going to process each line in the program. Another great way to say exactly the same thing is:

```
program    ::= [ line ]*
```

On each line, the parser starts by looking for a line number, or, as the scanner reports such things, an intconst (integer constant). If it finds one, the parser skips it by calling NextToken; if not, the compiler flags an error. Either way, we move on to a call to a procedure called Statement. Next we look for a colon, and if we find one, we skip the colon and call Statement again. We keep doing this until we don't see a colon. Said another way,

```
line       ::= integer statement [ : statement ]*
```

where it is an error if the integer constant is not there. We'll let the procedure Statement worry about whether the statement is actually there, since at this level, we don't care what a statement looks like. In fact, it could contain exactly nothing – and in languages like Pascal and C, that is a perfectly legal, and not uncommon, possibility.

At this point, we should be at the end of a line, so the parser checks to make sure that the next token is, in fact, and end of line mark. If not, something strange is happening. A lot of strange things can happen, so rather than try to make uneducated guesses at what this particular strangeness might be, the parser simply flags an error, then throws away anything else until it gets to the end of a line (or the end of the file – the input may be so strange that the end of line mark isn't there). The last step is to discard the end of line token and go back for more. Implicit in this step is the correct assumption that once the scanner finds the end of file token, it will continue to report that token no matter how many times you call it; this simple assumption saves us a lot of redundant checking in the parser.

It may not seem like it, but with just a couple more tricks you'll learn in a moment, you're now equipped to go out and write a parser for any computer language I've ever seen, assuming someone gives you the BNF or its equivalent. It seems like magic right now, but the whole essence of writing a recursive descent parser is really just repeating what was just done. You start with the BNF for the language, or a formal grammar, or even syntax diagrams – they all mean the same thing, even if some are more detailed than others – and you write the parser directly from the description of the language. Sure, we haven't done anything fancy yet, like handle expressions, but that's the beauty of a recursive descent parser: it's one of the purest forms of structured programming, where you start at the top level, and gradually refine the program by adding more and more detailed lower levels. When you run out of BNF, the parser is finished. It's really that simple.

The Statement Procedure

The next level of the parser again depends on a formal definition that you generally don't find in user's manuals for a language, and that is the BNF for Statement. It looks like this:

```
statement  ::= DimStatement
statement  ::= CallStatement
statement  ::= ColorStatement
statement  ::= EndStatement
statement  ::= ForStatement
statement  ::= GosubStatement
statement  ::= GotoStatement
statement  ::= GRStatement
statement  ::= HGRStatement
statement  ::= HlinStatement
statement  ::= IfStatement
statement  ::= InputStatement
statement  ::= NextStatement
statement  ::= PlotStatement
statement  ::= PopStatement
statement  ::= PrintStatement
statement  ::= RemStatement
statement  ::= ReturnStatement
statement  ::= TabStatement
statement  ::= TextStatement
statement  ::= Text80Statement
statement  ::= VlinStatement
statement  ::= VtabStatement
statement  ::= LetStatement
```

This looks obvious and even a little silly once you see it laid out in such unneeded detail, but all it says is that a BASIC statement consists of one of the BASIC statements described in Chapter 2. More important, from the standpoint of our parser anyway, is that a careful check of the formal descriptions of the statements shows that each and every one starts with a specific reserved word; these reserved words are all different from one another; and with the exception of the LET statement, the reserved word that starts the statement is required. This is a very good thing in a language. It means that we can look at the very first token and know without a doubt what statement we are supposed to compile. That, in a nutshell, is exactly what the parser's Statement procedure does:

```
begin {Statement}
case token.kind of
   dimsy:       DimStatement;
   callsy:      CallStatement;
   colorsy:     ColorStatement;
   endsy:       NextToken;
   forsy:       ForStatement;
   gosubsy:     GosubStatement;
   gotosy:      GotoStatement;
   grsy:        NextToken;
   hgrsy:       NextToken;
   hlinsy:      HlinStatement;
   ifsy:        IfStatement;
   inputsy:     InputStatement;
   nextsy:      NextStatement;
   plotsy:      PlotStatement;
   popsy:       NextToken;
   printsy:     PrintStatement;
   remsy:       RemStatement;
   returnsy:    NextToken;
   tabsy:       TabStatement;
   textsy:      NextToken;
   text80sy:    NextToken;
   vlinsy:      VlinStatement;
   vtabsy:      VtabStatement;
   letsy:       begin NextToken; LetStatement; end;
   otherwise:   LetStatement;
   end; {case}
end; {Statement}
```

There are only two minor oddities here. The first is that several of the statements, like the GR statement, are implemented simply as a call to NextToken to dump the token and get ready for the next line. Looking at the definition of the GR statement, though, you will see that it consists simply of the token GR. As far as the parser is concerned, then, all we have to do for a GR statement is dump the token and move on.

The other oddity comes at the end of the case statement, when we handle the possibility that the token wasn't anything we were expecting. In that case we call LetStatement, since that is the only possibility in a correct program. That's also why, on the line above, we see that the call to LetStatement is the only time the Statement procedure dumps the reserved word token before calling the subroutine that will process the statement. Since the last line can call LetStatement with no leading LET token, it makes the LetStatement procedure a little easier if we get rid of the leading LET token before calling it.

The remainder of the parser involves stepping through the various statements, implementing the formal definition of the statement from Chapter 2. After a while, it would get pretty boring to try and analyze each and every one of these subroutines. We'll just look at a few representative examples.

The LET Statement

So far, writing the parser looks pretty easy, but we haven't gotten to the statements yet, so you might be thinking I'm still avoiding the hard parts. Gradually, I hope you start to see that I'm not avoiding them at all. There aren't any hard parts to writing a parser.

Let's look at a real statement now, and one that is genuinely hard for a compiler to handle. A great deal of work and effort has gone into the assignment statement. Looking at the definition of the assignment statement, though, it doesn't look so frightening:

```
[ LET ] l-value = expression
```

And it isn't frightening, we just chug right along just like we've been doing:

```
begin {LetStatement}
LValue;
Match(eq, 3);
Expression([]);
CheckEol;
end; {LetStatement}
```

There are a couple of handy tricks here, but nothing earth shattering. Since the Statement procedure has already gotten rid of the LET token if there was one, we start in by evaluating the l-value. In great structured programming form, we do that by calling another subroutine. In case you haven't noticed, with the exception of the top level of the program, whenever the BNF description of the language refers to another, more detailed line of BNF, we follow suite by using a procedure call.

The BNF shows that the l-value is followed by an equal sign, which we handle by calling a procedure called Match. This procedure is just a short procedure that verifies that a particular token exists, flagging an error if it doesn't. We could do the same thing with

```
if token.kind = eq then
   NextToken
else
   FlagError(3);
```

but using a call to Match makes the parser shorter and easier to read. The parser uses surprisingly less memory, too.

The next line calls the procedure Expression. For the first time, though, we are passing along a parameter; this one is the empty set. This parameter tells Expression what tokens can follow the expression itself, other than the end of a line. It is used by Expression when it finds an error. Do you remember how the top level of the compiler handled an error? It skipped to the end of a line. Well, Expression will handle an error by skipping to the end of the expression, but in some cases, like this IF statement:

```
100 IF I = 4 + THEN 450
```

the expression isn't followed by the end of a line. If Expression were to throw away everything up to the end of the line, when we got back, the parser would start flagging other errors because the THEN token would have been thrown away, too. In the IF statement, though, Expression would be called with a parameter of [thensy], telling the Expression procedure that, if it finds an error, it should skip until it finds the end of a line *or a THEN token*. This simple trick is incredibly effective for flagging errors sensibly.

The last line is a call to CheckEOL, which simply makes sure that there is nothing else on the line. If so, an error is flagged, and all of the extra stuff is thrown away.

L-Values

When you saw the technical description of an l-value in the last chapter, I glossed over the concept of an l-value, saying it was something you could assign a value to. More specifically, in Integer BASIC, it is either a simple variable or an element of an array. Using BNF, it looks like this:

```
l-value    ::= identifier [ ( expression ) ]
```

Writing the same thing in Pascal, we get the parser's version of the LValue procedure called by LetStatement:

```
begin {LValue}
if token.kind = ident then begin
   NextToken;
   if token.kind = lparen then begin
      NextToken;
      Expression([rparen]);
      Match(rparen, 24);
      end; {if}
   end {if}
else
   FlagError(27);
end; {LValue}
```

There are certainly some details to handle later, when we try to attach meaning to the assignment statement, but from the parser's viewpoint, this is all there is to an l-value.

Expressions

Expressions must be hard to evaluate. Everyone seems to think so. Expressions are so hard to evaluate that the folks who created spread sheets got freaked, and didn't handle expression precedence. Expressions are so hard to handle that many authors of assemblers took a shortcut, and didn't implement expression precedence or very many operators. Expressions are so complex that HP couldn't get them right in their calculators for years.

Well, expressions *are* hard if you don't know what you are doing, but you do know what you are doing, now. Take another look at the BNF for expressions from the last chapter:

```
expression     ::= andop [ OR andop ]*
andop          ::= cmpop [ AND cmpop ]*
cmpop          ::= plusminus [ = | < | > | <= | >= | # | <>
                       plusminus ]*
plusminus      ::= mulop [ + | -  mulop]*
mulop          ::= exponent [ * | / | MOD  exponent ]*
exponent       ::= term [ ^ term ]*

term           ::= NOT term
term           ::= - term
term           ::= + term
term           ::= integer
term           ::= string
term           ::= identifier
term           ::= identifier ( expression [ , expression ] )
term           ::= ( expression )
term           ::= ASC ( expression )
term           ::= LEN ( expression )
term           ::= ABS ( expression )
term           ::= SGN ( expression )
term           ::= PDL ( expression )
term           ::= RND ( expression )
term           ::= SCRN ( expression , expression )
```

By now, just glancing at these lines, you should start to see subroutines pop into view. In Chapter 2, I showed you how this BNF for expressions implemented operator precedence, expressions imbedded in other expressions, and all of the other features we need to make the BASIC language sing in integers and strings. With what you know now, you can see how the parser can quickly process a token stream to make sure it is a legal version of an expression. Here's the whole implementation of expression parsing, all in a single lump. Start at the top of the BNF chart, and the bottom of the Expression procedure, and follow through to see how they match up exactly with one another. Truly, there is literally nothing more to writing a recursive descent parser than coding the BNF charts. Expressions, in fact, are the hardest part of the language to parse. They are one of the hardest parts of the compiler to handle semantically, as well, but when we look at these subroutines again for semantics, you will see that the way the parser breaks them down makes even the semantics seem almost trivial.

```
procedure Expression (stop: tokenSet);

   procedure AndOp;

      procedure CmpOp;

         procedure PlusMinus;

            procedure MulOp;

               procedure Exponent;

                  procedure Term;

                  begin {Term}
                  case token.kind of
```

```
notsy: begin       {not}
   NextToken;
   Term;
   end;

minus: begin       {-}
   NextToken;
   Term;
   end;

plus: begin        {+}
   NextToken;
   Term;
   end;

lparen: begin      {(}
   NextToken;
   Expression(stop+[rparen]);
   Match(rparen, 24);
   end; {lparen}

intconst: begin    {integer constant}
   NextToken;
   end;

stringconst: begin {string constant}
   NextToken;
   end;

ident: begin       {identifier}
   NextToken;
   if token.kind = lparen then begin
                   {handle an array}
      NextToken;
      Expression(stop+[comma]);
      if token.kind = comma then begin
         NextToken;
         Expression(stop+[rparen]);
         end; {if}
      Match(rparen, 24);
      end; {if}
   end;

ascsy: begin       {asc function}
   NextToken;
   Match(lparen, 23);
   Expression(stop+[rparen]);
   Match(rparen, 24);
   end;
   lensy: begin       {len function}
      NextToken;
      Match(lparen, 23);
      Expression(stop+[rparen]);
      Match(rparen, 24);
      end;

   abssy: begin       {abs function}
      NextToken;
      Match(lparen, 23);
      Expression(stop+[rparen]);
      Match(rparen, 24);
      end;

   sgnsy: begin       {sgn function}
      NextToken;
      Match(lparen, 23);
      Expression(stop+[rparen]);
      Match(rparen, 24);
      end;

   pdlsy: begin       {pdl function}
      NextToken;
      Match(lparen, 23);
      Expression(stop+[rparen]);
      Match(rparen, 24);
      end;

   rndsy: begin       {rnd function}
      NextToken;
      Match(lparen, 23);
      Expression(stop+[rparen]);
      Match(rparen, 24);
      end;

   scrnsy: begin      {scrn function}
      NextToken;
      Match(lparen, 23);
      Expression(stop+[comma,rparen]);
      Match(comma, 13);
      Expression(stop+[rparen]);
      Match(rparen, 24);
      end;

   otherwise:
      FlagError(27);

   end; {case}
end; {Term}
```

```
               begin {Exponent}
               Term;
               while token.kind = exp do begin
                  NextToken;
                  Term;
                  end; {while}
               end; {MulOp}

            begin {MulOp}
            Exponent;
            while token.kind in [mult, divd, modsy] do begin
               NextToken;
               Exponent;
               end; {while}
            end; {MulOp}

         begin {PlusMinus}
         MulOp;
         while token.kind in [plus, minus] do begin
            NextToken;
            MulOp;
            end; {while}
         end; {PlusMinus}

      begin {CmpOp}
      PlusMinus;
      while token.kind in [eq, lt, gt, le, ge, ne] do begin
         NextToken;
         PlusMinus;
         end; {while}
      end; {CmpOp}

   begin {AndOp}
   CmpOp;
   while token.kind = andsy do begin
      NextToken;
      CmpOp;
      end; {while}
   end; {AndOp}

begin {Expression}
AndOp;                                 {evaluate the term}
while token.kind = orsy do begin       {handle or operations}
   NextToken;
   AndOp;
                                       {skip any extraneous tokens}
if not (token.kind in stop+[eolnsy, eofsy, colon]) then begin
   FlagError(11);
   while not (token.kind in stop+[eolnsy, eofsy, colon]) do
      NextToken;
   end; {if}
end; {Expression}
```

There is only one interesting feature in this whole section of code, and that's how the stop symbol is handled. I could spend a lot of time and ink with a verbose description of this subroutine and how it matches the BNF, but you can do a much better job of convincing yourself just how easy and straight-forward the whole precess is by tracing through the BNF and the subroutine yourself. If you didn't do that, stop and take a few moments to do it now.

At the very end of the body of the Expression procedure, we come to this:

```
                                         {skip any extraneous tokens}
if not (token.kind in stop+[eolnsy, eofsy, colon]) then begin
   FlagError(11);
   while not (token.kind in stop+[eolnsy, eofsy, colon]) do
      NextToken;
   end; {if}
```

This is the only place where the code doesn't match the BNF exactly. All this code really does is implement some practical experience gained by a lot of compiler writers over the years. The result is much better error handling than you would normally get. What these statements do is handle extraneous tokens.

Let's say, for example, that you type the following statement, where you inadvertently left out a + character. It's an easy typographical mistake to make, even if you are an experienced programmer.

```
100 IF i = 4 J THEN 200
```

Tracing through either the Expression subroutine or the BNF, you can see that the expression parser will process "i = 4", then stop. After all, it has a complete expression, and it isn't expecting another identifier, so the expression parser assumes the token isn't part of the expression and returns. The parser for the IF statement will look at this token, realizing that it doesn't match the expected THEN statement, and start spewing out senseless errors.

The way the Expression procedure is actually implemented, though, when it finishes parsing the expression, it checks to see what the next token is. In fact, it asks if the token is in the set stop+[eolnsy, eofsy, colon]. Since the IF statement would have passed [thensy] as the stop parameter when Expression was called, the entire set is [thensy, eolnsy, eofsy, colon]; the next token is an identifier, which is not a member of the set, so the expression parser throws the token away, stopping when it sees the THEN token. This simple practice insures that the parser can go on, generating appropriate error messages for the rest of the program.

You can see this concept at work in the procedure Term, too, where expression is called recursively. When this happens, the stop set is passed as stop+[rparen], adding one more symbol to those the parser can stop on to synchronize itself in case of an error.

As an aside, this mechanism is very easy to implement in a language like Pascal, which supports sets. It is much more difficult, and generally less efficient, to implement this idea in languages like C that don't have sets.

The GOTO Statement, IF Statement, and FOR Statement

At this point, the only reason for looking at any more statements is to give you a little more practice and confidence. You have already seen all of the tricks and techniques you need to go right ahead and implement the entire parser for Integer BASIC.

For more practice, we'll look at three statements we'll spend more time on in semantics, when things admittedly get a little dicier. Here's the BNF for the GOTO statement, IF statement, and

FOR statement, followed in each case by the parser's code that handles the statement. As you can see, there isn't much to it.

```
GotoStatement     ::= GOTO integer

   begin {GotoStatement}
   NextToken;                          {skip the GOTO token}
   if token.kind = intconst then       {handle the line number}
      NextToken
   else
      FlagError(9);
   CheckEol;
   end; {GotoStatement}


IfStatement       ::= IF expression THEN integer
IfStatement       ::= IF expression THEN statement

   begin {IfStatement}
   NextToken;                           {skip the if}
   Expression([thensy]);                {evaluate the condition}
   Match(thensy, 15);                   {make sure the then is there}
   if token.kind = intconst then begin
      NextToken;
      CheckEol;
      end {if}
   else
      Statement;
   end; {IfStatement}
```

```
ForStatement      ::= FOR identifier = expression TO expression
                      [ STEP [-] integer ]

   begin {ForStatement}
   NextToken;                        {skip the for}
   if token.kind = ident then        {do the initial assignment}
      NextToken
   else
      FlagError(17);
   Match(eq, 3);
   Expression([tosy]);
   Match(tosy, 18);                  {check for the to}
   Expression([stepsy]);             {evaluate and save the stop value}
   if token.kind = stepsy then begin {evaluate the step size}
      NextToken;
      if token.kind = minus then
         NextToken;
      if token.kind = intconst then
         NextToken
      else
         FlagError(19);
      end; {if}
   CheckEol;                         {check for junk at the end of line}
   end; {ForStatement}
```

Other Uses for Parsers

A parser is a wonderfully flexible tool. If you stop to think about it for a moment, what we have done so far is to create a program that checks to make sure a file full of text characters matches the syntax of the Integer BASIC language. In one sense, we've checked a program for errors.

To see one way to put this to use, consider another language that is near and perhaps dear to your heart – and certainly within a foot or two of your eyes, as well: English. Back in grade school, some flaky lady spent a great deal of time indoctrinating me and a few other helpless children in the art of diagramming sentences. Guess what? Diagramming sentences is a whole lot like comparing an Integer BASIC line to the BNF for Integer BASIC. In short, with an appropriate dictionary and some judicious algorithms to back up and try a slightly different form when you get into trouble, you could create a parser for English, and use it to check little details like punctuation and in some cases grammar. Let me know when you finish it. I need a copy.

Semantics

Semantic analysis is the process of assigning meaning to the program. It is here, in the semantic analysis routines, that we decide what action needs to be taken to carry out the wishes of the programmer. Keep in mind that the ultimate goal for this compiler is to take an Integer BASIC program as input, and to write a machine language program that does exactly the same thing. It makes sense, then, that the semantic subroutines will decide what machine language instructions are equivalent to the Integer BASIC statements. To accomplish this, we'll use a series of subroutine calls which match up to machine language instructions. These subroutines are actually a part of the code generator, which is called by the semantic analysis subroutines. The semantic analysis routines themselves are imbedded in the parser. This makes sense if you think about how much we know about the meaning of a program simply by where we are in the parsing process. For example, after entering Statement, finding a GOTO token, entering GotoStatement, and finding an integer, it doesn't take a superhuman grasp of logic to realize that the semantic analysis subroutines need to generate some sort of jump instruction, probably a machine language JMP instruction, to some label. That, as you will see, is exactly what the semantic analysis routines will do.

Semantic analysis is also where we'll start creating and using the symbol table, which is a list of all of the variables in the program. When we are parsing a program, is really doesn't matter a great deal whether types are being used incorrectly. For example, the parser would be quite happy with the statement

```
100 I = "Hello" + 1
```

but this statement doesn't make sense. Semantic analysis is where we try to make sense of the program, so we will be checking the types of constants and variables to make sure they match what the language can do, flagging errors when there is a problem. To do that, we'll have to keep track of the variables, how large the arrays are, and so forth. This information will be used later, when the program is finished, to create the space needed by the variables when the program is executed.

We'll look at semantic analysis the same way we looked at the parser, by examining some specific statements fairly carefully. Unlike the parser, the scanner isn't something you can churn out almost by rote from a description of the language – semantics is where the process of compilation gets creative. As a result, you may want to spend a little more time looking at the complete program after reading this section, looking at the different ways the various statements are handled.

Compile: Back At the Top

When we looked at the parser, we started with the top level and worked down. We'll start there again, with the Compile procedure. Here's the complete procedure, this time with all of the semantic subroutines intact:

```
begin {Compile}
Gen0Name(d_bgn, @'BASIC');              {generate the start of the program}
Gen0Name(m_jsl, @'~BASICSTARTUP');      {set up the BASIC environment}
if debug then begin                     {set up the debugger}
   Gen0String(d_sub, @'BASIC');
   Gen0String(d_fil, @sourceFile);
   symlabel := GetLabel;
   Gen0Label(m_jsr, symLabel);
   end; {if}
while token.kind <> eofsy do begin
   if debug then                        {generate the line number}
      case lineType of
         step:         Gen1(d_lnm, lineNumber);
         breakpoint:   Gen1(d_brk, lineNumber);
         autogo:       Gen1(d_ago, lineNumber);
         end; {case}
   if token.kind = intconst then begin  {handle a line number}
      if token.ival <= slineNumber then
         FlagError(8);
      slineNumber := token.ival;
      Gen0Label(d_lab, StatementLabel(token.ival, true));
      NextToken;
      end {if}
   else
      FlagError(9);
   Statement;                           {compile a statement}
   while token.kind = colon do begin    {compile any other statements}
      NextToken;
      Statement;
      end; {while}
   if token.kind <> eolnsy then begin   {check for garbage on the line}
      FlagError(10);
      while not (token.kind in [eolnsy, eofsy]) do
         NextToken;
      end; {if}
   NextToken;                           {skip the eoln}
   end; {while}
if debug then                           {shut down the debugger}
   Gen0(d_exi);
Gen0Name(m_jsl, @'~BASICSHUTDOWN');     {return to the launcher}
if debug then begin                     {generate the symbol table}
   Gen0Label(d_lab, symlabel);
   GenDebugSymbols;
   end; {if}
GenStrings;                             {generate any string constants}
Gen0(d_end);                            {flag the end of the program}
GenSymbols;                             {create the symbol segment}
CheckLabels;                            {check for undefined labels}
end; {Compile}
```

As you can see, the semantic routines added quite a bit! The very first thing we do, in fact, has to do with semantics:

```
   Gen0Name(d_bgn, @'BASIC');              {generate the start of the program}
   Gen0Name(m_jsl, @'~BASICSTARTUP');      {set up the BASIC environment}
```

Both of these statements are calls to the code generator, telling it to take specific actions. From the perspective of the semantic analyzer, the code generator is a collection of procedures, each of which takes an instruction as the first parameter. The remaining parameters can vary quite a bit, or even be missing, depending on the call. In these examples, we are passing a name as a string.

The instruction names use a simple, clear convention that makes it easier to read the code generated by the code generator. Machine language instructions all start with m_, like the m_jsl you see here. In fact, the second line is pretty easy to figure out if you know assembly language fairly well – it is a JSL to the subroutine ~BASICSTARTUP. Instructions to the code generator itself, known as directives (or sometimes pragmas) to language writers, start with d_. These have wildly varying meanings; d_bgn tells the code generator to do whatever is necessary to start a new code segment in the output file, and to call the code segment BASIC. The two lines together, then, set up a segment for the program, and call a library subroutine called ~BASICSTARTUP that will set up the run-time environment.

Integer BASIC supports the PRIZM source-level debugger, and the next few lines get ready for the debugger.

```
if debug then begin                      {set up the debugger}
   Gen0String(d_sub, @'BASIC');
   Gen0String(d_fil, @sourceFile);
   symlabel := GetLabel;
   Gen0Label(m_jsr, symLabel);
   end; {if}
```

The first two lines create the subroutine name record and file name record, respectively, used by the debugger to tell which window to use and what to call the stack frame in the variables window. The next line calls yet another code generator subroutine, GetLabel. This subroutine reserves a label number in a table of labels, and the next line uses this label number, making a subroutine call. Later, we will use a directive to tell the code generator where the label should be placed in the program. The purpose of this statement is to create the symbol table that will be used by the debugger to form the variables window, but since we haven't compiled the program, we don't know what the variables are, yet. We put this task off by making a subroutine call to a subroutine we'll create later; we'll put the code to generate the symbol table in that subroutine.

Right inside the main parsing loop, we hit yet another group of statements to handle the debugger:

```
   if debug then                         {generate the line number}
      case lineType of
         step:         Gen1(d_lnm, lineNumber);
         breakpoint:   Gen1(d_brk, lineNumber);
         autogo:       Gen1(d_ago, lineNumber);
         end; {case}
```

These statements create another debugger record that tells the debugger what line we are about to execute. The PRIZM editor actually puts a special character at the start of lines that need a break point or that are auto-go; the scanner tracks this and places the result in a variable called lineType. We use that information to decide which of the three directives we will generate at the start of the line. When the program runs, the debugger uses this information to decide where in the source window the arrow should appear, and whether it should stop a trace (a breakpoint), show the arrow (a normal line number, shown here as d_lnm), or skip through without drawing the arrow (an auto-go line). The line number used here is the physical line in the file, not the line number required by Integer BASIC; it is tracked by the scanner solely for use in generating this debug code and writing error messages.

The parser checked to make sure that the first token on the line was an integer constant; the semantic analyzer is a bit picker. It also insists that the line number be larger than any previous line number. There is no real reason why this has to be true in BASIC, but since the BASIC interpreters most people use are line editors that order the lines automatically, the lines are always in the correct order in the interpreted BASICs. This compiler simply enforces that restriction. The other thing that needs to be done is to create a label, since the line number might be used as the destination for a GOTO statement, IF branch, or a GOSUB.

```
         if token.ival <= slineNumber then
            FlagError(8);
         slineNumber := token.ival;
         Gen0Label(d_lab, StatementLabel(token.ival, true));
```

The StatementLabel subroutine is used to get the label associated with this line number. The semantic analyzer builds a table of line numbers and associated code generator label numbers. When you call the subroutine with a line number it has never seen before, a new entry is created and GetLabel, called earlier when we were setting up the symbol table for the debugger, is called to get a code generator label number. This is saved in the table for future use, and returned as the result of the StatementLabel function. If you call StatementLabel with a line number it recognizes, it simply returns the value it allocated for the line number on the previous call.

Naturally, it would be bad form to branch to a line number that wasn't in the program, so StatementLabel has one other boolean parameter, which is true if we are about to define the label (which we do by calling the code generator right away with a d_lab directive). At the end of the compile, the table of line numbers is scanned, and any that have been used as the destination for a branch but were never found are flagged as an error.

The Compile procedure continues on with the parsing chores, processing statements until the end of the program. Other than repeatedly generating code to handle the start of a line and create line number labels, the semantic analyzer isn't involved in the process at this level. All of the semantic analysis is hidden in the call to Statement.

Once the program is finished, the semantic analyzer has more work to do to clean things up. The first of these tasks is to tell the debugger that the subroutine has finished executing; this is done with yet another special debugger directive:

```
   if debug then                          {shut down the debugger}
      Gen0(d_exi);
```

When the program started, the semantic analyzer made a call to ~BASICSTARTUP to set up the environment; we now make a call to ~BASICSHUTDOWN, another library subroutine, to shut down the run-time environment and return to the program launcher.

```
   Gen0Name(m_jsl, @'~BASICSHUTDOWN');     {return to the launcher}
```

The symbol table still needs to be created for the debugger. To do this, we create the label called at the start of the program, then call GenDebugSymbols to scan the symbol table, creating the table used by the debugger.

```
   if debug then begin                    {generate the symbol table}
      Gen0Label(d_lab, symlabel);
      GenDebugSymbols;
      end; {if}
```

Another subroutine tells the code generator to place any string constants used in the program in the executable file.

```
   GenStrings;                            {generate any string constants}
```

Finally, the work is an an end, and we use the d_end directive to tell the code generator to finish off the code segment that contains the executable program. Another segment holds the global variables; the GenSymbols procedure creates this segment, using the same directives to start and end a segment that were put to use here. Finally, CheckLabels is called to perform the check I mentioned earlier, verifying that any line number used as a destination actually exists in the program.

```
Gen0(d_end);                            {flag the end of the program}
GenSymbols;                             {create the symbol segment}
CheckLabels;                            {check for undefined labels}
```

This is a lot to digest, but you've also seen an enormous amount of the work needed to create a program. All of the code used to create the interface with the PRIZM debugger, for example, is in this subroutine – Statement and the various subroutines it calls are not involved in the debug process at all. You've seen how line numbers are used to create destinations for branch points, and even how they are used (remember the JSR to create the symbol table?). You can probably write the semantic subroutines for the GOTO or GOSUB statement yourself, having seen what you just did. In short, while this is all a bit strange right now, you have seen a great deal of what is involved in semantic analysis.

When we looked at the parser, we looked at the Statement procedure; that's one that doesn't change when we add the semantic routines, so we'll skip straight to the statements themselves. The same statements we looked at with the parser will be covered here, although in a slightly different order.

The +S Flag

While a compiler is under development, I find it handy to have several kinds of diagnostic output to see what the semantic analysis and code generation subroutines are actually doing. In Integer BASIC, you can turn on this diagnostic output by compiling with the +S flag. The output is, for the most part, assembly language source code, so you can see the actual code generated for a program. By turning on the listing with the +L flag, you can even match the assembly language source code up fairly closely with the lines that created the code.

To see how this works, I've compiled a BASIC program that consists entirely of a single comment. I used the flags +D +S +L, so the compiler generated debug code, the diagnostic output, and the source listing. Here's what I got:

```
Integer BASIC 1.0
Copyright 1991, Byte Works, Inc.

  47 BASIC    START
   0          jsl   ~BASICSTARTUP
   4          SUB   "BASIC"
  10          FIL   "/WORK/T.BAS"
  16          jsr   L2
  19          LNM   1
  23 L3       ANOP
   1 100 REM This is a test...
  23          EXI
  25          jsl   ~BASICSHUTDOWN
  29 L2       ANOP
  29 L1       ANOP
  29          dc    i1'5',c'BASIC',i1'11',c'/WORK/T.BAS'
  47          END

0 errors found.
```

This lets you see very clearly what the semantic subroutines we just looked at actually accomplish in terms of the instructions and directives issued by the code generator. To see the actual object file, I use the DUMPOBJ utility with the flags +D -H. Dumping this file, I get:

```
DumpOBJ 1.1


00003C 000000 |          LONGA ON
00003C 000000 |          LONGI ON
00003C 000000 | BASIC    START
00003C 000000 |          JSL   ~BASICSTARTUP
000042 000004 |          COP   $03
000045 000006 |          DC    I4'(BASIC+$0000001D)'
000049 00000A |          COP   $06
00004C 00000C |          DC    I4'((BASIC+$0000001D)+$00000006)'
000052 000010 |          JSR   (BASIC+$0000001D)
000058 000013 |          COP   $00
00005B 000015 |          ORA   ($00,X)
00005D 000017 |          COP   $04
00005F 000019 |          JSL   ~BASICSHUTDOWN
000064 00001D |          ORA   $42
000067 00001F |          EOR   ($53,X)
000069 000021 |          EOR   #$0B43
00006C 000024 |          AND   >$524F57
000070 000028 |          PHK
000071 000029 |          AND   >$422E54
000075 00002D |          EOR   ($53,X)
000077 00002F |          END
```

The COP instructions and the strange junk that follow them are for the debugger. Reading between these lines, you can see that DUMPOBJ is another very powerful tool for looking at the semantic analyzer and code generator. If you plan to use DUMPOBJ, though, you might leave debug code turned off (by not using the +D flag) so you don't have to wade through the COP instructions.

If you ever start to get a little confused about what the semantic analyzer is doing, be sure and take a look at the results using +S and DUMPOBJ – it seems like a simple step, but it will clear up a lot of mysteries very quickly.

The GOTO Statement

Let's start our look at the statements with an easy one, the GOTO statement. The complete code for the GotoStatement procedure is:

```
begin {GotoStatement}
NextToken;
if token.kind = intconst then begin
   Gen0Label(m_jmp, StatementLabel(token.ival, false));
   NextToken;
   end {if}
else
   FlagError(9);
CheckEol;
end; {GotoStatement}
```

This is one of the simplest subroutines to handle; the only line we had to add to handle semantics is the code generator call that creates the JMP instruction. StatementLabel works just like it did in the main body of the compiler, and in fact, this call to the code generator is very similar to the JSR generated to handle the debugger symbol table.

The LET Statement

The semantic routines for the LET statement are just as simple as the parsing routines, but as with parsing, most of the work is hidden. In fact, there is only one new line, a call to Store:

```
begin {LetStatement}
LValue;
Match(eq, 3);
Expression([]);
Store;
CheckEol;
end; {LetStatement}
```

As with parsing, the real work is hidden away in the subroutines.

L-Values and the Store Procedure

With the semantic routines in place, the complete LValue procedure looks like this:

```
begin {LValue}
if token.kind = ident then begin
   id := token;                          {save the identifier}
   symbol := FindSymbol(token.name);
   lkind := symbol^.kind;                {record the type}
   NextToken;                            {skip the identifier}
   if token.kind = lparen then begin
      if symbol^.newSymbol then begin  {make sure the variable is declared}
         symbol^.newSymbol := false;
         symbol^.size := 10;
         end; {if}
      if (symbol^.kind = int) and (symbol^.size = 0) then
         FlagError(31);
      indexed := true;                   {find the index}
      NextToken;
      Expression([rparen]);
      Match(rparen, 24);
      if symbol^.kind = int then
         Gen0(m_asl)
      else
         Gen0(m_dec_a);
      Gen1Byte(m_sta_dir, index);
      end {if}
   else
      indexed := false;                  {not indexed}
   end {if}
else
   FlagError(27);
end; {LValue}
```

To understand how this procedure works, we need to stop for a moment and remember what it is used for. We are basically trying to figure out where to put a value. On the left side of the equal sign, which this subroutine us handling, we can have a simple variable or an array reference. If we are dealing with an array reference, the array subscript is a general expression, which can involve other variables, function calls, or even other subscripted variables. In a more complicated compiler, we might handle each of these cases as a separate situation, and if we did, the code generated by the compiler would be considerably better than what you will see Integer BASIC creating, but in this compiler, the goal is to keep things simple, so we will only handle two cases: either the variable is a simple variable, or it is a subscripted variable. In the case of a simple variable, we will just need to remember if it is a string or integer, and the name of the variable; for a subscripted variable, we will also need to calculate and remember the array index. After we take care of the l-value, we will be evaluating another expression, the one on the right-hand side of the equal sign, so we can't count on the registers being preserved; for that reason, when we calculate an array subscript, we'll need to save the value somewhere. The compiler will use several temporary variables for little bookkeeping operations like this one; they are in direct page, and are assigned fixed locations. This one is handled by the constant INDEX.

Actually, LValue is really just setting things up, and perhaps calculating and saving an array subscript. The real work of storing a value will be done later, by a procedure that is cleverly called

Store to keep the meaning clear. The process of computing and saving a value, then, starts with a call to LValue to set things up, then moves to a call to Expression to evaluate the expression that is to the right of the equal sign, and concludes with a call to Store to save this value in the location identified earlier by LValue. LValue and Store communicate with a set of global variables that are set by LValue and used later by Store.

With this game plan in mind, it is a little easier to see what the LValue procedure is doing. The first step is to record the name and type of the identifier:

```
id := token;                          {save the identifier}
symbol := FindSymbol(token.name);
lkind := symbol^.kind;                {record the type}
```

The variables ID and SYMBOL will be used later, after the expression on the right-hand side of the expression has been evaluated. Right after these values are saved, the parser checks for a left parenthesis; if one is found, we are dealing with a subscripted variable, and INDEXED is set to true. If no left parenthesis is found, we assume that we are dealing with a simple variable, and INDEXED is set to false. Like ID and SYMBOL, INDEXED will be used later by the Store procedure.

If the l-value is subscripted, we compute the subscript and save it in the direct page location INDEX:

```
Expression([rparen]);
Match(rparen, 24);
if symbol^.kind = int then
   Gen0(m_asl)
else
   Gen0(m_dec_a);
Gen1Byte(m_sta_dir, index);
```

To understand this sequence of code completely, you have to know one assumption. It turns out that things are much more efficient if we assume that the Expression procedure generates code that leaves the result of any integer expression in the accumulator, so after the call to Expression, we can just do an ASL to convert the index value into a displacement into the integer array (integers are two bytes long, so the ASL multiplies the index by two to convert from the index to a displacement). Strings are arrays of one byte characters, so we skip the ASL for strings, but the character with an index of 1 is zero bytes past the name of the string, so we have to do a DEC of the value, instead. Remember, integer arrays have an initial value with a subscript of zero, while the first character in the string has a subscript of one. All we did here is convert that difference to code. Finally, the displacement into the array is saved for later use by the Store procedure.

Along the way, the semantic analyzer takes care of one other chore. In BASIC it is legal to use an array or variable before it is declared. Back when we were setting lkind to the type of the variable (str for string, or int for integer), we made a call to the symbol table handler to look up the identifier:

```
symbol := FindSymbol(token.name);
```

Because of the nature of the BASIC language, FindSymbol is a very forgiving call. It checks the symbol table to see if an identifier already exists with the name given. If not, it simply creates one! The type of the symbol is implied by the name of the symbol itself: strings end with the character $, while integer variables don't. When FindSymbol creates a new symbol, it creates and fills in a symbol record:

```
      symbolPtr = ^symbolType;       {pointer to a symbol table entry}
      symbolType = record            {symbol table entry}
         left,right: symbolPtr;        {tree links}
         name: nameType;               {symbol name}
         kind: expressionTypes;        {type of the symbol}
         size: integer;                {array size (1 if not an array)}
         newSymbol: boolean;           {was this symbol just created?}
         end;
```

The name is filled in from the parameter passed to FindSymbol, while the type (labeled kind in this record to avoid a conflict with the Pascal reserved word TYPE) is filled in after looking at the name. We could get by without the kind variable, of course, but the compiler would actually be larger and slower; it's easier to access an integer than to continually look at the last character in the name to figure out the type of the variable. Left and right are also filled in; these are links used to maintain the symbol table as a binary tree. Size tells how big an array is, and defaults to 1, which is a non array variable. The important variable, from our standpoint, is newSymbol, which is set to true to indicate that the variable has just been defined. Any time FindSymbol is called, it will set newSymbol to false if the symbol is already in the symbol table, so we know that newSymbol will only be true if the symbol was just declared by the FindSymbol call in LValue.

If the parser then stumbles across a left parenthesis, it makes some changes to the symbol, marking it as an array and using the default length of 11 elements. It can tell if the symbol has already been declared as an array by checking newSymbol.

```
         if symbol^.newSymbol then begin   {make sure the variable}
            symbol^.newSymbol := false;    { is declared          }
            symbol^.size := 10;
            end; {if}
         if (symbol^.kind = int) and (symbol^.size = 0) then
            FlagError(31);
```

We'll look at how expressions are evaluated in a moment, but first let's look at Store. The job of the Store procedure is to take a value in the accumulator and store it in the location identified by LValue. The subroutine starts by making sure that if the result of the expression was a string, then so is the variable we're about to store the value in, and if the expression returned an integer, we are about to store the integer to an integer variable:

```
   begin {Store}
   if expressionKind <> lkind then         {check for type conflict}
      FlagError(32);
```

The chore of saving a value is quite different for arrays and simple variables. The Store procedure handles saving to an array first. While this is the most code you've seen generated at one place so far, it's really very straight-forward assembly language to handle tasks that are fairly easy to understand. While the length of the code may be a little scary at first, reading through it, you shouldn't have much trouble figuring out what is happening. It might help, though, to write down the assembly language that will be generated for each of the four possible types of assignments: either integer or string; and either indexed or not indexed. If you have trouble, try the +s option.

```
if indexed then begin
   if lkind = str then begin
      Gen0NameShift(m_pea, @id.name);   {handle indexed string}
      Gen0Name(m_pea, @id.name);
      Gen1Byte(m_lda_dir, index);
      Gen0(m_clc);
      Gen1Byte(m_adc_s, 1);
      Gen1Byte(m_sta_s, 1);
      Gen0Name(m_jsl, @'~ASSIGNSTRING');
      end {if}
   else begin
      Gen1Byte(m_ldx_dir, index);       {handle int array assignment}
      Gen0Name(m_sta_absx, @id.name);
      end; {else}
   end {if}
else begin
   if lkind = str then begin
      Gen0NameShift(m_pea, @id.name);   {handle unindexed string}
      Gen0Name(m_pea, @id.name);
      Gen0Name(m_jsl, @'~ASSIGNSTRING');
      end {if}
   else
      Gen0Name(m_sta_abs, @id.name);    {handle int assignment}
   end; {else}
end; {Store}
```

Expressions

Expression evaluation turns out to be very easy from a recursive descent parser. Let's take a look at PlusMinus to see how things work:

```
begin {PlusMinus}
MulOp;
while token.kind in [plus, minus] do begin
   operation := token.kind;
   if expressionKind = str then begin
      FlagError(25);
      expressionKind := int;
      end; {if}
   NextToken;
   Gen0(m_pha);
   MulOp;
   if expressionKind = str then begin
      FlagError(25);
      expressionKind := int;
      end; {if}
```

```
         if operation = plus then begin
            Gen0(m_clc);
            Gen1Byte(m_adc_s, 1);
            Gen0(m_plx);
            end {if}
         else begin
            Gen1Byte(m_sta_dir, tempDP);
            Gen0(m_pla);
            Gen0(m_sec);
            Gen1Byte(m_sbc_dir, tempDP);
            end; {else}
         end; {while}
      end; {PlusMinus}
```

As you saw earlier, the assumption made throughout the compiler is that the Expression procedure returns integer values in the accumulator. (For a string the address of the string itself is pushed onto the stack.) The type of the expression is placed in a global variable called expressionKind. This convention is used for values inside Expression, too, so as the semantic analyzer kicks in to handle an operation, it can check the type of the first value by looking at expressionKind, flagging an error if the value is a string, since Integer BASIC doesn't define + or - for strings.

After recording the kind of the operation and checking to make sure the first value is an integer, the parser will be making a second call to MulOp, and that call will of course return the value of the second number to add or subtract in the accumulator. To preserve the first value, the semantic analyzer generates a PHA right before the second call to MulOp. Upon return, the job is to do an add or subtract, so the semantic analyzer calls the code generator to generate the proper instructions.

If you look at the code that gets generated for a subtract, you will see a strange sequence, needed because the two values come in the wrong order. In compilers that generate high-quality code, this sort of problem is generally avoided by an intermediate step. In larger compilers, the semantic analyzer doesn't deal with machine language directly; instead, it uses some intermediate representation for the program; these intermediate representations are called intermediate code. The code generator can manipulate the intermediate code to get the values in the order it really wants them, as well as avoiding a lot of other inefficiencies you will see in the code generated by Integer BASIC. The message, I suppose, is that this problem with the subtraction code can be solved, but the method that is generally used is pretty complicated for a small compiler like this one. If you would like to find out more, some of the books listed at the end of this chapter will go into plenty of detail.

The rest of the expression evaluator is handled the same way as PlusMinus. In a few cases, like MulOp, the operations are complicated enough that the compiler generates calls to library subroutines. In Term, you will find some cases where strings are handled, as well as the function calls – these are all interesting, and you should take a moment to look at the complete source code for Expression, but there really isn't anything new, so I won't beat you over the head with page upon page of commentary when you can get more out of a quick look at the compiler source itself. Instead, we'll move on and look at some genuinely new issues raised by some of the other statements.

The IF Statement

Most of the statements in Integer BASIC amount to nothing more than a subroutine call, possibly with a few parameters, but the flow of control statements do offer some challenge. The IF statement is the simplest of the flow of control statements, so we will look at it first.

```
begin {IfStatement}
NextToken;                           {skip the if}
Expression([thensy]);                {evaluate the condition}
if expressionKind = str then begin
   FlagError(25);
   expressionKind := int;
   end; {if}
Gen0(m_tax);
Match(thensy, 15);                   {make sure the then is there}
lab := GetLabel;                     {get a true branch label}
if token.kind = intconst then begin
   Gen0Relative(m_beq, lab);         {branch if false}
   Gen0Label(m_jmp,                  {handle an "if exp then number"}
      StatementLabel(token.ival, false));
   Gen0Label(d_lab, lab);
   NextToken;
   CheckEol;
   end {if}
else begin
   Gen0Relative(m_bne, lab);         {branch if true}
   lab2 := GetLabel;                 {handle a statement after the then}
   Gen0Label(m_jmp, lab2);
   Gen0Label(d_lab, lab);
   Statement;
   Gen0Label(d_lab, lab2);
   end; {else}
end; {IfStatement}
```

The job of the IF statement is to evaluate a condition, then either branch to a location or execute a statement if the condition is true (non-zero). As you can see, the semantic analyzer handles this by calling Expression, then doing a TAX, which is a quick way to test to see of the value in the accumulator is zero or non-zero. Based on that result, the semantic analyzer branches around a JMP instruction; the JMP is used because we can't be sure that the destination label is less than 127 bytes away, so we can't just do a relative branch to the destination. The creation and use of labels is something you've seen before. The nice news is that this fairly complicated statement turns out to be very easy to implement!

The FOR Statement

By far the most complicated statement in the entire Integer BASIC language is the FOR statement. Most of the complexity is due to the fact that the FOR statement is hooked with a NEXT statement, and they don't necessarily appear at the same place.

Let's start by outlining what the FOR statement will need to do. Here's an example we can use to make the discussion a little more concrete:

```
100 FOR I = 1 TO J/2 STEP 2
140    <other stuff>
150 NEXT I
```

The original FOR statement has several separate tasks to perform. First, it must do the same thing as an assignment statement, setting the variable I to 1. Next, it needs to do another calculation, figuring out what J/2 is, and saving that value for later. Finally, the step size needs to be stored for later use.

In BASIC, the body of the FOR statement always executes at least once. This is different from languages like C or Pascal. In those languages, we would check the loop condition right away to see if I (just set to 1) is greater than J/2. In BASIC, though, we leave this check for the NEXT statement.

When the NEXT statement is found, the step size is added to the variable I. After making this addition, the program checks to see if I is less than or equal to J/2. If so, the program jumps back to the statement right after the FOR statement, executing the body of the loop again.

Using BASIC statements, we can expand the FOR loop out to something like this:

```
100 I = 1
110 STOP = J/2
120 STEPSIZE = 2
130    <other stuff>
140 I = I+STEPSIZE
150 IF I <= STOP THEN 130
```

The compiler creates essentially this same code sequence using assembly language. The stop value, J/2, is computed and saved in a work area. Of course, if no for loops have been found yet, the work area has to be created first.

```
begin {ForStatement}
if firstFor then begin          {if this is the first, create ~FORTEMPS}
   firstFor := false;
   forTemp := '~FORTEMPS';
   symbol := FindSymbol(forTemp);
   symbol^.size := maxFor-1;
   end; {if}
if numFor = maxFor then begin    {handle too many nested for statements}
   FlagError(16);
   while not (token.kind in [colon, eolnsy, eofsy]) do
      NextToken;
   end {if}
else begin
   numFor := numFor + 1;             {get a for array area}
   NextToken;                        {skip the for}
```

With the housekeeping out of the way, the initial assignment of a value to the loop variable is handled pretty much the same way a LET statement was handled. Since we know we're not handling an array, we can dispense with the fairly complicated use of the LValue and Store procedures, but the generated code is the same as it would have been for "LET I = expression." The name of the loop variable will be needed later, so it is saved in a record. It's important to keep the distinction between what is saved in the compiler for later use when the NEXT statement is compiled, and what is saved in memory by the program the compiler creates. The stop value, J/2, is something that has to be computed when the finished program runs ("at run time" in compiler jargon), while the name of the loop variable is something the compiler needs to keep track of for later semantic checks (so it's needed at "compile time").

```
if token.kind = ident then begin  {do the initial assignment}
   symbol := FindSymbol(token.name);
   forStuff[numFor].symbol := symbol;
   if (symbol^.kind <> int) or (symbol^.size <> 0) then
      FlagError(17);
   NextToken;
   end {if}
else
   FlagError(17);
Match(eq, 3);
Expression([tosy]);
Gen0Name(m_sta_abs, @symbol^.name);
```

The next thing the parser hits is the expression that gives the upper limit for the loop variable, so this is when the semantic analyzer actually computes the stop value and saves it in the ~FORTEMPS area that was created at the start of the program. Pay special attention to the way numFor is being used, both here and in the forStuff array, to track nested for statements.

```
Match(tosy, 18);                  {check for the to}
Expression([stepsy]);             {evaluate and save the stop value}
if expressionKind = str then begin
   FlagError(25);
   expressionKind := int;
   end; {if}
Gen1Name(m_sta_abs, @'~FORTEMPS', (numFor-1)*2);
```

The Integer BASIC compiler restricts the step size to a positive or negative integer constant. As you may recall from the discussion of the scanner, there isn't really any such thing as a negative constant, so there is a little fancy footwork to handle a leading minus sign. When we evaluate the loop condition, we'll also need to know if we are looping up (a positive step) or down (a negative step), so seeing the negative sign as a separate token turns out to be convenient.

```
         if token.kind = stepsy then begin {evaluate the step size}
            NextToken;
            negative := false;
            if token.kind = minus then begin
               negative := true;
               NextToken;
               end; {if}
            if token.kind = intconst then begin
               if negative then
                  forStuff[numFor].step := -token.ival
               else
                  forStuff[numFor].step := token.ival;
               NextToken;
               end {if}
            else
               FlagError(19);
            end {if}
         else
            forStuff[numFor].step := 1;
```

When we get to the NEXT statement, the program will need to jump back to the top of the FOR loop. The last thing we do in ForStatement is to create a label for NextStatement to jump back to.

```
         with forStuff[numFor] do begin    {allocate loop variables}
            top := GetLabel;
            Gen0Label(d_lab, top);
            end; {with}
         end; {else}
      CheckEol;                           {check for junk at the end of the line}
      end; {ForStatement}
```

The FOR statement is really just the first half of a complete FOR loop, so we need to look at the NEXT statement to see how things finish up. You might stop and think about what would happen if the compiler got to the end of the program without finding a NEXT statement, though. You've seen how ForStatement increments numFor, which starts at zero, to keep track of how many FOR statements have been nested; NextStatement will decrement the value when it finishes off the for loop. When the compiler finishes, it makes a quick check of for numFor, flagging an error if numFor is not zero, which could only happen if a NEXT was not found for some particular FOR.

There is also the possibility that a programmer might accidentally put in a NEXT statement with no corresponding FOR statement; that's the first thing NextStatement checks for.

```
      begin {NextStatement}
   1: if numFor = 0 then begin
         FlagError(20);
         while not (token.kind in [colon, eolnsy, eofsy]) do
            NextToken;
         end {if}
      else begin
```

In Integer BASIC, you must place the loop variable on the NEXT statement, so NextStatement checks to make sure an identifier is given, and also checks to be sure it matches the one in the most recent FOR statement.

```
   NextToken;                        {skip the next or comma}
   if token.kind = ident then begin  {check the loop variable}
      symbol := FindSymbol(token.name);
      if symbol <> forStuff[numFor].symbol then
         FlagError(21);
      NextToken;
      end {if}
   else
      FlagError(17);
```

Now that the compiler knows the program is correct (or has at least flagged an error if it isn't), chunking out the code to increment the loop variable by the step size and branching back to the top of the FOR loop if we need to is all that's left.

```
   with forStuff[numFor] do begin
      Gen0(m_clc);               {increment/decrement the loop variable}
      Gen0Name(m_lda_abs, @symbol^.name);
      Gen1(m_adc_imm, step);
      Gen0Name(m_sta_abs, @symbol^.name);
      if step < 0 then begin        {test loop condition}
         Gen0Name(m_lda_abs, @symbol^.name);
         Gen0(m_pha);
         Gen1Name(m_lda_abs, @'~FORTEMPS', (numFor-1)*2);
         end {if}
      else begin
         Gen1Name(m_lda_abs, @'~FORTEMPS', (numFor-1)*2);
         Gen0(m_pha);
         Gen0Name(m_lda_abs, @symbol^.name);
         end; {else}
      Gen0Name(m_jsl, @'~CMP_GE');
      Gen0(m_tax);
      exit := GetLabel;
      Gen0Relative(m_beq, exit);
      Gen0Label(m_jmp, top);        {loop}
      Gen0Label(d_lab, exit);       {exit loop label}
      end; {with}
   numFor := numFor-1;
   end; {else}
```

If for loops have been nested, BASIC lets you finish off all of them with a single NEXT statement, assuming that the loop variables are separated by commas. The last thing NextStatement does is to check for this possibility, looping if there is another loop variable.

```
if token.kind = comma then         {allow for multiple next vars}
   goto 1;
CheckEol;                        {check for junk at the end of the line}
end; {NextStatement}
```

Looking back, you might be saying to yourself that *interpreted* BASICs can handle an expression as the step size, figuring out at run time whether you are going up or down. That's true. Take a moment and write the assembly language instructions this compiler creates to handle a for loop, then write the assembly language instructions that would be needed if you didn't know in advance if the step size was positive or negative. If you actually do this, you will see that it takes a lot longer to handle a FOR loop if you don't know the step size, and that's why this Integer BASIC compiler places a restriction on the step size, forcing it to be a constant.

There are a lot of solutions to this dilemma. One of the best is to have the compiler actually *look* at the step size. If it is a constant, the compiler can create the fairly clean code you see in Integer BASIC. If the step size is an expression, the compiler could create the more complicated, longer, and slower code needed to handle a non-constant step size. This is the sort of extra step a good commercial quality compiler would make, but which I skipped in Integer BASIC to keep things simple.

At this point, you probably have a lot of questions about how various statements are handled. How, for example, is something as messy as a PRINT statement done? The answer is that the PRINT statement isn't messy at all; it's a very straight-forward application of the same ideas you are already getting bored with while looking at these other statements. It only seems hard before you see how easy it really is – so bring up PARSER.PAS from the disk, and take a look!

Other Uses for Semantic Analyzers

What would happen if, instead of calling the code generator, the semantic analyzer just did what it was supposed to do? For example, when it is handling a PRINT statement and gets to the point where it has a string to print, the semantic analyzer tells the code generator to create the machine code to print a string. What would happen if the semantic analyzer just printed the string, instead? The answer is that you would have an interpreter!

Let's take a look at AppleSoft BASIC to see how this could work. In AppleSoft, the scanner is actually built into the line editor that you use when you type in a program. It "tokenizes" the input – which means that it converts the text into a shorter form of the same thing, a stream of bytes that represent the tokens our own scanner returned. When you run a program, AppleSoft kicks in the parser, which scans the token list. The semantic analyzer, built right in, takes action right away instead of creating code.

Basically, then, if you ditch the code generator and take action right away, you end up with an interpreter instead of a compiler. There are a lot of uses for interpreters besides just running BASIC programs, though. Interpreters can be used as very sophisticated data base query languages. Interpreters have found homes in many classy programs, like expensive CAD programs and HyperCard. A spread sheet uses these same ideas to calculate equations. Adventure games like Zork use an interpreter to move your character around, drop things, and kill monsters. In fact, almost any program that takes some action based on text input uses an interpreter of one form or another!

The Code Generator

In a high quality optimizing compiler, the code generator is easily the most challenging part of the compiler. In fact, in ORCA/Pascal and ORCA/C, the code generator accounts for about half the size of the compiler. In Integer BASIC, the code generator is a much simpler affair; in fact, it amounts to nothing more than taking the machine language instructions passed when the semantic analyzer makes a code generator call and stuffing these into an output file.

Code Generation

We'll start off by looking at Get0, the subroutine called by the semantic analyzer when it wants to write an instruction or directive that has no operand, like PHA.

One of the jobs of the code generator is to handle the +S option we looked at as a tool to see what code the compiler is generating, and that's the first thing Gen0 does. The global variable listSymbols is true if the +s flag has been used.

```
begin {Gen0}
if listSymbols then begin               {list the instruction}
   write(codelength:4, ' ');
   ListOp(op);
   writeln;
   end; {if}
```

There are two directives that have no parameters, d_end to end a segment, and d_exi to tell the debugger we're finished with a subroutine; you saw both of these when we looked at the semantic analyzer's version of the Compile subroutine. We'll look at d_exi first; here, the code generator creates a COP 4 instruction, writing two bytes to the output file and updating the codeLength variable.

```
if op = d_end then begin
   name := concat(keepFile, '.root');   {end the segment}
   EndSegment(@name);
   end {if}
else if op = d_exi then begin           {generate the end cop instruction}
   CPut1(m_cop);
   CPut1(4);
   codeLength := codeLength+2;
   end {else if}
```

CPutl is a procedure that places a byte in the output file. In this case, the byte must be a constant byte for the finished executable program. There's an entire family of different subroutines like CPutl in the code generator, each used to write a different kind of information to the OBJ file the compiler is creating. These include subroutines like PutL, which places a single byte in the OBJ file; Put2, which writes a two-byte integer to the output file; and PutName, which writes a relocatable expression involving a named label to the output file. These subroutines are collected at the top of CodeGen.Pas, and they all ultimately end up calling PutL to stuff bytes into a buffer in memory.

There is a difference between the number of bytes that have been written to the object file that the linker will read and the number of bytes that will be in the executable program, so throughout the code generator, when a value is written that will result in a byte in the finished program, the code generator updates codeLength. This value is used when the code generator comes across a label, when codeLength tells the code generator how far into the program the label is, in bytes. The code generator has just created a two-byte instruction, a COP instruction followed by a one byte operand. To keep the rest of the code generator up to date on where we are in the file, codeLength is incremented by 2.

Once the segment is finished, and it is time to write the file to disk, the semantic analyzer calls Gen0 with a d_end directive. Gen0 creates a file name by appending .root to the output file name, then calls EndSegment. The EndSegment procedure fills in a few values expected in the header of an object file, like the length of the segment, and then writes the segment buffer that has been created by all of the calls to PutL to the disk using FastFile calls. There are some details about how the +m flag is handled, but these details really aren't important to understand the way the output file is basically created.

After the directives, something as simple as a PHA is pretty easy to understand – a call is made to CPutl to record the byte, and codeLength is updated. It might seem strange to write the value of m_pha to the output file, but the code generator makes things easy for itself by declaring a series of constants like m_pha, and each of these constants corresponds exactly to the numeric value of the machine code instruction.

```
else begin
   CPut1(op);                          {write the opcode}
   codeLength := codeLength+1;         {update the length of the code}
   end; {else}
end; {Gen0}
```

With these ideas in mind, we can move on to a more complicated instruction. Many of the machine language instructions follow the operation code with a single, constant byte. A great example is this call from the parser, which is generating a store instruction, storing the value in a direct page location:

```
   Gen1Byte(m_sta_dir, wptr);
```

The Gen1Byte subroutine that handles this instruction looks like this:

```
procedure Gen1Byte {op, opnd: integer};

{ Generate a one byte constant operand instruction              }
{                                                               }
{ Parameters:                                                   }
{    op - instruction to generate                               }
{    opnd - operand                                             }

begin {Gen1Byte}
if listSymbols then begin              {list the instruction}
   write(codelength:4, ' ');
   ListOp(op);
   write(opnd:1);
   if op in [m_adc_s, m_and_s, m_ora_s, m_sta_s] then
      write(',S')
   else if op = m_lda_indl then
      write(']');
   writeln;
   end; {if}
CPut1(op);                             {write the opcode}
CPut1(opnd);                           {write the operand}
codeLength := codeLength+2;            {update the length of the code}
end; {Gen1Byte}
```

As you can see, it's really pretty simple. As with Gen0, the first thing that the subroutine does is handle writing the diagnostic output if the +S flag has been used. The part of the subroutine that actually does the work is very short, simply writing the operation code and operand, then updating codeLength. Gen1 does pretty much the same thing, but writes a constant word (two bytes) instead of a single byte; it is used with instructions like LDA #0, which need two bytes for the operand.

Variables are created using the equivalent of assembly language DS and ANOP instructions, and are referred to by name. There are several subroutines that handle these subroutines. Gen1Name, shown below, is a typical example.

```
procedure Gen1Name {op: integer; name: namePtr; opnd: integer};

{ Generate an instruction with a named label and offset         }
{                                                               }
{ Parameters:                                                   }
{    op - instruction to generate                               }
{    name - pointer to the named operand                        }
{    opn1 - numeric offset                                      }

begin {Gen1Name}
if listSymbols then begin              {list the instruction}
   write(codelength:4, ' ');
   ListOp(op);
   writeln(name^, '+', opnd:1);
   end; {if}
CPut1(op);                             {write the opcode}
PutName(name, opnd, 2, 0);             {write the name}
codeLength := codeLength+3;            {update the length of the code}
end; {Gen1Name}
```

Once again, the main portion of the subroutine is very simple, writing an operation code and the operand, then updating the codeLength variable. This time, PutName is called to write an expression to the object file, but the idea is no different than writing any other operand. This operand just happens to use a label name, so a different record has to be written to the object file. Generating this record is simple, although it involves several lines of code, and is used from more than one location, so it gets collected in a subroutine. There's nothing exciting there, but if you would like to look at the details you can find PutName in the file CodeGen.Pas.

Labels

While most of the code generator is pretty straight forward, there is one topic that deserves close attention, and that is how labels are handled. The problem is that we want to create a one-pass compiler, which means we only want to go through the code once, but a label can refer to a statement further along in the program, so we don't always know where we are going to branch to when the label is used.

Integer BASIC uses a popular method to handle this problem. When a label is used, the code generator keeps track of where the use occurred, as well as which label is being used. When the label is finally defined, so that a location is known, the code generator goes back and patches the various locations where the label was used, filling in the locations in the object file with the proper value. Of course, if a label is used after it has been defined, the code generator already knows where the label is located, and fills in the correct value right away.

We'll step through the various code generation subroutines as they are used by the semantic analyzer to see how these ideas are actually implemented.

When the semantic analyzer needs a new label, it calls GetLabel. GetLabel returns a label number that is actually used by the code generator as an index into an array; this array, along with the type and constant declarations that show how it is constructed, looks like this:

```
const
   labelSize = 1000;                    {max # of local labels}

type
   labelRefPtr = ^labelRefRecord;       {ptr to a label reference}
   labelRefRecord = record              {label reference}
      next: labelRefPtr;                { next entry}
      disp: 0..segmentSize;           { disp to label reference in segment}
      end;
   labelRecord = record                 {label entry}
      defined: boolean;                 { is the label defined?}
      disp: 0..segmentSize;             { disp to label location}
      refs: labelRefPtr;                { unresolved label list}
      end;

var
   labels: array[1..labelSize] of labelRecord; {label table}
   nextLabel: 0..labelSize;             {next label to allocate}
```

Looking at these declarations, you can see how the code generator will implement the ideas we just talked about. LabelSize is just a constant, setting an upper limit on the number of labels the code generator can deal with; you can change it if you like. The variable labels is an array with labelSize different spaces for labels, each of which is a record. The label record, cleverly called labelRecord so dummies like me won't forget what it's for, has three pieces of information: defined is a boolean variable; it tells us if the label has, in fact, been found in the program; disp is the location of the label, which is just the value of codeSize when the label is defined, or looked at another way, the number of bytes that are in the program before we get to the label; and refs is a linked list of records. The refs linked list is how forward references are found. If the code generator is asked to use a label, and the defined variable is currently false, then the code generator doesn't know where the label is, yet. In that case, it creates a labelRefRecord, filling in the location in the object file where the reference is located. When the label is finally found, the code generator will traipse through this list, filling in the proper value in the object file.

The label array is manipulated with a series of subroutine calls; the first is GetLabel, which reserves a space in the nextLabel array, returning the index into the array as the label number. Defined is set to false, since we don't know where the label is yet, and refs is set to nil, since there are no records in the linked list of forward references. A check to make sure we don't overflow the array rounds out GetLabel.

```
function GetLabel: integer;

{ Allocate a new numbered label                                 }

begin {GetLabel}
if nextLabel = labelSize then           {find and fill in a label record}
   FlagError(29)
else begin
   nextLabel := nextLabel+1;
   with labels[nextLabel] do begin
      defined := false;
      refs := nil;
      end; {with}
   end; {else}
GetLabel := nextLabel;                  {return the label number}
end; {GetLabel}
```

When the semantic analyzer wants to define or use a label, it does so by calling one of the code generator subroutines, using either a 65816 machine language instruction or the d_lab directive. The d_lab directive is used when the label is actually defined. You can think of it like writing an ANOP directive with a label attached in a 65816 assembly language program. The subroutine that handled all of this is Gen0Label:

```
procedure Gen0Label {op,lab: integer};

{ Generate an instruction with a numbered label                 }
{                                                               }
{ Parameters:                                                   }
{    op - instruction to generate                               }
{    label - label number                                       }

begin {Gen0Label}
if listSymbols then begin               {list the instruction}
   if op = d_lab then begin
      write(codelength:4, ' ');
      write('L', lab:1);
      if lab < 1000 then
         write(' ');
      if lab < 100 then
         write(' ');
      if lab < 10 then
         write(' ');
      write('   ');
      ListOp(op);
      writeln;
      end {if}
   else begin
      write(codelength:4, ' ');
      ListOp(op);
      writeln('L', lab:1);
      end; {else}
   end; {if}
if op = d_lab then
   DefineLabel(lab)                     {define a new label}
```

```
else begin
   CPut1(op);                            {write the opcode}
   ReferenceLabel(lab, 0, false, 2);     {reference a label}
   codeLength := codeLength+3;           {update the length of the code}
   end; {else}
end; {Gen0Label}
```

As with the other subroutines we've looked at that are called by the semantic analyzer to generate code, this subroutine spends most of it's code handling the +S flag, printing pretty diagnostic output. For the real work, it calls DefineLabel to define a label, and ReferenceLabel to use a label as the operand of an instruction. Those subroutines are where the work is done, so let's look at them more closely, starting with ReferenceLabel:

```
procedure ReferenceLabel (lab, poffset: integer; prelative: boolean;
   psize: integer);

{ Create a reference to a label                                   }
{                                                                 }
{ Parameters:                                                     }
{    lab - label number                                           }
{    poffset - offset from the start of the label                 }
{    prelative - is the reference to a relative label?            }
{    psize - size of the label field, in bytes                    }
{                                                                 }
{ Variables:                                                      }
{    disp - location of the label reference in the OBJ file       }

var
   i: integer;                           {loop variable}
   ldisp: integer;                       {local copy of disp}
   lptr: labelRefPtr;                    {work pointer}

begin {ReferenceLabel}
if prelative then begin
   Put1($EE);                            {write the expression opcode}
   Put1(psize);                          {write the expression size}
   Put4(psize);                       {write the disp to the end of the exp}
   end {if}
else begin
   Put1($EB);                            {write the expression opcode}
   Put1(psize);                          {write the expression size}
   end; {else}
Put1($87);                               {save space & loc of label value}
ldisp := disp;
Put4(0);
if poffset <> 0 then begin               {add in the offset}
   Put1($81);
   Put4(poffset);
   Put1(1);
   end; {if}
Put1(0);                                 {end of expression}
with labels[lab] do                      {create the label reference}
   if defined then
      MakeReference(ldisp, disp)
   else begin
      new(lptr);
      with lptr^ do begin
         next := refs;
         disp := ldisp;
         end; {with}
      refs := lptr;
      end; {else}
end; {ReferenceLabel}
```

There are really two distinct things going on in this subroutine. One the one hand, ReferenceLabel must write a relocatable label reference to the object file. This isn't a particularly difficult thing to do, but it does take several lines of code, and this obscures what we're really after. The second part is what interests us here, and that's how ReferenceLabel handles the creation of the forward reference list, or how it uses a label value if it is known. Since we haven't really looked at the output of a complex object record yet, though, we'll stop and take a look at what this subroutine is actually doing to create the object record.

One of the types of records that actually appear in the object file is called an expression. In an object file, an expression is used when some bytes can't be calculated exactly by the compiler. For example, a LDA #4 instruction can be resolved right away as the three constant bytes $A9 $04 $00, but the instruction LDA PIXEL can't, since the compiler has no idea where the variable PIXEL will ultimately be located in memory. The compiler handles this situation by writing a constant byte for the LDA operation code, followed by an expression that tells the linker that it should find a variable called PIXEL somewhere and fill in the appropriate address. The linker, in turn, will generally pass on the same sort of request to the loader, which is the part of the operating system that actually loads a program from disk and puts it in memory. The reason for all of this is that you really can't tell where any particular part of the program will actually be until the loader actually places it in memory.

Let's assume that the compiler is creating an expression to tell the linker that it will be doing a JMP instruction to a location that is $1234 bytes away from the start of the program, which is exactly the sort of situation ReferenceLabel is designed to handle. The record that appears in the object file will look like this:

bytes	meaning
$EB	This byte tells the linker that the record it is about to process is an expression.
$02	The operand for a JMP instruction uses 2 bytes of memory; this byte tells the linker that the result of the expression must be stuffed into two bytes. A JSL would use a 3 here to get a 3 byte (long) address, while a DC statement might use a 1 or 4.
$87	Starting with this byte, we are actually giving the various values and operations that make up the expression. The expression is a reverse polish expression, just like HP calculators, where the values are given, followed by the operations. The $87 is a code that tells the linker to expect an offset from the start of the current code segment. The next value...
$00001234	...is the four-byte offset from the start of the segment. Here the value is shown most significant byte first, the way we generally write numbers, but of course it is stored in the file with the bytes reversed, the way the 65816 uses numbers.

$81	It is possible to create an instruction that will branch a few bytes past a label, although this compiler will never actually do that. The compiler will create references to a few bytes past the label to get at string constants, though. To handle that, you would add the bytes $81, which is a byte that tells the linker to look for...
$00000004	... a four byte constant value, to be added to the address of the label itself, and...
$01	... a 1, which tells the linker to add the two previous values. If you don't need to add a constant offset to the label's location, these six bytes would be omitted.
$00	The end of the expression is marked with a 0.

There are several kinds of expressions. The kind we've looked at here is used for absolute addresses, like you would find on a JMP instruction. Another kind used by the compiler is used for relative expressions, like the operand for the BNE instruction. From the viewpoint of the compiler, the big difference between an absolute addressing mode instruction and a relative branch is the code that marks the start of the expression. For an absolute address, the expression starts with a $EB, while a relative branch operand starts off with a $EE. The linker can handle some other forms of relative expressions, so you also have to put the length of the instruction in twice, once as a one-byte value marking the number of bytes used by the expression, and once as a four-byte value telling how many bytes past the start of the expression to go to compute the displacement. The purpose for this extra field is so the linker can handle relative addresses as used by the 65816, where the address is relative to the location counter after reading the bytes in the relative address, or the way relative addresses might be used on other computers, which offset from the start of the value. In any case, you just have to remember to put the length in twice, once as a single byte and again as a four-byte field. Right at the top of the procedure, you can see ReferenceLabel doing just that, picking the kind of the expression based on a boolean parameter called prelative.

Of course, our main interest in ReferenceLabel is how it actually handles the value of the label, not the format used to stuff an expression into an object file. Looking at ReferenceLabel, you can see that when it writes the expression, it actually writes a value of 0 for the displacement from the start of the segment, instead of the $1234 we used in our example. After the expression is written to the object file, the label is handled by this code:

```
with labels[lab] do                     {create the label reference}
   if defined then
      MakeReference(ldisp, disp)
   else begin
      new(lptr);
      with lptr^ do begin
         next := refs;
         disp := ldisp;
         end; {with}
      refs := lptr;
      end; {else}
```

Here we see that if the label is defined, ReferenceLabel will call a subroutine called MakeReference, telling that subroutine the location in the object file where the label was used (ldisp) and the number of bytes that appear in the program before the label itself (disp, which is pulled from the record labels[lab]). This subroutine fills in the correct spot in the object file. We won't look at it in detail, since it is just stuffing the value of disp into the object file, ldisp bytes past the start of the object file.

If the label has not been defined, ReferenceLabel needs to create a record in the forward reference list to keep track of the location of the use of the label. The space for this record is reserved with a call to new, ldisp is saved in the record, and the record is added to the linked list of forward references.

When the semantic analyzer calls the code generator to declare the label itself, it makes a call to Gen0Label with a d_lab directive. As we saw earlier, this calls DefineLabel. Here's the DefineLabel procedure:

```
procedure DefineLabel (lab: integer);

{ Define a label                                                }
{                                                               }
{ Parameters:                                                   }
{    lab - label number to define                               }
{                                                               }
{ Variables:                                                    }
{    codeLength - disp into the code segment                    }

var
   lptr: labelRefPtr;                  {work pointer}

begin {DefineLabel}
with labels[lab] do begin
   disp := codeLength;                 {define the label}
   defined := true;
   while refs <> nil do begin          {resolve existing references}
      lptr := refs;
      refs := lptr^.next;
      with lptr^ do
         MakeReference(disp, codeLength);
      dispose(lptr);
      end; {while}
   end; {with}
end; {DefineLabel}
```

After ReferenceLabel, this procedure must seem mercifully short, yet there's more code here to handle the label table itself than there was in ReferenceLabel, which spent most of its time creating the expression in the object file. DefineLabel starts out by filling in the disp field, then marks the label as defined. Next, it checks to see if any forward references have been made, processing any entries in the forward reference list by again calling MakeReference, just like ReferenceLabel did when it wanted to fill in the actual value of the label. Any records that are no longer needed are dumped, so the memory can be reused later if any other forward references are made.

The rest of the code generator is a lot more of what you've just seen: workhorse subroutines to stuff bytes into the object file, more GenXXX procedures that the semantic analyzer can call to generate various forms of instructions, and a few housekeeping subroutines to set up the various values used by the code generator and do the actual work of writing the file. There's a lot in the subroutines, but nothing is new, so I'll leave it to you to skim through CodeGen.Pas to see how the other subroutines are set up.

Unusual Uses for Compilers

Back when we started this chapter, I said that a compiler is really a translator that reads on language and writes another. This compiler reads Integer BASIC and writes machine language. Maybe you can start to see now just how true this is. Thinking of a compiler as a translator, you can start to see a lot of really nifty applications for what you have learned so far.

For example, let's say you want to port Integer BASIC programs to the Macintosh. Why not write C instead of machine language? Another way to use the same idea would be to write a FORTRAN compiler for the Apple IIGS that wrote C instead of machine language. It would take a lot less time to write a compiler that wrote C programs as opposed to one that wrote machine language programs! Another example is the language RATFOR, which, as far as I know, never even existed as a machine language compiler, but it was very popular back in the mid seventies. RATFOR, or Rational FORTRAN (an oxymoron if there ever was one), is a structured version of FORTRAN that was implemented as a compiler that took RATFOR input and created FORTRAN programs. For still another example, Apple is currently selling a C++ compiler for the Macintosh that does something very similar, taking C++ as input and creating C programs as output. In fact, C's preprocessor is really a compiler, too, taking preprocessor input and creating straight C as output!

Or how about a compiler that takes Spanish as input and writes English as output? There are certainly some challenges there, but you could probably write a program that would do a good enough job that you could understand most of what it produced with no trouble.

Testing

At this point, a lot of people would say, "Hey, it compiled Hello, World - it's done."

Right.

Until a compiler is tested, it isn't finished. Testing a compiler is every bit as important as writing it in the first place. Compilers are large and complex programs, and it is very easy to break something in a compiler without realizing it as you add a new feature or correct a bug. As a result, it is very important to have some systematic way of checking the compiler to make sure it does what it is supposed to do. Fortunately, compilers can be tested almost mechanically. True, developing the tests is a challenging, time-consuming process, but compilers are tested by compiling a set of programs in a test suit, so once the test suite has been created, it is pretty easy to simply rerun all of the programs to make sure the compiler is still alive and well.

Ideally, a test suite for a compiler should do several things:

- The test suite should exercise every single line in the compiler.
- The test suite should verify that every feature of the language is in the compiler, and that each feature works.
- Incorrect programs should always create an error message, and the error message should be meaningful. The test suite should verify that these conditions hold.
- Real programs are often very long. The test suite should include some long programs that exercise the compiler in realistic ways.
- While two compilers may both create programs that work, the quality of the translation from the source language to machine language can be very different. A good code generator can generate programs that are twice as fast as those created by a poor code generator. In some cases, in actual compilers, I have seen reports of up to a 96% improvement in the code – the better code ran 20 times faster than the poor code! The

test suite should include tests that give some idea of the quality of the program created by the compiler.

Our test suite is pretty modest, just as the Integer BASIC compiler itself is pretty modest. It includes a group of test programs designed to do the second step, making sure that the Integer BASIC compiler handles all of the features this manual says it will. These tests are divided again into two subcategories. The first includes all of those programs that can tell you if they worked by printing a simple statement saying the compiler either flunked the test or passed. Here's the first of them, C001.BAS, which makes sure the = comparison operation works. You can find this program, along with the other tests, in the Tests folder.

```
100 rem
110 rem Make sure the = operator works
120 rem
130 sum = 0
140 if 3 = 3 then sum = sum + 1
150 if -4 = -4 then sum = sum + 2
160 if 0 = 0 then sum = sum + 4
170 if (0 = 0) = 1 then sum = sum + 8
180 if (0 = 1) = 0 then sum = sum + 16
190 if -4 = 4 then sum = sum + 32
200 if 0 = 1000 then sum = sum + 64
210 if sum = 31 then print "Passed C001"
220 if sum # 31 then print "Failed C001"
```

Some of the compiler's features can't be tested properly with a simple pass/fail test like this one. For example, how can you do a pass/fail test to see if the PRINT statement works? In cases like this, the person running the test has to look at the output, or maybe even type some input, to make sure the compiler is working properly. Here's an example of this kind of test, SC001, which tests the PRINT statement:

```
100 rem
110 rem  Test the print statement
120 rem
130 print "All of the following lines should be exact duplicates."
140 print
150 print "Hello, world."
160 print "Hello, ";
170 print "world."
180 print
190 print ,12345,"A",-3
200 print ,12;345,"A              -3"
```

While you won't find them at your local check out stand, there are some great books that deal with software testing, and compiler testing in particular. My personal favorite is listed at the end of this chapter. If you are serious about writing a compiler that will be used by anyone else, you owe it to them to write a test suite so you can be sure the compiler works. I would recommend reading at least one good book on testing before you start to write the next best selling compiler for the Apple IIGS!

Standards

There's a big difference between designing a new language and implementing one that already exists. The computer world would be a much nicer place to live if certain people at UCSD, Apple Computer, Borland, and MicroSoft would get that simple fact through their head. When you use a language like Pascal, for example, you have every right to expect that the FOR statement will work the same way in each and every compiler, yet compilers from the companies mentioned have errors in the way for statements are implemented in Pascal. Some folks may tout C as an alternative, since it is supposedly a portable programming language, but a careful comparison of C compilers will show the same sort of problems.

When a language is designed, one of the parts of the design that is critical is a very precise description of the language. In the early days when folks were coming up with C, FORTRAN, and BASIC, the design of the language went something like this: "C is the language compiled by the compiler I just wrote." Well, fortunately, times have changed, and for most major languages, including C, Pascal, COBOL, FORTRAN, and Ada, to name a few, language standards exist. These language standards are developed by a world-class group of experts on the particular languages, working under the auspices of either ANSI (American National Standards Institute) or ISO (International Organization for Standardization). These standards are so well recognized that for some of the languages, like COBOL, FORTRAN and Ada, it is tough to get anyone to buy a compiler that doesn't conform to the standard, and for C and Pascal, the standards are so well known that companies selling compilers have to at least acknowledge that the standard exists, and list the places where their compiler falls short (even if they never quite list *all* of the places!).

If you intend to write a compiler, you should start by finding out what the standard for the language you pick says. Unless you have an incredibly good reason (I can't think of one, but maybe you think you have one), you should implement the language exactly like the standard tells you to. All of the standards have some way to extend the language, so if you really need the language to do something extra, like handle the Apple IIGS toolbox, you can do that. But if the standard says a for loop variable should be protected, with no possible way of changing it inside the for loop, that's the way you should implement the for loop. The ISO and ANSI Pascal standards say exactly that, by the way. If you don't like this restriction, pick a different language to implement, or create your own new language based on an existing one – and pick a new name for the language while you're at it, because it isn't Pascal anymore!

If it seems like I'm a bit of a fanatic on this point – well, I am. I really think that there is a basic, fundamental difference between designing and implementing a language. Both are great fun, but if you are trying to create a compiler that others will use, they have every right to expect that the compiler will match accepted standards. Before you buy a compiler, my personal recommendation is to ask very pointed questions of the sales people, and perhaps make a call to the technical service department of the company that wrote the compiler, to make darn sure that the compiler follows the accepted standards. If it doesn't, don't buy it! And, while you're at it, make it very plain to the sales person and the manufacturer that you aren't buying the compiler, and exactly why you are rejecting the product. If enough people vote with their wallets on this issue, you stand a better chance in a few years of picking up a book from the local bookstore that shows a program in, say, C, and actually having it run on whatever machine you are using without any changes. And, if you write a compiler, you really have absolutely no excuse for ignoring the standards, and even less excuse to release the compiler using a name like C or Pascal, which should mean something specific!

Getting a Language Number

Compilers that run under the ORCA or APW environment have unique language numbers, used by the shell to decide which compiler should be used to compile a particular program. If you create a new language, or even create a new implementation of a language someone else has already released, you need to get a language number for your compiler. Language numbers are issued by Apple Computer. To reserve a language number for yourself, write to:

Apple II Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3A
Cupertino, CA 95014
ATTN: APW Language Number Administration

Apple Computer will want to see a finished compiler at some point. I would suggest waiting until your compiler is close to finished before asking for an official language number.

Some Good Books to Read

There are hundreds of books and magazine articles that talk about writing compilers. From these hundreds, here's a few of my favorites.

The BYTE Book of Pascal
Blaise Liffick
BYTE/McGraw Hill, 1979

This book reprints a wonderful series of three articles that originally appeared in Byte Magazine in 1978. These articles show a working subset of Pascal, written in BASIC of all things. Talk about a reversal of fortunes! The compiler generates p-code, which is interpreted by an 8080 microprocessor. The articles describe the structure of a compiler, how stack frames are implemented to handle nested procedures and functions in languages like Pascal and Ada, and how p-code can be used to create a compiler quickly, in a minimum amount of space, and in such a way that the compiler can be moved to other machines in a hurry.

If you are interested in writing a small compiler of your own, or want to follow up on the things you learned in this book, I can't recommend this series of articles highly enough.

Principles of Compiler Design
Alfred V. Aho, Jeffrey D. Ullman
Addison-Wesley, 1979

If you ask anyone in the compiler field about the dragon book, they will know exactly what you are talking about. This is the classic textbook on compiler design, and, I think, still the best. It covers all phases of language design and compiler construction, including how to create an optimizing compiler that is better than anything currently on the Apple IIGS, and better than anything I've seen on the Macintosh or IBM PC, although I admit that I don't know the compilers on those machines as well as those on the Apple IIGS.

The nickname for the book, incidentally, comes from the Don Quixote parody on the cover, where a knight armed with an LALR lance, the shield of syntax directed translation, and the armor of data flow analysis is attacking the dragon, labeled "complexity of compiler design." On the flip side, things are shown in their proper perspective, with the good old Don armed with merely a pen, attacking a windmill. In tribute, in my window is a beautiful stained glass picture showing a dragon picking his teeth with a lance, over the body of a slain knight, with the caption "Sometimes the Dragon Wins." I guess there's two sides to every story...

There is a second addition of this book, and I have that one, too. Get the first edition, if you can. The second one is great, too, but the first edition is shorter, simpler, and clearer.

Brinch Hansen on Pascal Compilers
Per Brinch Hansen
Printice-Hall, 1985

This book shows you the insides of a significant subset of Pascal in a real, working compiler written by one of the greats in the field. While the book doesn't try to cover all of the various compilation techniques you will find in books like the dragon book, it does something they rarely do: you get to see the insides of a real, working compiler, which is implemented and explained by a master in the field. The book doesn't just cover the compiler, either – testing is given it's proper place as a part of the development cycle.

Pascal – The Language and its Implementation
D.W. Barron
John Wiley & Sons, 1981

This collection of essays shows that real compilers have warts, and that the most obvious way to handle something in a compiler isn't always the best. It's great reading for anyone who is actually implementing a full-featured compiler, especially one for Pascal.

Pascal Compiler Validation
Brian A. Wichmann, Z. J. Ciechanowicz
John Wiley & Sons, 1983

Here's a great book for anyone interested in what goes into testing a compiler, how the standards institutes pick standard test suites, and what some of the top folks in the Pascal field thought about the process and results of developing a test suite for ISO Pascal.

Software Manual for the Elementary Functions
William J. Cody, Jr., William Waite
Printice-Hall, 1980

One of the dirty little jobs many language implementors are faced with is implementing a floating-point package. Even if you're on a machine with a floating-point processor, like an IBM PC equipped with am 8086, you will still have to implement some of the more complex functions, like sine and cosine, at least partially in software. This book tells you how it's done, and also gives some great ways to test floating-point subroutines – ways that will break simpler ones, like AppleSoft BASIC.

Chapter 4
Customizing Integer BASIC

Make it Yours!

If you ask five different programmers to design a program, you are going to get five different programs. Interestingly enough, though, they are all right, in a sense. Each programmer comes up with a design that meets his own goals for the program. Sometimes when a programmer looks at the efforts of someone else writing a similar program, they like something they see, but it is pretty rare for a programmer to prefer the efforts of another programmer wholesale to their own effort. The reason is really pretty simple, after all – we're all different, with different backgrounds and tastes, so we each need something different from a program.

Well, compilers are no different. Maybe you would like to write a few real programs in Integer BASIC, but it drives you nuts to use line numbers. Maybe the compiler is almost what you need as the basis for a game design program, and with a few additions, you could use it to write your games quicker. Perhaps you want to create a language that will use fixed-point math to do 3-D graphics calculations much faster than SANE numerics. Or maybe you want to take apart the front end of the compiler to create a parser for a ZORK-like game, or a cross reference generator, or...

Well, whatever the reason, this is your chance! Integer BASIC is a very straightforward program, and with a few helpful pointers, you can make all sorts of changes. In this chapter, I'm going to suggest a few projects, and give you some hints to get you going. I'm not going to tell you exactly how to make the changes – that would take the fun out of it! – but I will try to guide you to roughly the right area so you can make the changes in a reasonable amount of time. With a little poking around and some imagination, I'm sure you will come up with other changes you want to make, too.

If you do make some changes, I'd like to hear from you. If you get a chance, drop me a line and let me know what you changed.

Copyrights and All That Stuff

It's worth pointing out a few legal tidbits at this point. While I highly encourage you to make changes to this program for your own enjoyment, the compiler is still a fully copyrighted program. Even if you make changes, you can't call the program your own and distribute it. While you can take any ideas away from the program that you like, you can't take the compiler away wholesale, even with changes or additions. If you want to distribute changes to the compiler, you must first get permission from the copyright owner (the publisher, in this case). The same is true, incidentally, for any copyrighted program you may get the source for.

Make Line Numbers Optional

Nothing is more universally hated in BASIC than those unnecessary line numbers. They're pretty easy to get rid of in Integer BASIC. I did leave them in because I didn't want to make changes from the original language without good reason (remember the discussion of standards?), but I also left them in to get you to change something!

Getting rid of line numbers is really pretty easy. In the Parser, you will find the section that handles line numbers roughly in the middle of the Compile procedure, which is the last subroutine in the file Parser.Pas:

```
if token.kind = intconst then begin  {handle a line number}
   if token.ival <= slineNumber then
      FlagError(8);
   slineNumber := token.ival;
   Gen0Label(d_lab, StatementLabel(token.ival, true));
   NextToken;
   end {if}
else
   FlagError(9);
```

This one is so easy I can't help but give it away: all you have to do is get rid of the else clause. That's where the error is flagged, and there is no penalty for simply not having a line number.

Add MoveTo, LineTo

Adding statements to BASIC is really pretty easy, and you might want to add these two graphics statements to prove that to yourself. There are several steps involved. The first, of course, is to get a precise definition of the syntax for the statements:

```
MOVETO expression , expression
LINETO expression , expression
```

These statements will add two new reserved words, so the first step is to make the appropriate changes to the scanner so it recognizes the new reserved words. You need to start by adding movetosy and linetosy to the tokens enumeration in BASICCom.Pas. This creates two new tokens. Next, you need to move to Scanner.Pas, adding the two tokens to the tokenNames array, initialized in InitArrays. Assuming you add these tokens in alphabetical order, you won't need to change the tokenIndex array, but you should make sure you understand how this array is used by NextToken so you believe you don't need to change it, and aren't just taking my word for it.

With the scanner handling the new reserved words, it's time to move on to the parser to add the two new statements. In the Statement procedure, add two new possibilities to the case statement that branches to the various subroutines that compile individual statements. These two statements look almost exactly like a PLOT statement, so make two copies of the PlotStatement procedure, and change the names of the copies to MovetoStatement and LinetoStatement. The only other change you'll have to make is to the last line, which makes a call to a library subroutine, ~PLOT. You'll want to substitute your own subroutine names, here.

That finishes the changes to the compiler, but you will need to add two new subroutines to BASIC's subroutine library, too. To support super high resolution graphics only, you can just call QuickDraw in your subroutines. Supporting both low resolution graphics and super high resolution graphics is a little tougher. The easiest way to support both is to write a subroutine based on the sample program given in Chapter 3, under the description of the PLOT statement. That sample implements one popular method for drawing a line by plotting individual points. You will also need to change the various graphics subroutines so they keep track of the current pen location. That's easier than it sounds, since all of the graphics subroutines call ~PLOT to plot points; you can just make sure ~PLOT records the location of the last point plotted.

Support the 8-bit Apple //

For a fairly lengthy project that isn't too difficult in terms of the amount of research you have to do or the amount of understanding you have to have about the internals of the compiler, you might want to try this project. Converting the code generator to handle 6502 code, rather than 65816 code, is pretty straight forward.

There are a few decisions you will have to make, though. For example, the libraries that come with Integer BASIC will be easy enough to convert, but what do you do about the ones that call the tools? The calls to QuickDraw could either be replaced by similar calls that handle the old 8-bit high resolution graphics mode, or you could dump the HGR and TEXT80 statements entirely, and stick with low resolution graphics and 40 column text. You could even leave out the graphics calls. Multiply and divide are a little tougher. You will have to get the code for an eight-bit version of multiply and divide, or develop the subroutines for yourself. I think developing math routines is a kick, but I'm also well knows as a warped intellect, so you might want to look into the subroutine library source for the old 8-bit version of ORCA/M, which contains multiply and divide subroutines. You can also find these routines in really old issues of magazines that covered the 6502, some old 6502 subroutine library books, and in source listings for the old Integer BASIC ROM.

Another choice is how to handle the output file. You can continue to use the ORCA linker to link your libraries, using MAKEBIN to convert the finished program to ProDOS 8 format, but you could also convert the compiler to create BIN files that can be executed directly from ProDOS 8's BASIC.SYSTEM. If you create BIN files directly, you are faced with the issue of how to merge the libraries with the code your compiler generates. Do you add limited linker facilities to the compiler? What about adding subroutines that use the compiler's code generator to "generate" the libraries on the fly? Of course, there's the old user-hated but easy choice: but the libraries in a lump that must be loaded before your program can run.

These sorts of issues keep compiler writers awake at night. It's not just technical issues that must be solved, here. Sure, you can make a choice that's right for you, but is it right for everyone else, too? From personal experience, I would say that, while the choices here may seem obvious to you, some other programmer reading this paragraph is also going to find obvious choices – and they will be different from yours.

Create a Toolbox Interface

For a fairly sporting challenge, but one that can be very rewarding, try adding a toolbox interface to Integer BASIC. The hardest part will be typing in the tool interface!

One way to handle this is to read a text file from disk when the compiler starts. The text file could contain lines like

```
LINETO 3C04 INT INT
```

to define, in this case, the LineTo call from QuickDraw, which has a call number of $3C04 and takes two integer parameters. Converting the parser's Statement subroutine to check this table is pretty easy. You will want to create a linked list of tool entries that can be searched by the parser. Once a tool entry is found, of course, the parser would look at how many and what type of parameters are needed, generating the code to put these on the stack, and finally, it would use the tool number to generate the actual tool call. While you are at it, be sure and save the error code returned by the tool, and provide a new function so the error code can be checked from inside a BASIC program.

If you make these easy changes, you will start to see some problems crop up, and solving all of them is far from trivial. The first is that the toolbox uses far more data types than Integer BASIC. To get at most of the tool calls, you will need to at least add long integers (see the next section). You can handle most of the other tool calls by coming up with a scheme to allow you to pass the address of an array to a tool; this array can be used like Pascal's record or C's struct to pass more complex data types to the tools.

Another problem you will hit head on is a speed bottleneck. Reading in a text file with all of the tool definitions is a great way to get things working, but it takes time. Once things are working, you might take a serious look at the format you used to store the text information once it is read into the compiler. By writing the file back out in this internal format, and reading the processed file, you can save a lot of time.

There are also hundreds of tool calls, and searching a linked list or array to see if an identifier is one of them takes an enormous amount of time. This is a great opportunity to put binary trees or hash tables to work; you can find good descriptions of either of these in almost any algorithms book or data structures book, as well as many introductory programming books and magazine articles.

Add Long Integers or Floating-Point

The first few changes were relatively easy, but this one is not. Adding a new variable type will require a fairly large number of changes to the compiler. You will need to track down how integers and strings are handled in the scanner and parser. You will also need to make extensive changes to the expression handling subroutines, since it will now be possible for +, for example, to be either an integer addition or a floating-point addition. The operands could even be of two different types, and you will need to convert the integer operand to a floating-point value, then perform a floating-point add.

While these modifications are extensive, they are not out of range of someone willing to spend a few weekends reading the SANE manuals and plowing through the compiler. The reward is fairly great, too: with real numbers supported, and with the addition of multiply subscripted arrays, you are well on your way to implementing a compiler for AppleSoft! Most of what is left is really pretty simple, falling into the category of adding more statements like LINETO and MOVETO, which, as you saw in the last section, is really very easy. Top that off with subroutines and a judicious mix of statements from other languages and you end up with a very nice BASIC compiler. Oh, and don't forget your test suite!

Create a Cross Reference Program

So far, all of the projects have been to change the compiler, either adding new features or making changes to the language. Here's a simple project that shows you how to put some of the code from the compiler to work to create a new program.

One utility that you will find around programming circles is a cross reference generator. A cross reference generator is a program that scans a program, creating a list of all of the identifiers and where they appear. This can help you track down bugs in programs that involve two different parts of the program modifying the same variable. You can also find variables that are declared, but never used, or track down all of the places a particular variable is used.

You can create a cross reference program for Integer BASIC with a few quick changes to the scanner. Start by removing the parser, code generator, and semantic routines. What you are left with is the scanner and some support subroutines.

Next, change the program around so it reads the command line to find out what file to process, instead of using a GetLInfo call. You can find out how to read a name from the command line by looking up the CommandLine subroutine in your ORCA/Pascal manual.

The main program should be changed to a quick loop that calls NextToken until the scanner finds an end of file token. At this point, you have a working program. It doesn't do anything useful, but this is a great time to stop and make sure nothing is broken.

The cross reference generator should list the file, printing line numbers, and later list all of the symbols found, with the line numbers the symbols are used on. The scanner can already print a line, so you just have to turn that feature on permanently. As for keeping track of the symbols used, the scanner is also telling you what sort of token has been found. Each time the scanner reports that an ident token was found, stick it in a linked list (checking first to see if it is already there, of course). Each of the identifiers should have a linked list of line numbers. If you're a little week on linked lists, you could use arrays instead, but linked lists are better, since you don't have to worry about having too many identifiers or too many line numbers for one identifier. The scanner is already keeping track of line numbers, so you can pluck the proper line number right from the scanner.

Once you get to the end of file token, step through your list of symbols and print them.

Changing your cross reference generator to handle Pascal or C is pretty easy. All you really have to do is change the way strings and comments are handled, then change the list of reserved words. For C, you also need to make the program case sensitive. In other words, once you get the first one working, it won't take long to create cross reference generators for other programs, too!

Appendix A
Porting Old Integer BASIC Programs

Using Convert

If you have access to any old Integer BASIC programs, you might take a crack at converting them to the Apple IIGS. The most likely place to find an Integer BASIC program is on a DOS disk, so the first step in converting them is to move the program to ProDOS. You can use Apple's FID program from their ProDOS utility disk, Copy II Plus, or some other personal favorite for this step. Integer BASIC programs have their own file type. Internally, they are a series of tokens, so the next step is to convert the Integer BASIC token file into an ASCII text file. If you've installed the CONVERT utility from your Integer BASIC program disk, you can do this step like this:

```
convert basicprog >prog.bas
```

In this example, BASICPROG is the tokenized Integer BASIC program. The CONVERT utility writes the ASCII text to standard out. To put it in a file, you redirect output, as we've done here.

The last step is to change the language type of the ASCII file using the CHANGE command. You do that like this:

```
change prog.bas ibasic
```

While a surprising number of Integer BASIC programs run just fine under this compiler, there are some things to watch out for. The most serious is a trick people used to use to tack a machine language program onto the end of an Integer BASIC program. If you try to convert a program that used this trick, the CONVERT utility will give up: it will report a few of the tokens as something it couldn't translate, then print a message telling you that the program can't be translated.

Another problem that will crop up in a few programs is the use of disk commands. Like AppleSoft programs, Integer BASIC programs got at the disk by printing strings that started with control D. This just doesn't work under ORCA/M. Sure, you could intercept these strings in the subroutine library and reroute the commands to the shell. Some of them, like CATALOG, would even work. Loading and saving files, though, has changed so much that the old Integer BASIC programs doing a BLOAD or BSAVE don't stand a chance of working correctly on the Apple IIGS. If you really want to convert a program that uses the disk, you could write some assembly language subroutines to do the disk access.

The last problem is that peeks and pokes just aren't safe on the Apple IIGS, and the CALL statement has been changed to call 65816 subroutines instead of fixed 6502 subroutines. CONVERT checks for these three statements, and warns you if there are any. You can still convert most programs that use PEEKs, POKEs and CALLs, though – the next section shows you how.

Useful Subroutines

While most of the old Integer BASIC programs from the golden days of the Apple II used PEEKs, POKEs or CALLs, a surprising number of these programs can still be compiled under Integer BASIC with just a few minor changes. That's because most programs used PEEKs, POKEs or CALLs to do a fairly small number of things. I've added a short list of subroutines you can call to convert these programs. The table below shows the old statement with the newer call to the right, and a short description telling what the subroutine does. All of these subroutines are included in the BASICLIB library you installed to use Integer BASIC, so all you have to do to use these subroutines is:

```
CALL subroutine
```

where subroutine is one of the names from the table.

Old Form	New Form	Use
`CALL -198`	`CALL BELL`	Beep the speaker.
`CALL -380`	`CALL NORMAL`	Switch to normal text.
`CALL -384`	`CALL INVERSE`	Switch to inverse text.
`CALL -868`	`CALL CLEAREOL`	Clear to the end of the line.
`CALL -936`	`CALL HOME`	Clear the screen and home the cursor.
`CALL -958`	`CALL CLEAREOS`	Clear to the end of the screen.
`CALL -1008`	`CALL CURSORLEFT`	Move the cursor left one space.
`CALL -1052`	`CALL BELL`	Beep the speaker.
`PEEK -16336`	`CALL CLICK`	Click the speaker.
`PEEK -16384`	`CALL KEYBOARD`	Read the keyboard. The value of $00C000 is placed in a variable called KEY, which must be used somewhere in the Integer BASIC program.
`POKE 50,63`	`CALL INVERSE`	Switch to inverse text.
`POKE 50,255`	`CALL NORMAL`	Switch to normal text.
`POKE -16368,0`	`CALL CLEARSTROBE`	Clear the keyboard strobe.

Index