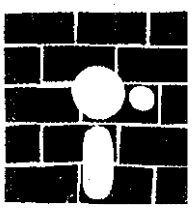
A SIMPL COMPILER PART 1: THE BASICS

BY JONATHAN AMSTERDAM

An implementation of a compiler for a simple structured language



In this article—the first of a three-part series for the construction of a compiler for a high-level language—I will discuss the basics of the compiler. Next month I will talk about procedures and functions, and in the third part of the series I will describe some of the compiler's extensions.

Three of my earlier Programming Projects are prerequisites for this one. "Context-Free Parsing of Arithmetic Expressions" (August, page 138) explains the parsing technique I will be using. "Building a Computer in Soft-ware" (October, page 112) describes VM2, the virtual machine for which my compiler is targeted. And "A VM2 Assembler" (Nov-ember, page 112) details the assembly-language code that the compiler will generate.

THE SIMPL PROGRAMMING LANGUAGE

I will be describing a compiler for a language of my own design, called SIMPL. SIMPL, which stands for "SIMPL Isn't Much of a Programming Language," isn't much of a programming language. SIMPL's grammar is given in figure 1. There are a few points that are not described by the grammar. An identifier is any string of letters and numbers beginning with a letter. Unlike most implementations of Pascal, SIMPL is case-sensitive, so the identifiers READ and Read mean different things. SIMPL key-words, like PROGRAM and BEGIN, are capitalized. Comments in SIMPL are delimited by braces ({ }). As in Pascal, character constants are delimited by single quotes, but SIMPL also allows the backslash character (\) to act as an escape. When followed by an n or a t, the backslash denotes a new line (carriage return) or tab; when followed by any other character, it denotes that character. For example, the character constant for the single quote looks like ' \ '.

SIMPL's WHILE and IF statements, like those of Modula-2, are explicitly terminated by an END. The AND operator has the same precedence as OR, and both have weaker precedences than those of all other operators, so it is unnecessary to put parentheses around expressions connected by AND and OR. Furthermore, expressions surrounding an AND or OR will be evaluated from left to right, and no more than necessary will be

(continued)

Jonathan Amsterdam is a graduate student at the Massachusetts Institute of Technology Artificial Intelligence Laboratory. He can be reached at 1643 Cambridge St. #34, Cambridge, MA 02138.

evaluated. For example, in the expression TRUE AND FALSE AND TRUE, the first TRUE will, be evaluated and then the FALSE will be evaluated; at that point, evaluation of the second TRUE will be skipped because the value of the whole expression must be FALSE.

SIMPL has procedures and functions much like those of Pascal, but rather than assigning a value to the name of the function, at value is returned from a SIMPL function using the RETURN statement. A RETURN without an expression can be used to exit from a procedure. It cannot be used in the main program.

SIMPL has two built-in procedures, READ and WRITE. These procedures can only read or write integers or characters and only from the key-board or screen. Both take any number of arguments. The arguments to READ must be variables; those to WRITE can be any expression.

SIMPL supports variables of types INTEGER, BOOLEAN, and CHAR. It has five kinds of statements: WHILE, IF, RETURN, assignment, and procedure call. SIMPL has no FOR loops, strings, case statements, arrays, constants, reals, type declarations, records, sets, repeat loops, GOTOs, labels, files, scalar types, VAR parameters, pointers, or math functions. Indeed, SIMPL lives up to its name.

Still, even for this language, writing a compiler is not easy. The compiler consists of 12 modules and is over 3000 lines long. The source code, written in Modula-2, is available on BYTEnet Listings at (617) 861-9764.

WHAT IS A COMPILER?

A compiler is a program that translates other programs from one form to another. The compiler's input is a source file, which is a sequence of characters that constitutes the human-readable text of the program. Some compilers translate this directly into object code, which can be loaded and executed by the machine. Other compilers produce as output another text file containing an assembly-language version of the program. This text file must then be translated into object code by an assembler. Although going directly from source code to object code saves a step in the translation process, my compiler takes the second approach for two reasons. First, producing an assembly- language version of the program makes it easier to write the compiler because I don't have to worry about bookkeeping details (like forward references of labels) that the assembler can handle. Second, I can examine and change the human-readable assembly-language file. This can be useful for debugging the compiler and hand-optimizing its output.

Readers of "A VM2 Assembler" will recall that an assembler is also a pro-gram translator. But whereas assembly is relatively easy, compilation is considerably more difficult. There are two reasons for this, the first of which is fairly clear: Assembly-language pro-grams correspond line for line with the object code that has to be generated, but high-level-language programs, by definition, do not. A standard computer instruction set does not provide WHILE loops, IF. . . THEN. . . ELSE statements, evaluation of arithmetic expressions, etc. Furthermore, at the machine level a variable is simply a memory location, but a high-level-language variable is something else. For one thing, it has a particular type associated with it, as well as a scope, or range of visibility (a topic I'll be discussing at length in part 2 of this series). If you compare the program in listing 1 with the one in listing 2, you'll get a sense of how different; high-level language and assembly language are from each other.

Compilation is difficult for another reason as well: Users of high-level languages would like the object code produced by the compiler to be just as short and run just as fast as hand-coded assembly-language programs. So compiler writers strive to improve the compiler's generated code. For example, it is a good idea to keep frequently used values in the registers of the computer because they can be accessed faster than memory locations; therefore compiler writers have developed algorithms for optimizing register usage. Also, most computers have "special case" instructions that can speed up certain common opera-

```
program ::= PROGRAM id; vars routines block
vars ::= empty | VAR varlist varlist ::= decl | decl varlist
decl ::= idlist : type ;
idlist :: = id | id , idlist
type ::= INTEGER | BOOLEAN | CHAR
block ::= BEGIN stmts END
stmts ::= empty | stmt : stmts
stmt ::= while | if | return | assign | call
while ::= WHILE expr DO stmts END
if ::= if elsif END
elsif ::= expr THEN stmts else
else ::= empty | ELSIF elsif | ELSE stmts
assign ::= id := expr
exprlist ::= expr | expr, exprlist
expr ::= expr | relexpr | relexpr OR expr | relexpr AND expr
relexpr ::= intexpr | intexpr relation intexpr
intexpr ::= term | term + intexpr | term — intexpr
term ::= factor | factor * term | factor / term
factor ::= id | number | funcall | char |—factor | NOT factor
| ( expr)
relation ::= > | < | = | <> I >= I <=
```

Figure 1:
The SIMPL grammar used by the parser

Listing 1: A program written in SIMPL that calculates the greatest common divisor of two integers, using Euclid's algorithm.

```
PROGRAM Euclid;

VAR m, n, temp, r:INTEGER; BEGIN
WRITE('?');   { Prompt user for two integers }
READ(n);
WRITE('?');
READ(m);
IF n < m THEN            { Make sure n is the larger of the two }
   temp := n;
   n := m;
   m := temp;
END;
r := n - m*(n/m);         { r : = n MOD m }
WHILE r > 0 DO
n := m;
      m := r;
 r := n - m*(n/m);         { r := n MOD m }
END;
WRITE(m);   { m is the GCD—output it }
END.
```

tions (such as incrementing a number), and a good compiler will use these instructions where appropriate.

While writing my compiler, I did have to deal with translating high-level SIMPL statements into low-level VM2 instructions, but I could avoid some of the complexities of generating good code because I was compiling for VM2, a machine I designed to make it easy to compile high-level languages. For example, I didn't have to worry about register allocation because VM2 doesn't have any registers (except for some special-purpose ones with which the Compiler needn't be concerned). This simplification was behind my decision to make VM2 a stack machine. Also, VM2's instruction set is simple and provides no special-case instructions, so I don't have to worry about using them.

A compiler's job can be divided into at least four phases: lexical analysis, parsing, type checking, and code generation.

LEXICAL ANALYSIS

The compiler's first task is to translate the stream of characters that constitute the input into a more agreeable form. The lexical analyzer transforms the character stream into a stream of tokens, or lexical items that are meaningful to the compiler.

What counts as meaningful depends on the program, of course. My lexical analyzer for SIMPL has many different types (or classes) of tokens, including IDENTIFIER for variables, INT for integers, and a different class for each keyword. For example, the lexical analyzer will consume from the input the five characters W, H, I, L, and E, when they occur consecutively and are delimited on both sides by white space (spaces, tabs, or carriage returns), and will then return a token of class WHILE. If the lexical analyzer sees the characters 3, 4, and 5 occurring consecutively, it will convert them into an integer, 345, and return a token of class INT, which also contains the number 345.

The lexical analyzer's design is similar to that of the VM2 assembler's lexical analyzer. Where the lexical analyzer of the assembler allowed you to "unget" the last character that was taken from the input, the SIMPL lexical analyzer allows you to unget a token—that is, to arrange matters so that the next call to the lexical analyzer

Listing 2:
VM2 assembly- language code produced by the SIMPL compiler upon compilation of the program in listing 1.

```
        BRANCH Euclid
m:      0
n:      0
temp:   0
r:      0
Euclid:
        PUSHC   '?
        WRCHAR
        RDINT
        POPC    n
        PUSHC   '?
        WRCHAR
        RDINT
        POPC    m
        PUSH    n
        PUSH    m
        LESS
        BREQL
LI
        PUSH    n
        POPC    temp
        PUSH    m
        POPC    n
        PUSH    temp
        POPC    m
L1:
        PUSH    n
        PUSH    m
        PUSH    n
        PUSH    m
        DIV
        MUL
        SUB
        POPC    r
L2:
        PUSH    r
        PUSHC   0
        GREATER
        BREQL   L3
        PUSH    m
        POPC    n
        PUSH    r
        POPC    m
        PUSH    n
        PUSH    m
        PUSH    n
        PUSH    m
        DIV
        MUL
        SUB
        POPC    r
        BRANCH  L2
L3:
        PUSH    m
```

will return the same token.

PARSING

SIMPL programs are more than just lists of tokens. They have a complex structure, as reflected by the grammar shown in figure 1. A WHILE loop, for example, consists of a Boolean test and a group of statements; an assignment statement has a variable on the left side and an expression on the right; and so on. It is the parser's job to impose structure on the token stream. In my compiler, the parser will actually construct a parse tree—a data structure that reflects the structure of the program. A typical parse tree is shown in figure-2.

Readers of "Context-Free Parsing of Arithmetic Expressions" will recall that I built a parse tree from an expression by writing a procedure for each rule of the grammar and by having the procedure consume just as much of the input as was necessary to parse its particular rule. I'll use the same technique, called top-down or recursive-descent parsing, for my compiler.

TYPE CHECKING

Every variable and function in SIMPL has a particular type—integer, character, or Boolean—and the use of these types is governed by several rules. For example, you can only add integers, you can only compare two expressions of the same type for equality, and the types of the arguments to a procedure and the procedure's formal parameters must be identical. Pascal programmers should be familiar with these rules. The compiler enforces them by checking each expression as it is parsed to make sure it conforms. Since variables, procedures, and functions must be declared before they are used, the compiler always knows the types of the variables involved in the expressions.

CODE GENERATION

In the fourth phase of compilation, the compiler translates the parse tree into the actual assembly-language code. As I mentioned, this process can be very involved, but for my compiler it is fairly straightforward.

Some compilers have additional phases. Often, an optimization phase occurs either just before or just after code generation. In this phase, trans-formations are made to either the parse tree or the assembly-language code to make the generated code more efficient.

INTERMEDIATE REPRESENTATION

As I've described it, the SIMPL compiler constructs a parse tree from the input and then generates code from the tree. Why not skip the parse tree altogether and have the parser call the code generator directly? This is certainly possible and has the advantage of speed—constructing the parse tree takes some time. But I think it's a good idea to have some sort of intermediate representation (IR) like a parse tree for a couple of reasons.

First, it allows you to separate the "front end" of the compiler—the lex- i cal analyzer, parser, and type checker—from the "back end"—the code generator. The IR serves-as a common language that lets the two ends communicate. For instance if your compiler is too big to fit into memory all at once, you can first - generate the IR, then swap in the back end to generate the code. Such multi--pass compilers are common, but mine will only make a single pass over the input.

An IR also makes it easy to mix and match compiler parts, which means you can use the same code generator for Pascal and C or the same front end for a compiler that generates VM2 code and for one that produces Motorola 68000 code. Or, instead of generating code from the IR, you can write an interpreter for it. In short, an IR helps make your compiler more modular, and modularity is the essence of good-software engineering.

There's another important reason for using an IR: It provides a more abstract view of the program being compiled. The source code and object code are just lists of characters or numbers, but the IR can represent the program in a way that more clearly reveals its structure to the compiler. For instance, if the compiler wants to know what statements are part of the WHILE loop in listing 1, it can much more easily determine that information from the parse tree in figure 2 than from the source code in listing 1. A compiler may want to use this in-formation for optimizing the code. The IR can aid code generation by making certain aspects of the code explicit. For a stack machine like VM2, the parse tree is a natural choice.

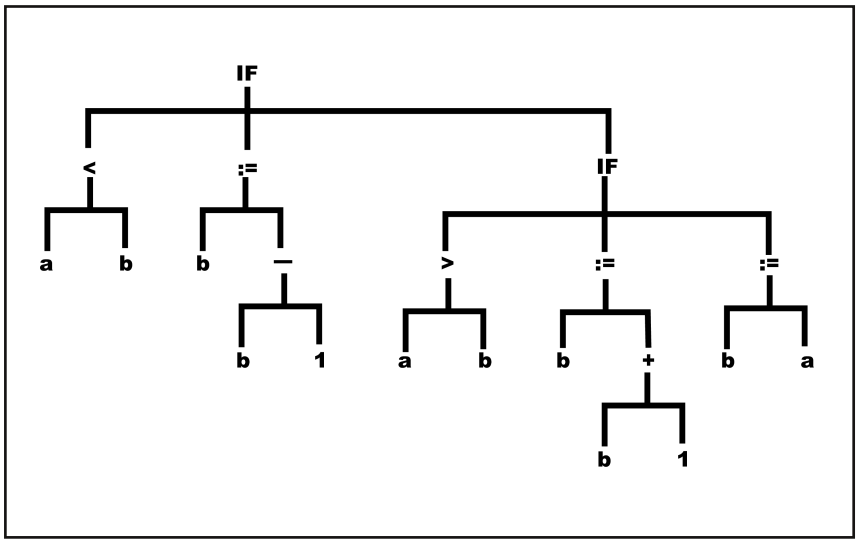


Figure 2:
Parse tree for the SIMPL statement IF a < b THEN b := b - 1; ELSIF a > b THEN b:= b + 1; ELSE b := a.

THE COMPILER IN DETAIL

I described the various phases of compilation as if they occurred one after the other in a simple procession. In fact, they overlap in a com-

plex dance choreographed by the parser. In the text that follows I'll explore the compiler in greater detail, using the grammar of SIMPL as my guide. I will be referring to specific VM2 instructions

(the instruction set can be found in my October Programming Project "Building a Computer in Software"). | Editor's note: lowercase routine names refer to routines written in Modula-2, which im-plement the compiler; uppercase names are either SIMPL keywords or VM2 instructions.|

Compilation begins with the first line of the grammar. This line corresponds to a procedure in the parser named, not surprisingly, program. The program procedure first calls the lexical analyzer to make sure the first token is the keyword PROGRAM. Then it gets another token, which should be an identifier. This token is the name of the program. The parser now calls the code generator to out-put a BRANCH instruction using the program name as a label. This must be done because the main program code is not compiled until after the routines and global variables are com-piled; you have to jump over the routine and variable code to get to the main program. The main program occurs at the end of the file because the SIMPL compiler is a one-pass compiler—everything has to be de-fined before it's used, including the main program's procedures, functions, and variables.

The program procedure then calls two other parser procedures, vars and routines. Then it outputs the program name as a label, calls the block procedure to compile the text of the pro-gram, and finally, when it sees the dot token, outputs a HALT instruction. A degenerate SIMPL program—one with no variables, routines, or main program body—such as PROGRAM foo; BEGIN END. compiles into the following two-line program:

```
      BRANCH foo
foo:  HALT
```

VARIABLE DECLARATIONS

The next five rules—those for vars, varlist, decl, idlist, and type—handle variable declarations. You'll note that the vars rule can be empty; that is, the program might not have any variable declarations. The parser can easily recognize this simply by checking whether the next token is VAR:. If so, the varlist procedure is called; if not, the parser calls the lexical analyzer's unget Token routine to return the token to the input for future consumption.

The varlist procedure calls the decl procedure, then examines the next token to see if it is a BEGIN, PRO-CEDURE, or FUNCTION. If it's any of these three, then there are no more variable declarations; if it isn't, then there are more declarations, and var-list calls decl again. In either case, the token is ungotten.

A declaration consists of a list of identifiers—the variable names- followed by a colon and then by a type name. The decl procedure begins by calling idlist, which returns the iden-t ifiers in a linked list. Then decl reads the type name and calls the code generator to generate code for the variables. Then, a label—the variable's name—has to be generated, followed by some initial value for the variable. I use 0 as an initial value, which is interpreted as FALSE for Boolean variables and as the ASCII NUL character for character variables.

However, the compiler must do more than merely generate code. It also has to store information about these variables for use later in the compilation. The type of a variable is needed for type checking, and the fact that a variable is global will affect how the code generator produces references to it. My compiler puts all this information into a record called a symbol and inserts it into its symbol table. The symbol table is a close relative of the label table used in the VM2 assembler—it has a similar purpose and is also best implemented as a hash table. The com-

(a)	var := expr	|	(code for expr)
		|	POPC var
(b)	WHILE expr DO	|	START: (code for expr)
	stmts;	|	BREQL end
	END;	|	(code for stmts)
		|	BRANCH START
		|	END:
(c)	IF expr THEN	|	(code for expr)
	stmts	|	BREQL END
	END;	|	(code for stmts)
		|	END:
(d)	IF expr THEN	|	(code for expr)
	stms1	|	BREQL ELSE
	ELSE	|	(code for stmts1)
	stmts2	|	BRANCH END
	END;	|	ELSE: (code for stmts2)
		|	END:
(e)	WRITE(intexpr, charexpr);	|	(code for intexpr)
		|	WRINT
		|	(code for charexpr)

Figure 3:
VM2 code generated for SIMPL statements.

piler's symbol table is rather more complex, though, because it also has to handle local variables. I'll defer discussion of its complexities to the second part of this series.

STATEMENTS

The bulk of the compiler is involved in translating SIMPL statements. Statements occur in lists, as the stmts line in figure 1 indicates. A list of statements can be empty. How does the stmts procedure recognize this? If you examine the grammar closely, you'll notice that any statement list is ended by one of the three keywords ELSE, ELSIF, or END, so these can be used to tell when a statement list is empty.

Let's examine each of the SIMPL statements in turn. I will defer treatment of the procedure call and RETURN statements to part 2 of this series. Figure 3 illustrates the code generated for each type of statement.

ASSIGNMENT STATEMENTS

To parse an assignment statement, the parser first calls the lexical analyzer to get the identifier on the left side of the statement. The parser can't distinguish an assignment statement from a procedure call until it calls the lexical analyzer to read the next token. If the next token is a :=, the parser knows this is an assignment statement. Then the parser checks to see if the identifier has been defined by looking it up in the symbol table. It also checks to make sure the identifier is the name of a variable, not a procedure or function. The parser then calls the expr routine to parse the expression and subsequently calls a special tree-building procedure to construct the parse tree from the variable and the expression. Parse trees consist of several different types of nodes. The tree-building procedure for the assignment statement creates a symbol node for the variable, an expression node for the expression, and makes both these nodes the children of an assignment node. It also checks the types of the variable and the expression to make sure they match.

To generate the code for an assignment statement, the code generator first generates code for the expression. At run time, after this code is executed, the result of the expression will be on the top of the stack: To store it in the variable, the compiler heeds only to generate a POPC instruction with the variable's name as an argument (see figure 3a).

THE WHILE STATEMENT

The parser's while procedure reads the expression following the WHILE token and checks to make sure its type is BOOLEAN. It then reads the DO followed by a list of statements. A tree-building procedure creates a WHILE node and makes the Boolean expression and the statement list its children.

To generate code for a WHILE statement, the compiler first creates two new labels; let's call them START and END for now, although in the actual code generator there is a special function that generates a unique label

name each time it is called in order to avoid name conflicts. The code generator begins by outputting the START label and then generates code for the Boolean expression. If this expression evaluates to FALSE, the loop shouldn't be executed, so the instruction BREQL END is output; this will have the effect of branching to the END label if the expression evaluates to FALSE. Now the body of the loop is generated, followed by a BRANCH START instruction to repeat the loop. Finally, the code generator outputs the END label (see figure 3b).

THE IF STATEMENT

The IF statement is a bit tricky to parse because it may contain ELSIFs and an ELSE. I have divided the work among three procedures, if, elsif, and else. You can tell when an ELSE is empty by seeing if the next token is END.

As with the WHILE statement, I check to make sure that I have parsed a Boolean expression. When I am done parsing, I build a tree whose root is an IF node and whose three children are the Boolean test, the THEN part, and the ELSE part. Note that the ELSE part may itself be another IF statement; this is what hap-pens when ELSIF is used (see figure 1). If there is no ELSE part, I fill the ELSE slot of the IF node with the value NIL.

To generate code for an IF with no ELSE part, I create a single new label: END. I generate code for the Boolean expression and then output a BREQL END instruction, because I want to skip the THEN part if the expression is false. Then I generate the THEN part and finally output the

```
(a) 1 + x = 5 - y * z          |    PUSHO1
                               |    PUSH  x
                               |    ADD
                               |    PUSHC 5
                               |    PUSH  y
                               |    PUSH  z
                               |    MUL
                               |    SUB
                               |    EQUAL
-------------------------------------------------------------------------------
-----
(b) a = b AND c < > d          |    PUSH  a
                               |    PUSH  b
                               |    EQUAL
                               |    BREQL FALSE
                               |    PUSH  c
                               |    PUSH  d
                               |    NOTE  Q
                               |    BRANCH      END
```

Figure 4:
VM2 code generated for SIMPL expressions.

END label (see figure 3c).

When an IF has an ELSE part, I create two new labels—ELSE and END. Again, I first generate the Boolean expression. Now, if this expression is false I want to branch to the ELSE part, so I generate a BREQL ELSE instruction. Then I generate the THEN part of the code, but I follow it by a BRANCH END instruction so control doesn't fall through to the ELSE code. I then out-put the ELSE label followed by the code for the ELSE part. Finally, I out-put the END label (see figure 3d).

THE READ AND WRITE STATEMENTS

The parsing of READ and WRITE statements is similar; in both cases, i read a list of expressions and attach this list to either a READ or a WRITE node, as the case may be. I also check each argument to make sure it is of type INTEGER or CHAR. For a READ statement I check to make sure the argument is a variable.

To generate code for a WRITE statement, I generate the code for each expression, immediately followed by either a WRINT or a WRCHAR instruction, depending on the type of the expression. For each variable in a READ statement, I first generate either a RDINT or a RDCHAR instruction, then a POPC with the name of the variable (see figures 3e and 3f).

EXPRESSIONS

The parser used for expressions is similar to the one i described in "Context-Free Parsing of Arithmetic Expressions." It has been expanded to handle variables, function calls, and Boolean operators, and it has been made left-associative so that it parses arithmetic operators in this way: a + b + c will be parsed as (a + b) + c rather than as a + (b + c).

The compiler type checks expressions as their trees are constructed. Before constructing the tree for a + b, for example, the compiler makes sure that a and b have been defined as variables of type INTEGER. The Boolean operators AND, OR, and NOT require Boolean operands. For relational operators like = and >, it doesn't matter what type the two operands are, so long as they are of the same type.

Because of the way I designed VM2's instruction set, generating code for expressions is easy. I first generate the code to place the operands on the stack using the instructions PUSH (for variables) or PUSHC (for constants), then I output the instruction corresponding to the operator. All operators but AND and OR have a corresponding instruction. See figure 4a for the code generated by an expression.

The operators AND and OR are special cases because the second argument shouldn't be evaluated unless absolutely necessary. I treat them much like IF statements. For AND, I first create two new labels, which I'll call FALSE and END. I generate the code for the first argument, then generate a BREQL FALSE instruction. After that I generate the code for the second argument and then a BRANCH END. Then I output the label FALSE, followed by a PUSHC 0 instruction. Finally, I output the END label. This code will have the following effect at run time: If the first argument evaluates to FALSE, then a FALSE (that is, a 0) is pushed on the stack and the code for the second argument is skipped. If the first argument comes out TRUE, then the second argument is evaluated, and its result is the result of the entire AND. Things are reversed for OR. If the first argument is TRUE, I push TRUE and skip the second argument; otherwise, the result of the second argument is the result of the entire OR. The code generated for an AND statement is in figure 4b.

ERROR HANDING

It is notoriously difficult to write a compiler that can recover gracefully from an error in the source program and continue compiling. On the other hand, you wouldn't want to make the compiler too lenient when it en-counters an error—a friend of mine speaks of how he used to feed interoffice memos to a COBOL compiler to see if they would compile successfully. They often did.

One solution to the error-handling problem is to stop the compiler after the first error is found, but this is a cop-out. I'd like the compiler to find as many errors as it can so I can try to fix them all at once. At the same time, I want to avoid cascades of errors, where the first error triggers new ones that wouldn't have appeared if the first error had not occurred. A compiler that frequently produces error cascades is just as bad as one that dies after the first error because you will trust the compiler only for the first error anyway. Error cascades are particularly common for syntax errors because it's hard to know where to continue parsing when something unexpected appears.

My compiler is far from ideal at handling syntax errors, but it deals with other errors reasonably well. The basic rule I use in parsing is, if an expected token does not appear in the input, pretend it did appear and continue on without it. For example, if the keyword PROGRAM does not appear at the beginning of the pro-gram, the compiler prints out an error message and acts as if it did appear. Similarly, if the program name is omitted, the compiler prints an error message, makes up a dummy name, and continues. Sometimes such errors are not handled so gracefully. If you meant to write WHILE a > b DO ... and instead omitted the WHILE key-word, the parser would read the identifier a and begin compiling the statement as if it were a call or assignment. The resulting error cascade isn't pretty.

A common source of error cascades is an undeclared variable. Some compilers will tell you over and over again that a variable is undeclared, outputting a message every time the variable is encountered. A much better solution is to print the message once and then ignore that variable from then on. I do this by inserting the variable in the symbol table once it has been seen. But what type should it be given? If I decide to make it an integer when in fact the programmer intended it to be a Boolean, I will find myself with, a bunch of unnecessary type errors.

My solution is to give undeclared variables the special type Unknown. Unknown is compatible with

every other type, so the type-checking routines will never find a problem with a variable of type Unknown.

IMPLEMENTATION NOTES

The key to writing a program as large as a compiler is breaking it up into small, independent parts. As I said before, my program is divided into 12 modules.

The main module is called Compiler. It is short and does little more than read an input filename and call the parser. Because of its size, the parser is divided into three separate modules: Parser handles the bulk of SIMPL, ExprParser takes care of expressions, and Routines is concerned with procedures and functions.

A common source of error cascades is an undeclared variable.

The LexAn module contains the lexical analyzer. The Token module defines the token data structure as well as some other useful types and constants. TypeChecker handles type checking, of course. Equally obvious is the function of SymbolTable. The Symbol module defines the data structure used for storing identifiers in the symbol table, and the Node module defines the data structure used to construct parse trees. Node also contains the important tree- building procedures.

Two other modules constitute the compiler's back end. CodeGen takes parse trees and Calls procedures in CodeWrite to actually output the VM2 instructions.

My motivation for dividing the work up as I did was to distribute the load evenly. The code generator's time is split between the abstract work of generating code from parse trees, handled by CodeGen, and the nittygritty details of outputting VM2 assembly-language instructions, handled by CodeWrite. The parser definitely has the hardest job in this compiler, so I tried to make its tasks as simple as I could. The lexical analyzer looks up keywords in the symbol table and provides several routines that handle the work of generating syntax errors, thus relieving the parser of those burdens. The tree-building routines of the Node module not only construct trees but do much of the type checking as well. The code generator need only be given the parse tree to generate code.

CONCLUSION

As it stands now, the compiler is in-complete. The implementation of procedures and functions remains to be done, but the framework is in place. At heart, all compilers resemble the program described here. ¶