A Mechanic's Guide to Grammar

Part III: A homemade compiler

By Brian Hayes

The gap between theory and practice is one of life's recurrent disappointments. In predicting the weather, navigating by the stars, writing a fugue or fixing a carburetor, a grasp of theory is valuable, perhaps even essential, but never quite sufficient. It is the same in designing a compiler for a programming language. Grammatical principles guide the construction, but craft is needed too.

Parts I and II of this series focused on the theoretical view of language. It was shown that a language can be defined as a set of strings made up of symbols drawn from a specified alphabet. Rules of grammar determine which strings are sentences of the language. The grammar can be embodied in an abstract machine called a recognizer, which reads an input string and either accepts it as a sentence or rejects it as being ill-formed.

The recognition machines of linguistic theory are idealized devices, built out of pure thought stuff. Nevertheless, an ideal recognizer can be closely approximated in a program written for a real computer. The structure of the program follows directly from that of the underlying grammar. In some cases the transformation of a grammar into a recognizer can even be automated, a sure sign the process is well understood.

But a recognizer is not a compiler. A programmer wants more in the way of output than a pass-or-fail indication of whether a program has any syntax errors; what is wanted is object code, ready to execute on some machine. In the generation of that code a formal grammar offers only limited help. It contributes even less to the many housekeeping tasks of the compiler: recording the names of variables and procedures, allocating and reclaiming storage space in the target machine, calculating addresses, and so on.

In this third and final article I shall examine how theory is reduced to practice in the art of compiler writing. The discussion centers on a compiler for a toy language I call Dada. The full program listing for the compiler is available from the COMPUTER LANGUAGE Bulletin Board Service, CompuServe account, and Users Group (see page 4).

Dada grammar

In developing Dada my one aim was clarity. Speed, efficiency, and utility were all sacrificed to keep the compiler simple. The vocabulary of the language and many syntactic structures are borrowed from Pascal, another language whose design was guided by a concern for ease of compilation. To simplify further I left out many of Pascal's useful features, and I have consistently done things the easy way, not the best way. The result is a good-for-nothing language but one with a compiler whose functioning can be traced in detail.

Dada has only two data types: Boolean values (true or false) and integers. There are no characters, strings, arrays, records, sets or pointers, and there is no facility for defining new types. The *if. . . then . . . else* statement is available to control conditional execution, and a *while* statement is provided for loops; the *case, for, repeat* and *goto* statements are not included. The most serious deficiency of Dada is the absence of local variables and parameters passed to procedures. All variables are global.

When so much is omitted, what remains? The most conspicuous feature of a Dada program is its organization into blocks of statements delimited by the keywords *begin* and *end.* The program starts with a header, which is followed by declarations of variables and procedures and concludes with a statement block. A procedure has the same structure on a smaller scale, except that it cannot declare variables: a header is followed by further procedure declarations and a block. Procedures can be nested to any depth.

The grammar for Dada given in Table 1 defines the syntax of the language in 21 production rules. A rule such as *Type ::= integer* | *boolean* states that wherever the symbol *Type* is encountered, it can be replaced by either *integer* or *boolean. Type* and all the other symbols that appear on the left side of the rules are nonterminals, which represent categories of symbols; only the terminals, such as *integer* and *boolean,* appear in the actual program text. Any string of terminals that can be generated by applying the production rules is a valid Dada program.

A program is written as a linear sequence of symbols, but under the surface it has a more elaborate structure. It is a tree: a hierarchy of nodes linked in parent-child relationships. The arrangement of the nodes and the pattern of links

between them encode all the syntactic relations among the elements of the program.

A grammar is a compact way of representing an infinite family of trees. Nonterminal symbols label the interior nodes of each tree, with the special symbol *Program* at the root, and terminal symbols are hung on the leaf nodes. If the grammar is unambiguous, every program corresponds to exactly one tree. Parsing the program, or analyzing its syntax, is a matter of converting the one-dimensional string of symbols into a two-dimensional tree.

Grammars and parsers

The most useful grammars for programming languages are the ones called context-free grammars. They are powerful enough to describe most programming concepts and yet are simple enough to allow efficient compilation. The definitive property of a context-free grammar is that every production rule has just one symbol on the left side of the "::=" sign. The grammar for Dada satisfies this condition, and so it is context-free. (The language itself is not entirely context-free, but for the time being I shall discuss it as if it were.)

The recognizing machine for a context-free language is a pushdown automaton, a computer in which the only available storage is a stack and the only accessible item on the stack is the one at the top. Linguistic theory offers the assurance that every context-free language can be recognized and parsed by some pushdown automaton. On the other hand, the theory does not promise that the task can be done efficiently enough to be completed in a reasonable period (say a human lifetime).

Efficiency is in question because a parser working by the most general method has a vast array of choices. It can attempt to match any production rule to any string of symbols anywhere in the source text; if the match fails, it can go on to any other combination of rules and strings. In the worst case the process does not end until all possible combinations have been tried.

The way to avoid this combinatorial explosion is to limit the parser's freedom of choice. One must guarantee that if the input can be parsed at all, the correct parse will be found by reading the symbols and applying the rules in a fixed sequence. In order to make that guarantee, further constraints must be put on the grammar. It must not only be context-free but must also satisfy additional conditions.

Dada grammar

Syntactic rules

<table>
<tr><td>Program</td><td>::=</td><td>*program* Identifier ; Declarations Block .</td></tr>
<tr><td>Declarations</td><td>::=</td><td>*var* VarList | ϵ</td></tr>
<tr><td>VarList</td><td>::=</td><td>Variable | Variable VarList</td></tr>
<tr><td>Variable</td><td>::=</td><td>Identifier : Type ;</td></tr>
<tr><td>Type</td><td>::=</td><td>*integer* | *boolean*</td></tr>
<tr><td>Block</td><td>::=</td><td>Procedures Statements</td></tr>
<tr><td>Procedures</td><td>::=</td><td>Procedure Procedures | ϵ</td></tr>
<tr><td>Procedure</td><td>::=</td><td>*procedure* Identifier ; Block ;</td></tr>
<tr><td>Statements</td><td>::=</td><td>*begin* StmtList *end*</td></tr>
<tr><td>StmtList</td><td>::=</td><td>Statement | Statement ; StmtList | ϵ</td></tr>
<tr><td>*Statement*</td><td>::=</td><td>AssignStmt | IfStmt | WhileStmt | ProcStmt | Statements</td></tr>
<tr><td>*AssignStmt*</td><td>::=</td><td>Identifier := Expression</td></tr>
<tr><td>IfStmt</td><td>::=</td><td>*if* Expression *then* Statement ElseClause</td></tr>
<tr><td>ElseClause</td><td>::=</td><td>*else* Statement | ϵ</td></tr>
<tr><td>WhileStmt</td><td>::=</td><td>*while* Expression *do* Statement</td></tr>
<tr><td>ProcStmt</td><td>::=</td><td>Identifier</td></tr>
<tr><td>Expression</td><td>::=</td><td>SimpleExpr | SimpleExpr RelOp SimpleExpr</td></tr>
<tr><td>SimpleExpr</td><td>::=</td><td>Term | Term AddOp SimpleExpr</td></tr>
<tr><td>Term</td><td>::=</td><td>SignedFactor | SignedFactor MultOp Term</td></tr>
<tr><td>SignedFactor</td><td>::=</td><td>Factor | Sign Factor</td></tr>
<tr><td>Factor</td><td>::=</td><td>Number | BooleanValue | Identifier | (Expression)</td></tr>
</table>

Lexical rules

<table>
<tr><td>Identifier</td><td>::=</td><td>Letter (Letter | Digit)*</td></tr>
<tr><td>Letter</td><td>::=</td><td>A . . Z | a . . z</td></tr>
<tr><td>Digit</td><td>::=</td><td>0 . . 9</td></tr>
<tr><td>Number</td><td>::=</td><td>Digit Digit*</td></tr>
<tr><td>Sign</td><td>::=</td><td>+ | − | *not*</td></tr>
<tr><td>RelOp</td><td>::=</td><td>= | > | < | <> | >= | <=</td></tr>
<tr><td>AddOp</td><td>::=</td><td>+ | − | *or*</td></tr>
<tr><td>MultOp</td><td>::=</td><td>* | / | *mod* | *and*</td></tr>
<tr><td>BooleanValue</td><td>::=</td><td>*True* | *False*</td></tr>
</table>

Grammar for the language Dada is defined in 21 syntactic production rules and nine additional rules that establish lexical categories. In each rule the strings of symbols to the right of the "::=" sign represent all the possible expansions of the symbol to the left. The symbol " | " is pronounced "or" and separates alternative productions. In the lexical rules an asterisk signifies repetition zero or more times.

Table 1.

Left and right, top and bottom

All practical parsers scan their input from left to right. A limited amount of backtracking or looking ahead may be allowed, but no symbol is added to the parse tree until all the preceding symbols have been parsed. Thus when the end of the input is reached, the tree must represent a complete and valid program.

The subclasses of context-free parsers are distinguished by the sequence in which nodes are added to the tree. The main division is between top-down and bottom-up parsing methods. (The terms employ the usual inverted frame of reference for tree structures, putting the root at the top and the leaves at the bottom.) A

top-down parser begins with the root node and applies production rules to expand each interior node until the leaves are reached; if the terminal symbols hung on the leaves match the complete input string, the program has been parsed. Bottom-up parsing begins with the leaves and tries to build a superstructure of higher level nodes; the parse is successful if the root node is reached.

There are still more choices to make. At any node having more than one child, the parser must decide which branch of the tree to explore first. In applying the production *Term ::= Factor MultOp Term* a leftmost-first parser would expand *Factor* and then *MultOp* and finally *Term*, whereas a rightmost-first parser would deal with the symbols in the opposite order. Current fashion favors bottom-up, rightmost-first parsers. They accept a large class of grammars and tend to be highly efficient. Furthermore, widely available programs automatically generate a parser of this type from a grammar specification. The best known of the programs is Yacc (Yet Another Compiler Compiler), written by Stephen C. Johnson of AT&T Bell Laboratories.

Although creating a bottom-up, rightmost-first parser is easy with power tools such as Yacc, it is difficult to do by hand. In writing a compiler for Dada I have therefore taken the opposite path of top-down, leftmost-first parsing. The technique I chose is the one known as recursive-descent. It is by no means the most efficient method, but it offers a remarkable transparency. You can see straight through the structure of the parser to the underlying grammar.

If a grammar is to be suitable for recursive-descent parsing, the production rules must be free of left-recursion. Because the parser follows a leftmost-first convention, a rule of the form *S ::= Sx* leads into an infinite loop. The parser creates a node labeled S and then, expanding the leftmost symbol of the production, appends a child node also labeled S. It then expands the leftmost symbol and continues adding S nodes indefinitely.

Eliminating left-recursion often entails introducing new nonterminal symbols and epsilon productions, which yield the empty string. The grammar for Dada given in Table 1 is written without left-recursion. The symbol on the left side of a rule is never the first symbol on the right side.

Structure of a compiler

Translation, which is the basic task of a compiler, cannot be done by merely looking up synonyms in a bilingual dictionary. The translator must take in the source text, build an internal model of its structure and meaning, then create a new text in the target language based on the model.

Most compilers break the process into at least four phases. The first phase is lexical analysis, which assembles the source text into the fundamental units called tokens. In Dada tokens include numbers, the names of variables and procedures, keywords such as *begin* and *end*, and various signs and marks of punctuation. The lexical analyzer, or scanner, reads a stream of characters and emits a stream of tokens.

Parsing is the next phase. The parser's input is a stream of tokens and its output is a tree encoding the syntactic structure of the program. The tree organizes the tokens into higher level groups, such as expressions, statements, and procedures. For example, the sequence of tokens $X + 1$ would yield a node labeled + with two children labeled *X* and 1.

Semantic analysis, the third phase, assigns meaning to the nodes of the tree. But what does "meaning" mean in a computer that lacks consciousness and the capacity for understanding? A pragmatic answer will do for present purposes: the meaning of a program is the series of actions elicited when the program is executed. The task of the semantic analyzer is to interpret the abstract symbols manipulated by the parser as a prescription for action.

In many cases the semantic analysis is nothing more than a change in point of view. When the parser reads the expression $X + 1$, it installs three tokens in nodes of the parse tree without attaching any significance to them. They are mere markers, empty of content. The semantic analyzer views the same tokens as instructions to add 1 to the value of the variable *X*. No change is made to the structure of the tree since the programming language itself can express this meaning as well as any other notation would.

One job often given to the semantic analyzer is type checking. If *X* is a floating point variable and the literal value 1 represents an integer, the semantic analyzer would be expected to detect the type mismatch. In many languages it would automatically insert an additional node in the tree calling for a type conversion.

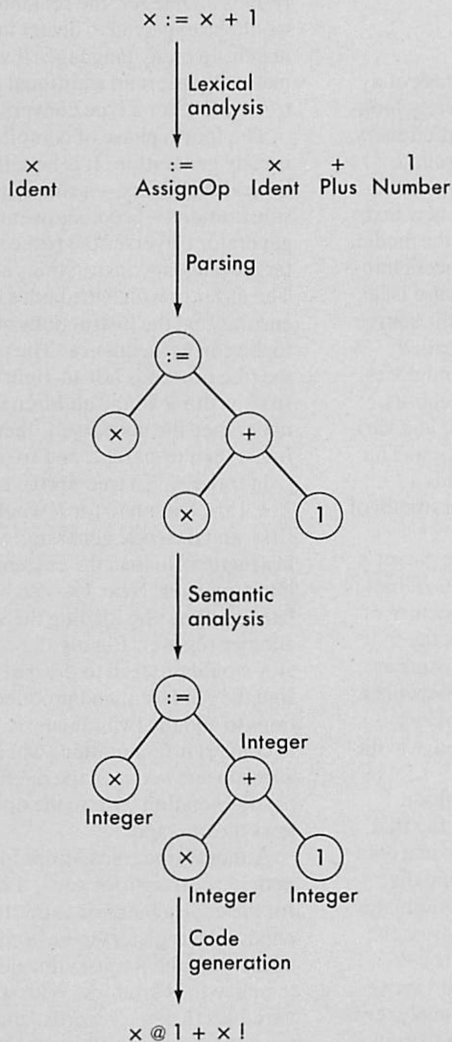
The fourth phase of compiler operation is code generation. It is here that the bilingual dictionary—a simple table of substitutions—becomes useful. The code generator traverses the parse tree, issuing target-language instructions at each node. The order in which the nodes are visited ensures that the instructions are generated in the correct sequence. The traversal of the tree proceeds left-to-right and depth-first, visiting all the children of a given node, then the node itself, then its siblings, then its parent, and so on.

In traversing a tree for the expression $X + 1$ the leaf node for *X* would be reached first, and the code generator would issue instructions to load the current value of *X* into a register. Next 1 would be translated, perhaps by loading the value into another register. Finally the code generator would proceed to the parent node, find the plus sign, and produce instructions to add the two registers. The effect is to convert infix notation (where the operator appears between the operands) to postfix notation (where the operator follows the operands).

Although the translations made at some nodes are straightforward, there is more for the code generator to do. It must choose the registers to be used in each operation, and it must allocate space in memory for variables. With structured variables (arrays, records, and the like) the allocation algorithms can become fairly elaborate, and local variables complicate the process still further. Control-of-flow statements bring another burden: at any branch point in the program logic, the code generator must calculate the target address for a jump instruction.

One way to cope with the complexity of code generation is to subdivide the task. Many compilers generate a form of intermediate code, made up of instructions for a fictitious machine with a regular archi-

Four phases of compilation



Phases in the operation of a compiler transform source text into object code. Lexical analysis breaks the stream of text into the fundamental units called tokens. Parsing assembles the tokens into a tree structure. Semantic analysis attributes meaning to the symbols at the nodes of the tree. Finally code generation creates a program of equivalent meaning in the target language (in this case Forth).

Figure 1.

tecture. A subsequent phase translates the intermediate code into machine-language instructions, or a small interpreter executes the intermediate code on a real processor. An example of this strategy is the P-system developed at the University of California at San Diego.

The final phase of compiling is optimization. The output of a typical code generator is very inefficient, and there is no limit to the effort that can be put into improving it. When the job is taken seriously, the optimizer may well be the largest component of the compiler. The Dada compiler goes to the opposite extreme: the optimizer is omitted entirely.

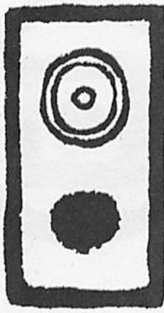
Passes and modules

Perhaps the most obvious way of organizing a compiler is to make each phase an independent module or even a separate program. The scanner reads the source text (a file of characters) and writes a file of tokens. The parser reads the file of tokens and produces a third file of linked records that encode the structure of the parse tree. The subsequent phases also communicate by means of temporary files. Each trip through the program is called a pass.

For the Dada compiler I have taken a different approach, in which all the phases are condensed into a single pass. The modules are organized as a bucket brigade; they process small units of program text and pass them on to the next module in line. The scanner, for example, reads just enough characters to assemble a single token, and then hands it on to the parser.

A major advantage of the one-pass compiler is that no explicit representation of the parse tree is needed. The operation of the parser itself can be interpreted as a traversal of the tree. Each time a procedure is called, the parser effectively moves down one level in the tree, to a child node; when the procedure returns, the parser climbs back up the tree to the parent node. Semantic analysis and code generation can be done during this traversal rather than in separate passes over the tree.

The scanner, parser, semantic analyzer, code generator, and optimizer are the "mainstream" modules of a compiler. Their sequential actions transform the



program text into object code. There are other modules as well. They do not act directly on the stream of text, but they are no less essential.

The importance of an error-handling routine should be readily apparent. Most of the time, for most programmers, what a compiler produces is not object code but error messages. The error handler in the Dada compiler is the simplest one I could devise. When it detects an error, it prints a message and halts the program.

A run-time library is needed to complement the facilities of most programming languages. Input and output statements, for example, require dozens or even hundreds of machine-language instructions; rather than generate all this code for each statement, the compiler calls a preassembled routine. The library procedures can be appended to the object code by the compiler itself or by a separate linker.

I have left to the end of this catalogue the data structure at the heart of many compiler operations: the symbol table. It is the *deus ex machina* that enables the compiler to enforce rules the grammar cannot describe.

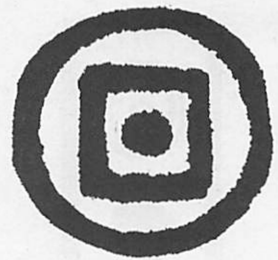
A context-free grammar can balance pairs of symbols, such as nested parentheses, but it cannot match arbitrarily long strings of symbols arbitrarily far apart in the source text. In particular, it cannot match a reference to a variable with the corresponding declaration. Verifying declarations requires a context-sensitive grammar, with multiple symbols on the left side of production rules. Because Dada requires variables to be declared, it is not a fully context-free language.

Rather than complicate the grammar with context-sensitive productions, the almost universal practice in compiler writing is to collect context-sensitive information in a symbol table. The table records information about all the identifiers, or named elements, of a program: variables, procedures, and so on. The parser consults the table to make certain all identifiers have been properly declared. The semantic analyzer relies on it for type checking, and the code generator finds addresses there.

Structure of the scanner

Lexical analyzer, or scanner, is constructed as a finite-state automaton. Each character that can begin a token leads to a distinct state. States enclosed by a double line are accepting states, which the scanner enters when it has recognized a token (or an error). The open-box symbol represents any character not individually specified.

Figure 2.



Useful as the symbol table is, from a theoretical and aesthetic point of view it rather mars the elegance of a compiler's design. Without the symbol table the compiler is a purely sequential machine, climbing up and down the parse tree one node at a time and carrying nothing with it as it moves. All the information needed is encoded in the tree itself. The symbol table introduces new linkages between arbitrarily distant nodes. Connecting declarations to later invocations obscures the spare, planar form of the parse tree behind a topological knot of extraneous links.

The Dada compiler

Writing a compiler is a trilingual project. It involves a source language, a target language, and an implementation language (the one in which the compiler itself is written). In this case the source language is Dada and the implementation language is Borland International's Turbo Pascal. The selection of a target language raises difficult questions. If assembly or machine language were chosen, the code generator would overwhelm the rest of the program. Furthermore, the output of the compiler would be intelligible only to those who happen to know the instruction set of the processor chosen.

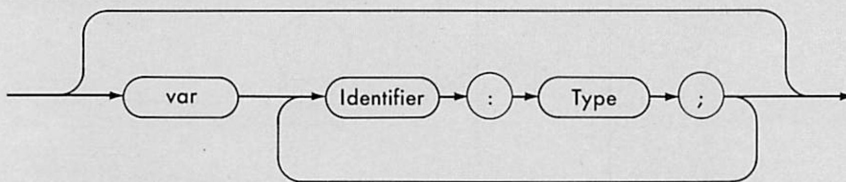
Intermediate code is easier to produce and understand, but it can be executed only with an interpreter. Thus it would be awkward to demonstrate that the compiler actually works. The compromise I have reached is to generate a form of intermediate code recognized by a familiar and widely available interpreter. The output of the Dada compiler consists of "words" in the programming language Forth. This intermediate code should be accepted (perhaps with a little fiddling) by any Forth interpreter. Since there are now processors for which Forth is the assembly language, the Dada compiler might also be considered a native-code compiler for one of those machines.

The place to begin in describing the compiler is the scanner, or lexical analyzer. Figure 2 is a diagram of its structure. It is a finite-state automaton, a logic network without auxiliary memory, with an accepting state for each token or class

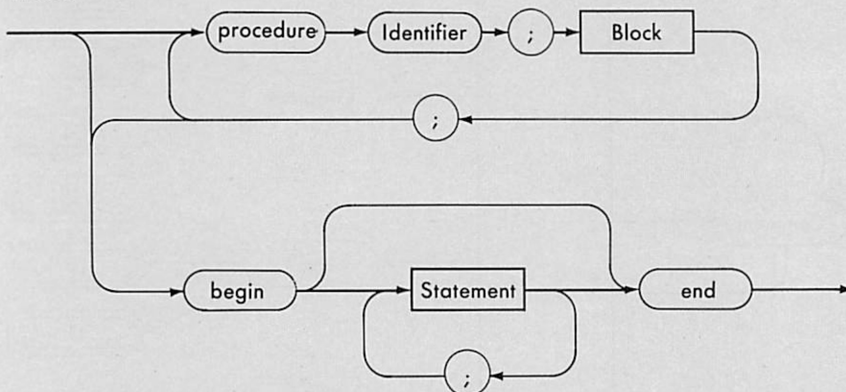
Dada syntax

Program

Declarations



Block



Statement

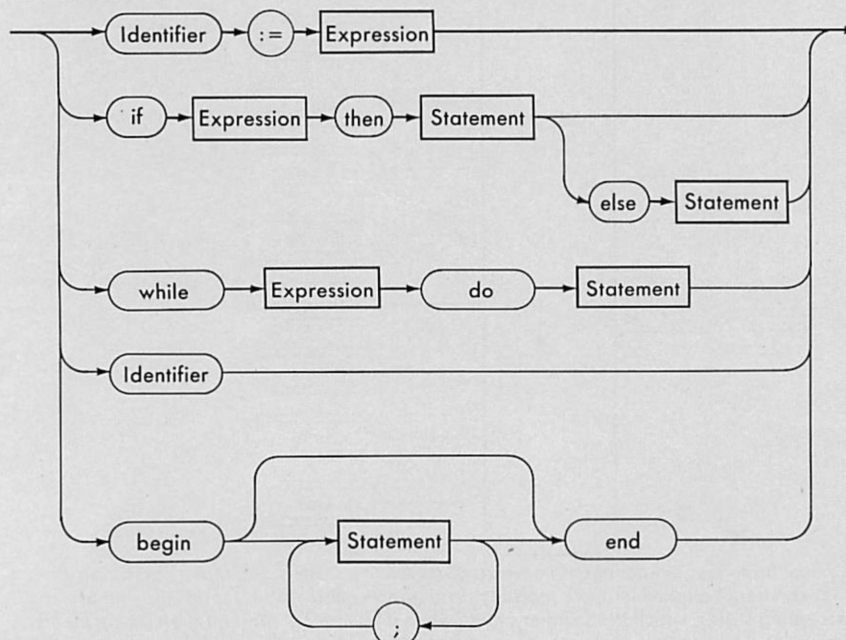


Figure 3. *(Continued on following page)*

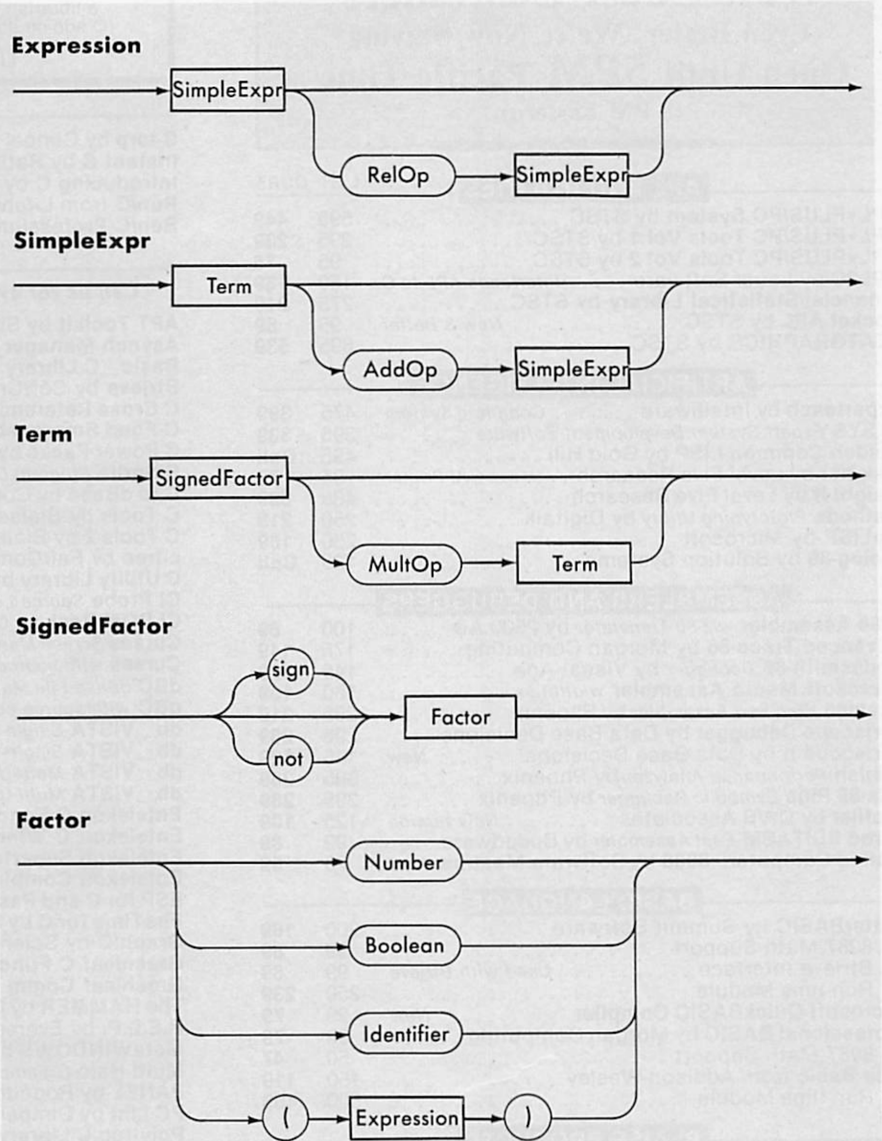
of tokens. The automaton is implemented in software by means of a *case* statement with a clause for each character that can begin a token.

For one-character tokens such as "+" and ";" the clause is simple: the token is recognized immediately. A few other tokens require two steps, for instance to distinguish a colon from the assignment operator ":=". Numbers and identifiers in Dada can be of unlimited length (although they are truncated in the symbol table). A number is any sequence of digits; an identifier is any sequence of letters and digits that begins with a letter. These patterns are identified by *while* loops.

The definition of a Dada identifier also encompasses all the keywords of the language, such as *if, begin,* and *end.* The scanner must recognize these words as individual tokens and not confuse them with named variables or procedures. One way of identifying them is by creating a tree of states for each keyword. In recognizing *begin* the scanner would pass through states for *b, e, g,* and so on. There is an easier way. The scanner initially classifies any string of letters as an identifier and then checks it against a list of keywords; only if a match is not found is the code for an identifier returned.

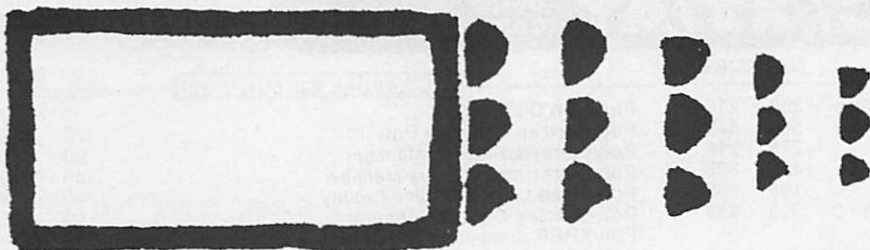
Although lexical analysis is the simplest phase of compilation, the design of the scanner raises questions that are not answered in the grammar. Is the language to be case-sensitive, or do *FOO, Foo,* and *foo* all refer to the same object? How are whitespace characters, such as blanks, tabs, and carriage returns, to be treated? What about comments in the source text?

It is possible to be vague about these issues in designing a language but not in writing a compiler. I have generally followed the example of Pascal. The scanner converts all alphabetic characters to upper case, and so case is ignored. Whitespace is allowed between any tokens but is required only to separate identifiers, keywords, and numbers. A comment is allowed anywhere a whitespace character could appear. A left brace signals the start



Syntax diagrams derived from the grammar define the flow of control through a Dada program. The structure of the program as a whole is given by the first diagram: a program must begin with the keyword *program* followed by an identifier and a semicolon. Control then passes to the Declarations diagram and the Block diagram. Any sequence of tokens corresponding to a path through the diagrams is a valid Dada program.

Figure 3. *(Continued from preceding page)*



```
procedure ParseProgram;
 procedure ParseVariables;
  begin
   if TK.Code = VarSym then
    begin
     GetTK;
     repeat
      if TK.Code <> Ident then Error(XIdent); GetTK;
      if TK.Code <> Colon then Error(XColon); GetTK;
      if not (TK.Code in TypeSet) then Error(XType); GetTK;
      if TK.Code <> Semi then Error(XSemi); GetTK;
     until (TK.Code in [ProcSym,BeginSym]);
    end;
  end;
 procedure ParseBlock;
  procedure ParseStatement;
   var IdentPtr : SymPtr;
   procedure ParseExpression;
    procedure ParseSimpleExpr;
     procedure ParseTerm;
      procedure ParseSignedFactor;
       procedure ParseFactor;
        begin                                          { ParseFactor }
         case TK.Code of
          TrueSym, FalseSym, Number, Ident : GetTK;
          LeftParen : begin
                       GetTK; ParseExpression;
                       if TK.Code <> RightParen
                        then Error(XParen); GetTK;
                      end;
          else Error(XFactor);
         end;
        end;

       begin                                     { ParseSignedFactor }
        if (TK.Code in [Plus, Minus, NotSym]) then GetTK;
        ParseFactor;
       end;

      begin                                            { ParseTerm }
       ParseSignedFactor;
       if (TK.Code in MultOpSet) then
        begin GetTK; ParseTerm; end;
      end;

     begin                                       { ParseSimpleExpr }
      ParseTerm;
      if (TK.Code in AddOpSet) then
       begin GetTK; ParseSimpleExpr; end;
     end;
```

Listing 1. *(Continued on following page)*

of a comment, and all subsequent characters are ignored until a right brace is found.

The parser

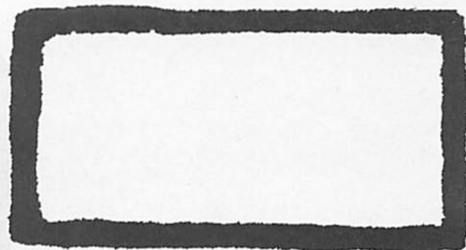
Building a parser is the best way to appreciate the worth of a formal grammar. Comparing the routines of the parser with the production rules of the grammar reveals an obvious one-to-one correspondence.

The transformation of a grammar into a parser can be done directly; that is what Yacc does. When working by hand, however, it is easier to create a series of syntax diagrams based on the grammar and then to build the parser from the diagrams. Where a production rule calls for generating a series of symbols, they are written down in sequence and connected by arrows. Alternative productions correspond to branches in the diagram, and a recursive rule—one that invokes itself—implies a diagram with a loop. A set of syntax diagrams for Dada is shown in Figure 3.

Each of the diagrams is represented by a procedure in the parser. At the top level the procedure *ParseProgram* expects to see the keyword *program* followed by an identifier and a semicolon. It then calls *ParseVariables* to handle any declarations of variables; when that procedure returns, a semicolon is expected. Next there is a call to *ParseBlock*, which analyzes the rest of the program. When *ParseBlock* returns, all that remains is to check for the concluding period.

Some of the other diagrams are more elaborate; *ParseBlock* serves as a good example. It first checks for the keyword *procedure*; if it is present, *ParseBlock* reads an identifier and a semicolon and then calls itself recursively. The new invocation of *ParseBlock* immediately looks for *procedure* again, so that nested declarations can be accommodated. Eventually, however, each invocation of *ParseBlock* must find not another procedure but the keyword *begin*. *ParseStatement* is then called repeatedly until the token *end* is encountered.

The organization of the parsing procedures follows directly from the structure of the language. A Dada program consists of variable declarations and a block; a block consists of procedure dec-



larations and a list of statements; a procedure declaration consists of subsidiary procedures and another block. Accordingly, *ParseProgram* is the outermost routine, and both *ParseVariables* and *ParseBlock* are nested within it. *ParseBlock* in turn encloses *ParseStatement*, and there is a further hierarchy of nested routines from *ParseExpression* down to *ParseFactor*.

All the routines work by a common mechanism. At any moment the parser can take only two possible actions. If a terminal symbol is expected, the current token is examined. If it is one allowed by the grammar at this point in the program, the scanner is called to get the next token and the parser moves on; otherwise an error message is issued. If the symbol expected is a nonterminal, the parser merely calls the appropriate routine.

If a parser for a context-free language is a pushdown automaton, where in this scheme of operations is the pushdown stack? It is present but well hidden. It is the return-address stack maintained by any executing Pascal program, in this case the Dada compiler itself. One of the advantages of the recursive-descent technique is that it relieves the programmer of responsibility for managing the stack.

Table 2 shows the parser in action, analyzing the expression 3*(4+5). According to the syntax diagrams, an expression consists either of a simple expression or of two simple expressions separated by a relational operator. Thus to identify the input as an expression, the first step is to look for a simple expression. A simple expression in turn begins with a term which begins with a signed factor, which consists of an optional sign followed by a factor. Finally a factor, at the bottom of the hierarchy of program structures, can be a number, a Boolean value, an identifier, or a parenthesized expression.

Routines to recognize each of these elements are activated in a cascade of procedure calls. *ParseExpression* calls *ParseSimpleExpr* which calls *ParseTerms* which calls *ParseSignedFactor* which calls *ParseFactor*. At last, after five nested calls, the first token of the input (the number 3) is identified as a factor.

```
   begin                                        { ParseExpression }
    ParseSimpleExpr;
    if (TK.Code in RelOpSet) then
     begin GetTK; ParseSimpleExpr; end;
   end;

  begin                                          { ParseStatement }
   case TK.Code of
    BeginSym : begin
                GetTK; while TK.Code <> EndSym do
                 begin
                  ParseStatement;
                  if not (TK.Code in [Semi,EndSym]) then
                   Error(XSemEnd);
                  if TK.Code = Semi then GetTK;
                 end;
                GetTK;
               end;
    IfSym    : begin
                GetTK; ParseExpression;
                if TK.Code <> ThenSym then Error(XThen); GetTK;
                ParseStatement;
                if TK.Code = ElseSym then
                 begin GetTK; ParseStatement; end;
                end;
    WhileSym : begin
                GetTK; ParseExpression;
                if TK.Code <> DoSym then Error(XDo); GetTK;
                ParseStatement;
               end;
    Ident    : begin
                IdentPtr := Find(TK.Name);
                if IdentPtr^.Class = Variable then
                 begin
                  GetTK; if TK.Code <> AssignOp then
                   Error(XAssgn);
                  GetTK; ParseExpression;
                 end
                else GetTK;
               end;
    else       Error(XStmt);
   end;
  end;

 begin                                               { ParseBlock }
  while TK.Code = ProcSym do
   begin
    GetTK; if TK.Code <> Ident then Error(XIdent);
    GetTK; if TK.Code <> Semi then Error(XSemi);
    GetTK; ParseBlock;
    if TK.Code <> Semi then Error(XSemi); GetTK;
   end;
  if TK.Code <> BeginSym then Error(XBegin); GetTK;
  while TK.Code <> EndSym do
   begin
```

Listing 1. *(Continued on following page)*

As each of these calls is made, a record of the calling routine is pushed onto the return-address stack. When *ParseFactor* completes it work, therefore, it returns to the point from which the call was made in *ParseSignedFactor.* That routine is also finished, and it returns to *ParseTerm.* There is now a choice between two paths of execution. If the next token is anything other than a multiplying operator, *ParseTerm* simply exits. In fact the next token is "*", and so the other path is taken. The token is consumed and *ParseTerm* is called again, recursively, to initiate another cascade back down to *ParseFactor.*

The gyrations of the parsing routines eventually trace out the tree structure for the expression. At the root of the tree is the "*" sign, and under it are nodes for "3" and for "(4 + 5)." The latter node has another subtree under it, with leaf nodes for the factors "4" and "5." The parser has no need to keep a diagram of these relations because they are effectively recorded in the sequence of return addresses on the stack. Each call to a routine creates a new child node on the tree, whose parent is indicated by the address at the top of the stack. No matter how deeply nested the routines become, the parser can always find its way back to the correct parent node by following the trail of stacked addresses.

Ambiguities

The Dada compiler employs a form of the recursive-descent technique known as predictive parsing, which forbids looking ahead at the next input symbol or backtracking to earlier symbols. In all circumstances the parser must be able to predict, based on the current token alone, which of the paths available to it should be followed.

If a language is to be suitable for predictive parsing, the grammar and the syntax diagrams must have a particular form. At any point where a diagram branches, the first token expected along each branch must be unique. Consider the syntax diagram for a Dada statement. There are five branches, corresponding to an assignment statement, an *if* statement, a *while* statement, a procedure call, and a compound statement bracketed by *begin* and *end.* Three of the branches cause no difficulty. If the first token in a statement is *if, while,* or *begin,* the parser knows which path to follow. The assignment statement and the procedure call, however, both start with an identifier. Thus it seems Dada does not qualify for predictive parsing.

Three solutions to this problem present themselves. One could change the language, perhaps requiring a procedure to be invoked by the keyword call, as in PL/I. One could change the grammar, introducing distinct tokens for procedure names and variable names. Or one could change the parser, allowing it to look ahead to see if the next token is an assignment operator.

I have adopted a fourth solution: cheating. At the time an identifier is declared, a symbol-table entry is created, recording (among other things) whether the identifier names a variable or a procedure. When the same identifier appears again, it is a simple matter to check its classification.

A parser can construct a unique tree for a program only if the grammar is unambiguous. What assurance can be given that the Dada grammar meets this condition? As it happens, the grammar does include an ambiguity, common to many programming languages, known as the dangling-*else* problem. In a nested conditional statement, such as if *A then if B then C else D,* the grammar does not state which *then* the *else* is to be associated with. Is *D* executed when *A* is false or when *B* is false? The two possibilities correspond to different tree structures.

The dangling-*else* problem is solved by making an arbitrary choice. By convention each *else* is associated with the closest preceding *then* that does not already have an *else.* The parser needs no modification to enforce this rule. The sequence of procedure calls and returns always generates the correct tree.

The grammar for Dada fails to settle a number of other issues that must be resolved in writing the compiler. The grammar implies that procedures must be declared, but it does not say they have to be declared before they are first called. The decision to distinguish between assignment and call statements by consulting the symbol table effectively requires the declaration to come first.

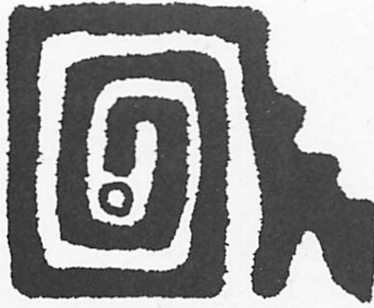
What about a procedure called from within its own declaration, that is, a recursive procedure? Nothing in the grammar forbids recursion, and the Dada compiler will accept recursive calls without complaint. The trouble is, a Forth

```
      ParseStatement;
      if not (TK.Code in [Semi,EndSym]) then Error(XSemEnd);
      if TK.Code = Semi then GetTK;
     end;
    GetTK;
   end;

  begin                                     { ParseProgram }
   if TK.Code <> PgmSym then Error(XPgm); GetTK;
   if TK.Code <> Ident then Error(XIdent); GetTK;
   if TK.Code <> Semi then Error(XSemi); GetTK;
   ParseVariables; ParseBlock;
   if TK.Code <> Dot then Error(XDot);
  end;
```

Dada parser has a procedure for each of the syntax diagrams in Figure 3. The nesting of the procedures reflects the structure of the grammar in that a statement is subsidiary to a block, an expression is subsidiary to a statement, and so on. In the form shown here the parser acts as a recognizer, a machine that accepts any valid stream of tokens but ignores their semantic content. In the complete compiler statements for type checking and for code generation are interleaved with the parsing routines.

Listing 1. *(Continued from preceding page)*



interpreter will not execute the recursive definition; it causes a run-time error.

Another issue the grammar does not address is the redefinition of keywords. What if some programmer wants to declare a variable named *if* or a procedure named *begin*? Some languages allow such shenanigans and some do not. Dada is in the latter category because of the way keywords are detected.

Semantics

The scanner and the parser together make a recognizer for Dada programs. They accept only syntactically valid strings of symbols. Unfortunately, the category of syntactically valid strings includes many that are semantically deviant. The expression $X + Y$ has impeccable syntax, but if X is an integer and Y is a Boolean variable, it has no clear meaning. What is the value of 8 + True?

The type checking done in the course of semantic analysis improves the odds that a program will be accepted only if it makes some sense. In the Dada compiler there is no separate module for type checking; it is done by statements intercalated into the parser. Most of them refer to the symbol table, which should therefore be discussed in greater detail.

The symbol table is a linked list of records with five fields. The name field stores the first 31 characters of an identifier's name (an arbitrary length limit). The class field indicates whether an identifier is a variable or a procedure, and the type field further classifies variables as integer and Boolean values. A scope field holds a number that represents the nesting depth of procedures. The final field is a pointer to the next record.

Symbol-table entries are created by a procedure named *Declare*, which is called with an identifier as one of its arguments. *Declare* checks to see if the identifier is already present in the table; if not, it is installed at the head of the list. The complementary function, *Find*, searches the list for an identifier and returns a pointer to it.

The one subtlety in the handling of the symbol table has to do with nested procedures, which are intended to be local to the block in which they are declared. In other words, a procedure declared inside another procedure should "disappear" when the compiler passes beyond the end of the enclosing block. This is accomplished by monitoring the scope field in each symbol-table record. A procedure named *Blot*, called at the end of a block, removes all entries that can never legally be accessed again.

Type checking seems simple in principle, but it can become surprisingly messy. It is easy enough to verify the semantics of a statement such as $X := Y$: just look in the symbol table to make sure X and Y are of the same type. Where difficulties arise is in assigning types to expressions. In analyzing $X + Y * Z$ the types of the variables must be checked, then a result type must be determined according to combinatorial rules.

Such rules exist, but they do not form a coherent system. In most cases the operands and the result are all expected to be of the same type, but relational operations are an exception: they always yield a Boolean result. Whether logical operators (*and, or, not*, and so forth) can be applied to numerical values is a matter of opinion and taste. With languages that have a wider selection of data types the issues get stickier. Can strings be compared or added? What about records and pointers?

The type checking done in the Dada

Parsing an arithmetic expression

Active procedure	Remaining input	Action
Expression	3 * (4 + 5)	
SimpleExpr	3 * (4 + 5)	
Term	3 * (4 + 5)	
SignedFactor	3 * (4 + 5)	
Factor	3 * (4 + 5)	Gen('3')
SignedFactor	* (4 + 5)	
Term	* (4 + 5)	HoldMultOp := '*'
Term	(4 + 5)	
SignedFactor	(4 + 5)	
Factor	(4 + 5)	
Expression	4 + 5)	
SimpleExpr	4 + 5)	
Term	4 + 5)	
SignedFactor	4 + 5)	
Factor	4 + 5)	Gen('4')
SignedFactor	+ 5)	
Term	+ 5)	
SimpleExpr	+ 5)	HoldAddOp := '+'
SimpleExpr	5)	
Term	5)	
SignedFactor	5)	
Factor	5)	Gen('5')
SignedFactor	)	
Term	)	
SimpleExpr	)	Gen(HoldAddOp) {'+'}
SimpleExpr	)	
Expression	)	
Factor		
SignedFactor		
Term		
Term		Gen(HoldMultOp) {'*'}
SimpleExpr		
Expression		

Expression is parsed by a series of nested procedure calls. The process begins when *ParseExpression* calls *ParseSimpleExpr*, which in turn calles *ParseTerm*; ultimately when *ParseFactor* is called, it recognizes and consumes the first token of the input. Actions taken by the compiler include calls to *Gen*, the code generator. The output resulting from the calls is the sequence of symbols 3 4 5 + *.

Table 2.

compiler is not rigorous or exhaustive. Each of the routines in the expression-parsing hierarchy is defined as a function that returns the type of the structure it has parsed. Hence types propagate upward in the tree from the leaf nodes to the main expression node.

The code generator

The last major subunit of the Dada compiler, the code generator, is deceptively simple. It is deceptive because all of the difficult tasks—such as calculating addresses and allocating storage—are done by the Forth target system.

In a pinch, a single *Write* statement would make a serviceable code generator for a Dada-to-Forth translator. I have added a few lines of code to format the output in standard Forth "screens" of 1,024 characters. A subsidiary routine adds a small run-time library to each program. The main components of the library are *Read* and *Write* procedures that provide rudimentary communication with the console.

The real work of translating Dada into Forth is done not by the code-generating routine itself but by the placement of calls to the routine throughout the parser. The calls cause the code generator to define a Forth word for each procedure in the Dada program and another word for the main block.

A Forth definition begins with a colon and the name of the word, followed by the body of the definition and a terminating semicolon. Any previously defined Forth word can be invoked by giving its name. All operations are carried out on values at the top of a stack. Statements and expressions are described in postfix notation.

The operating principles of the code generator can be illustrated by tracing the translation of a small procedure. The body of the procedure consists of a single statement whose effect is to halve the value of a previously declared variable *N*. The Dada statements read as follows:

```
procedure DownStep;
    begin
        N := N/2
    end;
```

When any procedure declaration is reached in a Dada program, the active routine in the compiler is *ParseBlock*. From the initial keyword it deduces that what follows is a procedure. The identifier *DownStep* is then stored in a local variable (*HoldID*), but no code is generated. The parser cannot yet know whether there are further, nested procedure declarations. If there are, object code for them will have to be created first.

When the keyword *begin* is parsed, *ParseBlock* is called recursively with the contents of *HoldID* passed as a parameter. The body of *DownStep* has now been reached, and so code generation can begin. A colon is written to the output file, followed by the passed parameter (namely, the string *DownStep*). Thus a new Forth definition is begun.

ParseBlock now calls *ParseStatement*, the next routine in the hierarchy. The first token of the remaining input is found to be an identifier, and a search of the symbol table reveals it represents a variable; the statement must therefore be an assignment. The string *N* is stored in another instance of the local variable *HoldID*, and *ParseExpression* is called to deal with the right side of the statement.

ParseExpression sets in motion the cascade of expression-handling routines, which ultimately identify the next token, *N*, as a factor. At this point the first instructions for the body of the procedure can be issued. Since the factor is a reference to a variable, the appropriate action is to copy the value of the variable from its location in memory onto the top of the stack. The code for doing this consists of the variable name *N* followed by the Forth word "@," pronounced "fetch." Hence the symbols *N@* are appended to the output file.

The compiler now makes its way back upward through the series of expression-handling functions as far as *ParseTerm*, which recognizes the division sign as a member of the set of multiplying operators. Code for division is not generated, however, since expressions in Forth must be given in postfix form. The "/" token is merely stashed in another local variable, *HoldMultOp*.

ParseTerm now calls itself, so that two instances of the function are active, and the compiler thereupon descends again to the level of *ParseFactor*. Here the second operand, "2," is recognized. Because it is a literal numeric value, which requires no memory reference, the token alone, without accompanying symbols, is output.

On returning to the first invocation of *ParseTerm* the compiler retrieves the

```
program Hailstone;
var
 N   : integer;
 Odd : boolean;
procedure NextTerm;
 procedure CheckOdd;
  begin
   if (N mod 2 = Ø) then
    Odd := False else
     Odd := True
  end;
 procedure DownStep;
  begin
   N := N/2
  end;
 procedure UpStep;
  begin
   N := 3*N+1
  end;
 begin   { NextTerm }
  CheckOdd;
  if Odd then UpStep else
   DownStep
 end;
begin    { main program }
 ReadLn N;
 while N > 1 do
  begin
   WriteLn N;
   NextTerm
  end;
end.
```

Sample Dada program tests the operation of the compiler. The algorithm has been broken up into several small procedures to exercise the parsing routines.

Listing 2.

pending operator from the variable *HoldMultOp* and sends it to the code generator. The expression $N/2$ has now been fully parsed, and the corresponding Forth words have been issued. Execution of these words will leave the new value of N on the top of the Forth stack. It remains for the parser to return to *ParseStatement*, where the first occurrence of N, from the left side of the assignment statement, has been sequestered. The goal now is to store the value on the top of the stack at the address represented by N. The necessary Forth instructions are *N !* (the exclamation point is pronounced "store"), and they are duly issued.

There is one further addition to the object code. When *ParseBlock* reads the *end* keyword, signifying that there are no more statements in the block, it calls for the generation of a semicolon to mark the end of the Forth definition. The complete translation is:

```
: DownStep N @ 2 / N ! ;
```

It defines a new Forth word called *DownStep*. Whenever it is executed, it retrieves the current value of the variable N, divides it by 2, and stores the new value at the address reserved for N.

Pieces of the puzzle

Throughout this series of articles I have been probing the curious, murky relations between syntax and semantics, form and content, recognition and understanding. A sentence or a program can be grammatically flawless and yet signify nothing. Indeed, it is easy to create entire languages that convey no meaning. When meaning does emerge, then, where does it come from?

One approach to this question views language as a jigsaw puzzle. Grammar determines the shapes of the pieces and how they are to be assembled. From the standpoint of grammar alone, however, the pieces are empty markers to be shuffled about on the table top. Meaning exists only in the picture printed on the surface of the puzzle. For the picture to make sense the pieces must be assembled correctly, and yet there is no necessary correspondence between the pattern of the picture and the pattern of the jigsaw cuts.

What is troubling about this metaphor is that it makes the relation between grammar and meaning completely arbitrary. A given picture could be cut up into many different sets of pieces, and a given cutting of the puzzle could have any picture whatever printed on it. This is the world of Humpty Dumpty, where words mean whatever we want them to.

The arbitrary mapping from syntax to semantics is all too apparent in the Dada compiler. The scanner and the parser have sound theoretical underpinnings; one can give algorithms for their construction. The semantic tasks of type checking and code generation, on the other hand, are done by ad hoc procedure calls hung on the parse tree like Christmas ornaments.

```
Ø  VARIABLE N                          ( Code for program Hailstone )
Ø  VARIABLE ODD
:  CHECKODD N @ 2 MOD Ø = IF FALSE ODD ! ELSE TRUE ODD ! THEN ;
:  DOWNSTEP N @ 2 / N ! ;
:  UPSTEP 3 N @ * 1 + N ! ;
:  NEXTTERM CHECKODD ODD @ IF UPSTEP ELSE DOWNSTEP THEN ;
:  HAILSTONE N READ BEGIN N @ 1 > WHILE N WRITE NEXTTERM REPEAT ;
;S

The output of the compiler consists of "words" to be executed by
a Forth interpreter.
```

Listing 3.

Artwork: Anne Doering

Turbo Pascal Science, Engineering and Data Acquisition Tools

Save time and money by incorporating these accurate, pretested Turbo Pascal procedure libraries into your own custom application programs. Developed by experts in each field, these tools can save hundreds of hours of research and program development time. All tools are supplied on IBM PC compatible diskettes and come complete with detailed documentation and source code listings. All of the tools are compatible with Turbo and Turbo-87 Pascal Rev. 3.0 and higher. Example programs on disk make learning how to use the procedures quick and easy.

General Science and Engineering Tools (IPC-TP-006)
Include these procedures in your Turbo Pascal application programs for general statistics, multiple regression, curve fitting, integration, FFT's, file transfers to Lotus 1-2-3, solving simultaneous equations, matrix math, linear programming, data smoothing and graphics (line plots, bar graphs, scatter plots, semi-log graphs, log graphs and windows). **$69.95.**

Data Acquisition and Control Tools (IPC-TP-007)
The Turbo Pascal Data Acquisition and Control Tools package supports the IBM DACA (Data Acquisition and Control Adapter), Cyborg Isaac 41I and Cyborg Isaac 91I. Analog inputs can be sampled at up to 18K samples per second. Procedures are also supplied for analog output, digital input and output, thermocouple linearization, PID control, real-time graphics (bar graphs and line plots) and FFT's. Menu-driven example programs for data logging into Lotus 1-2-3 files, high speed data acquisition, process control and real-time graphics let the user start acquiring and analyzing analog data immediately. The data acquisition and control package requires the IBM Data Acquisition and Control Adapter Programming Support software. **$94.95.**

To order or for free information call or write:
Quinn-Curtis (617) 969-9343
Software Division MasterCard and
7 Fredette Rd. VISA accepted
Newton Centre, MA 02159

Add $5.00 for shipping and handling in US and Canada and $10.00 for shipping and handling outside US and Canada. Mass. residents add 5% sales tax.

PC Software for Scientists and Engineers

Turbo Pascal is a registered trademark of Borland International, Inc. IBM and DACA are registered trademarks of International Business Machines. Lotus 1-2-3 is a registered trademark of Lotus Development Corporation. Isaac 41I and 91I are registered trademarks of Cyborg Corporation.

CIRCLE 59 ON READER SERVICE CARD

Personal REXX for the IBM PC

★ Interpreter for the full REXX language, including all of the standard REXX instructions, operators, and built-in functions
★ Sophisticated string manipulation capabilities
★ Unlimited precision arithmetic
★ Direct execution of DOS commands from REXX programs
★ Built-in functions for DOS file I/O, directory access, screen and keyboard communication, and many other PC services
★ Compatible with VM/CMS version of REXX
★ Uses include:
- Command programming language for DOS
- Macro language for the KEDIT text editor
- Can be interfaced by application developers with other DOS applications, written in almost any language

Mansfield Software Group
P. O. Box 532
Storrs, CT 06268
(203) 429-8402

$98 plus $3 shipping until 1/1/86
$125 plus $3 shipping after 1/1/86
MC, VISA, COD, PO, CHECK

CIRCLE 38 ON READER SERVICE CARD

The Best 68000 & 32000 Compilers

GUARANTEED.

OEMs: You get an unconditional 30 day money back guarantee that our compilers generate faster and smaller code than any other corresponding 68000 or 32000 industry standard compiler.

All compilers include 1 year of maintenance and updates.

C – FORTRAN 77 – Pascal

Available NOW for:

Motorola 68000, National 32000

UNIX 4.2 BSD, UNIX System V

Green Hills Software

425 E. Colorado Blvd., Suite 710
Glendale, California 91205
(818) 246-5555

Leaders in stamping out vaporware and puffware.

UNIX is a trademark of AT&T Bell Laboratories

CIRCLE 37 ON READER SERVICE CARD

There is no semantic specification of the language comparable to the syntactic specification given in a set of production rules.

Schemes for semantic specification do exist. In one such scheme, called an attribute grammar, a semantic rule is associated with each syntactic production rule. Attributes, which can be thought of as slots for semantic values, are attached to each node of a parse tree. Whenever a production is invoked during the parsing of a program, the associated semantic rule is executed to fill in the attribute slots of the current node. The rule calculates the attribute values based on the values stored in other nodes. The values can include data types, fragments of object code, or any other computable information.

Attribute grammars and other inventions of formal semantics promise to improve the methodology of compiler design. An open question is whether they can also improve understanding. A grammar is not only a useful tool but also an aid to comprehension. You can look at a few dozen production rules and see in them an infinite variety of program structures. As yet no semantic rules offer a comparable grasp of the infinite range of program meanings. ∎

References

Aho, Alfred V., and Jeffrey D. Ullman. *Principles of Compiler Design.* Reading, Mass.: Addison-Wesley Publishing Co., 1977. [Known as the "dragon book," from the illustration on the cover. Gives an introduction to the theory, with emphasis on bottom-up parsers and automated compiler-writing tools.]

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Reading, Mass.: Addison-Wesley Publishing Co., 1986. [A new dragon book, revised and enlarged, with more material on formal semantics.]

Wirth, Niklaus. *Algorithms + Data Structures = Programs.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1976. [Includes a brief but lucid treatment of the practicalities of compiler design, with a fully worked example.]

Kernighan, Brian W., and P.J. Plauger. *Software Tools.* Reading, Mass.: Addison-Wesley Publishing Co., 1976. [Concludes with a presentation of a RATFOR-to-FORTRAN translator, based on a hand-crafted compiler.]

Amsterdam, Jonathan. "Context-Free Parsing of Arithmetic Expressions." *BYTE.* Aug., 1985. [A recent expression-parsing program, with source code in Pascal.]

Holub, Allen. "How Compilers Work—A Simple Desk Calculator." *Dr. Dobb's Journal.* Sept. 1985. [Another recent expression parser, with source code in C.]

Brian Hayes is a writer who works in both natural and formal languages. Until 1984 he was an editor of Scientific American, *where he launched the Computer Recreations department.*