Code compilation

There is no shortage of compilers for program translation, but how easy is it to write your own? Martin Davies gives comprehensive details, and presents his own example in Modula-2.

Compiler details

The compiler on page 158 is written in Modula-2, which is a language very similar to Pascal. In addition to the features of Pascal, Modula-2 has the concept of a module. A module is a collection of procedures, types and variables. Each module has a definition part, which contains definitions of those procedures, types and variables which can be accessed by other modules; and an implementation part, which contains the actual code. The main module of the program does not need a definition module.

If you want to access a procedure X from module P then your program must contain either the statement:

```
   FROM P IMPORT X;
```

in which case you access X as though it had been declared in the current module:

```
   X; (*Procedure X from module P*)
```

or

```
   IMPORT P;
```

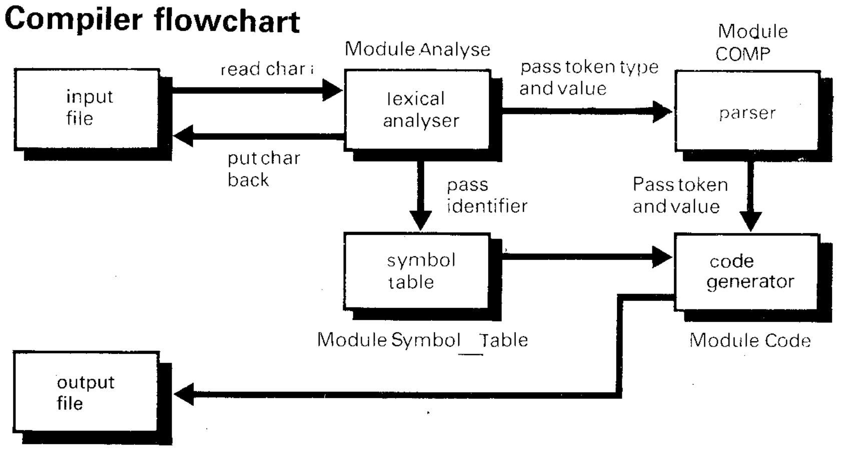
in which case you have to qualify X by prefixing the module name to it:

```
   P.X; (*Access procedure X from module P*)
```

The advantages of modules are that you can compile things separately, and they are type secure as the compiler can check a call against its definition stored in the definition module. Programs can be broken up into logical re-usable units which can be written by different people.

The following chart illustrates the structure of the compiler and the flow of data between modules:

Compiler flowchart



COMP is the main module. It contains the parser which checks that the program is well formed. The parser receives tokens from the lexical analyser, which reads the source file and breaks it up into logical units (tokens). It passes these tokens on to the code generator, which then generates code for them. The symbol table is used by the lexical analyser to store identifier names which are retrieved by the code generator.

The process of translation of programming languages is quite transparent but have you ever wondered what a compiler does? How is it different from an interpreter and why do different compilers produce different code?

What is a compiler?

A compiler is a program which reads in a program in a high-level language and translates it into an equivalent machine code program. This is different to the Basic interpreters which are supplied with most PCs, which read in a program and then perform the actions implied by that program.

Compilers generate a new, equivalent machine code program which means that compiled programs:

- execute faster than interpreted programs because the interpreter has to translate the program as well as performing its function;
- don't require the compiler to be present. This means that one copy of a compiler can generate programs to run on hundreds of machines. An interpreter has to be present for interpreted code to run, so you require as many copies of the interpreter as you have of the program;
- often require less space. An interpreter requires the original program, the interpreter and workspace to run. Compiled code only requires space for the generated program and workspace; and
- can often be linked with programs written in other languages and assembler.

What does a compiler do, and how?

In order to translate a program, a compiler has to break it into individual statements, check that those statements are valid and generate equivalent statements in machine code. To demonstrate this I have produced a very simple compiler in Modula-2 (page 158). This compiles a simple language I have called C--.

Lexical analysis

Breaking up statements is called lexical analysis. The lexical analyser does not need to know what each language token means, just how to tell when it has ended. Our simple language has only three classes of tokens:

- identifiers — a character followed by zero or more alphanumeric characters;
- integers — a digit followed by zero or more digits; and
- special symbols — either single characters, '-', or composite symbols, ':='.

These tokens are analysed by procedure Lex in module Analyse, which reads the next token from the input file. It returns the type of the token and stores its value.

When Lex encounters a digit it

continues to read digits until it reaches the end of the number. It stores the value of the number it has found, sets the type of the token to number and places the character it has read last back on the input stream, as it was not part of the number. This is done by setting UnGetChar to TRUE, which means that the next time Lex is called, that character will be examined before a new one is read.

When Lex encounters an alphabetic character, it has found an identifier. It continues to read alphanumeric characters until it finds a character which cannot be part of an identifier. It then looks for the character in the symbol table. If it is already in the symbol table Lex stores its position and returns its type, otherwise it inserts it into the table, stores its position and returns type identifier.

If the character is not an identifier or a number it must be a special symbol or white space. Lex returns the type of special character and ignores white space.

A lexical analyser performs exactly the same function as the code to check read statements you might have in any simple program.

Storing information

A compiler needs to associate information with identifiers. This is done in a symbol table which stores the identifier name and its associated type. The symbol table contains all the identifiers used in the program. In our example implementation only, two classes of tokens are stored: identifiers, which are added by the lexical analyser to the symbol table; and the reserved words which are added during initialisation of the compiler. In more complicated compilers, the symbol table will also hold information about the scope of the variables and procedures declared.

Module Symbol__Table contains two procedures which manipulate the symbol table:

- Insert (S,T,N), which returns the index of a new entry for string S and token T. N denotes the number of characters to be allocated to store S.
- Lookup(S), which returns the index of the entry for string S or zero if S is not found.

Parsing

By parsing, you check that the program is legal. The parser checks that the tokens occur in the correct order, and from the ordering and the token type calls a procedure to generate code for each language construct. The parsing is done in Module Comp.

If you look at the language definition of C-- on page 156, you will see how the parser was constructed. For each rule: statement, IF statement, REPEAT statement, assignment statement, expression and term, there is an associated procedure. The token returned by the lexical analyser unambiguously determines which procedure is called.

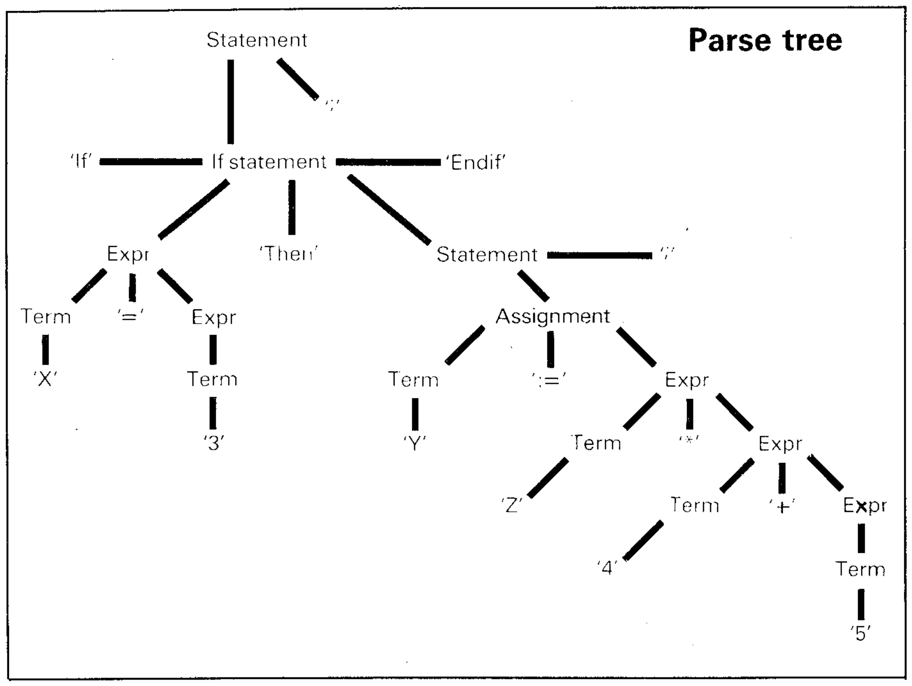
As an example of how the parser checks statements, we can trace the following program fragment in what is called a parse tree (see above).

```
IF X=3 THEN
  Y:=Z*4+5;
ENDIF;
```

The procedure statement looks at the next token; as it is 'IF' it calls procedure IF__STATEMENT. An IF statement is either

(A) IF expression THEN statement ENDIF, or

(B) IF expression THEN statement



Parse tree

```
Source code          Assembler       Comment
BEGIN                __TMP EQ 1000; declare compiler temp
                     A EQ 1002      ; allocate space for A
A:=3332*3+4*5;       LOAD 3332      ; load the register
                     MUL 3          ; register contains 3332*3
                     ADD 4          ; register contains (3332*3)+4
                     MUL 5          ; register contains ((3332*3)+4)*5
                     STORE A        ; assign result to A
                     BERT EQ 1004   ; allocate space for BERT
BERT :=2;            LOAD 2         ; load 2
                     STORE BERT     ; assign it to BERT
IF A = BERT THEN     LOAD A         ; load register
                     XOR BERT       ; clears register if BERT=FRED
                     JNZ L1:        ; JUMP to L1 if false
A:= A*(2/BERT)       LOAD A         ; load register
                     PUSH           ; store partial result on stack
                     LOAD 2         ; load start of bracketed expression
                     DIV BERT P     ; 2/BERT is now in the register
                     POP __TMP      ; pop stack into temp (A)
                     MUL __TMP      ; (2/BERT) *A
                     STORE A        ; assign it to A
ELSE                 JMP L2         ; goto the end of the if statement
                     L1:            ; entry point for else code
A:=0;                LOAD 0         ; load register with zero
                     STORE A        ; assign it to A
ENDIF                L2:            ; end of if statement
REPEAT               L3:            ; mark start of loop
A:= A+1;             LOAD A
                     ADD 1
                     STORE A
UNTIL A=100          LOAD A
                     XOR 100        ; clears register if A=100
                     JNZ L3:        ; jump to start of loop
END;
```

Example output from the C-- compiler

ELSE statement ENDIF

The 'IF' is read and procedure EXPR is called to parse the expression. An expression is either:

(1) term operator expression, or

(2) term.

Procedure TERM is called, reads the identifier 'Z' and returns it to EXPR. EXPR looks at the next token '=', and as this is an operator it matches with case (1), the operator is read and EXPR is called again. EXPR calls TERM which reads '3'. However, this time when EXPR looks at the next token it is 'THEN', which is not an operator, so this matches with case (2). EXPR returns 3 as the value of the expression. The whole expression has now been parsed as A=3, which is returned to IF_STATEMENT.

IF_STATEMENT reads 'THEN' and calls STATEMENT. The next token is the identifier 'B', therefore ASSIGN_STATEMENT is called. The identifier and the ':=' are read and then EXPR is called to parse the expression in a similar way to the first expression. Its value is returned and the next token (a semicolon) is read. STATEMENT now returns to IF_STATEMENT the next token is 'ENDIF', which matches with case (A), so the complete IF statement has been parsed.

Code generation

Code is generated by this compiler by substitution of a set piece of assembler for each operator.

The parser calls the procedure to generate code for a construct after it has generated code for all operands required for the operation to be calculated. For example, the code to assign an expression to a variable has to be generated after the expression has been calculated. For the statement X:=Y+Z; the parser calls the code generator as though the tokens arrived in the order Y, Z, +,X, :=.

The action of the generator at the start of an expression is to load the register. This loads the identifier or number with which the expression starts. Until the end of the expression, Generate is called with an identifier or number which is stored, then with an operator. On this, the second call code is generated.

The IF statement requires labels to be generated. A conditional GOTO is generated for the THEN token. The target label for this goto is generated when the ELSE token is encountered, or, if there is no ELSE part to the IF statement, when the ENDIF token is encountered. When an ELSE clause is present before generating the target GOTO, an unconditional GOTO must be generated whose target label is generated at the ENDIF token.

The REPEAT statement generates a label at the beginning and a conditional GOTO to that label, which is generated after the expression following the UNTIL token is calculated.

Data space is allocated for new variables when they are inserted into the symbol table.

The example output on page 155 illustrates how simply a compiler can be constructed. However, the generated code is inefficient and the compiler does not implement scope or types.

Commercial compilers generate more efficient code by examining the expressions in far greater detail before producing code. This process is called optimisation.

Optimisation techniques

There are a number of different optimisations a compiler may perform including:

Register allocation

One of the most important factors in generating fast code is using the full power of the microprocessor. If it has a set of fast registers, then these should be used for storing variables which are used most often. Each variable can be assigned a weight based upon how often it is used in the program, where it is in a program, and the number of instructions between uses. Loop control variables are often stored in registers as the loop is likely to be executed many times.

Constant folding

If a program contains a code fragment such as:

```
CONST PI = 3.2;

read(radius);
Diameter:=2*PI*radius;
```

part of the calculation of [illegible]eter can be done at compile time because P1 and 2 are both constants known to the compiler. The compiler can calculate 2*P1 and therefore generates code as though the source were:

```
Diameter := 6.4*radius;
```

This optimisation can only be used when there is more than one constant in an expression.

Peephole optimisation

A peephole optimiser reads through the code a compiler has generated and removes or replaces assembler instructions. If a compiler generated a pair of instructions like:

```
MOV AX,VAR ;store AX in VAR
MOV VAR,AX ;load AX from VAR
```

The second MOV instruction can be removed as the first inst[illegible]tion will ensure that register AX has the same value as VAR. A peephole optimiser will also swap an instruction for a more efficient one.

For example:

```
MOV 0,AX ;load zero into AX
```

may be replaced by

```
XOR AX,AX ;exclusive-or AX with
itself
```

This has the same effect as the first operation, but as it is a register-to-register operation it is more efficient.

Other optimisation techniques are loop invariant removal and inlining.

Conclusion

A simple compiler is quite easy to design. The real skill of the compiler writer is in generating efficient code. Good register allocation and peephole optimisation (or just good code generation) are the most important factors in the efficiency of the code generated by a compiler. ▶

Language definition

C-- uses the characters 0-9, the letters A-Z and the special characters '*/+-=:;'. from which the following keywords are defined:

BEGIN, IF, THEN, ELSE, ENDIF, REPEAT, UNTIL

C-- has two control constructs, IF and REPEAT, and assignment. A C-- program consists of:

```
BEGIN
 statements;
END
```

There are three types of statement:

Assignment has the form: identifier :=expression;

IF has the form: IF expression THEN statement ENDIF or IF expression THEN statement ELSE statement ENDIF

REPEAT has the form: REPEAT statements UNTIL expression

Expressions are defined below:

```
expression: term operator expression or term
term: (expression)
      or identifier
      or number
operator: *or/or-or+
```

All operators are of equal precedence and expressions are evaluated from left to right. Variables are all of type integer and do not have to be explicitly declared.

Lexical analyser interface

```
DEFINITION MODULE Analyse;    (* Interface for the lexical analyser *)
TYPE
  (* Types of tokens recognised by the lexical analyser *)
  TOKEN =  (TIF, TTHEN, TELSE, TENDIF, TREPEAT, TUNTIL, ID, DONE, NUM, DIVI,
           MULT,TBEGIN, TEND, PLUS, MINUS, BECOMES, EQUALS, LBRACKET,
           RBRACKET, TLABEL, TGOTO, GOFALSE, SEMICOLON);
VAR
  tokenval : INTEGER;                (* Number associated with each token *)

PROCEDURE Lex():TOKEN;               (* Returns next token *)
PROCEDURE Error(s:ARRAY OF CHAR);    (* Outputs an error message *)
END Analyse.
```

```
IMPLEMENTATION MODULE Analyse;                    (* Lexical analyser *)
FROM FIO IMPORT File, RdChar, WrStr,WrLn,         (* Modula-2 File I/O *)
                WrInt,WrChar, Open;
IMPORT IO;                                        (* Modula-2 Screen I/O *)
FROM Symbol_Table IMPORT Insert, Lookup, Symtab;
CONST
   BUFFERSIZE = 29;                     (* Size of input buffer *)
TYPE
  setofchar = SET OF CHAR;              (* Declare a character base type *)
VAR
  LineNo: INTEGER;                      (* Current line number *)
  In : File;                            (* Input file *)
  TokenSize : INTEGER;                  (* Size of token found *)
  Ident  : ARRAY [0..BUFFERSIZE] OF CHAR;  (* Declare buffer *)
  UnGetCh : BOOLEAN;                    (* Set if the buffer is full *)
  ch : CHAR;                            (* One character buffer *)

PROCEDURE Error (str : ARRAY OF CHAR); (* Called when an error is detected *)
BEGIN
  IO.WrStr(str); IO.WrStr('Line: '); IO.WrCard(LineNo,0); IO.WrLn;
END Error;

PROCEDURE Lex() : TOKEN;
BEGIN
  LOOP
    IF NOT UnGetCh THEN ch := RdChar(In)ELSE UnGetCh:=FALSE; END;
    IF ch IN setofchar('0'..'9') THEN (* Token is a Number *)
          tokenval := ORD(ch) - ORD('0');
          ch := RdChar(In);
          WHILE ch IN setofchar('0'..'9') DO
            tokenval := ORD(ch)-ORD('0') +CARDINAL(tokenval*10);
            ch := RdChar(In);
          END;
          UnGetCh := TRUE;
          RETURN NUM
    ELSE
      IF ch IN setofchar('A'..'Z') THEN (* Token is Ident  or Keyword *)
         TokenSize := 0;
         REPEAT
           Ident [TokenSize] := ch;
           ch := RdChar(In);
           INC(TokenSize);                  (* Increment token size *)
         UNTIL NOT(ch IN setofchar('A'..'Z','0'..'9','_'))
                 OR (TokenSize > BUFFERSIZE);
         IF TokenSize > BUFFERSIZE THEN Error('Token too large'); END;
         Ident [TokenSize] := 0C;           (* end of string marker *)
         tokenval := Lookup(Ident );        (* Is it in the symbol table? *)
         IF tokenval = 0 THEN               (* If not add it to the table *)
           tokenval :=                      (* Tokenval now contains tokens*)
              Insert(Ident ,ID,TokenSize);  (* position in the symbol table*)
         END;
         UnGetCh := TRUE;                   (* push current ch onto input *)
         RETURN Symtab[tokenval].t          (* return token type *)
      ELSE  (* Special characters *)
         CASE ch OF
           '('   : RETURN LBRACKET|
           ')'   : RETURN RBRACKET|
           '*'   : RETURN MULT|
           '/'   : RETURN DIVI|
           '+'   : RETURN PLUS|
           '-'   : RETURN MINUS|
           ';'   : RETURN SEMICOLON|
           '='   : RETURN EQUALS|
           ':'   : IF RdChar(In) <> '=' THEN  (* scan for ':=' *)
                     Error('equals expected') ELSE RETURN BECOMES END|
           15C   : INC(LineNo)|          (* Newline *)
           12C,0C, ' ' : |               (* Ignore LF, NUL, SPACE chars*)
           32C   : RETURN DONE ;         (* EOF found *)
          ELSE    Error('Unknown character');
          END;
       END;
     END;
  END;
END Lex;
BEGIN
  UnGetCh := FALSE;                 (* Initially no character in the buffer *)
  LineNo := 1;
  tokenval := Insert('IF',TIF,3);        (* Insert keywords in the symbol table *)
  tokenval := Insert('THEN',TTHEN,5);
  tokenval := Insert('ELSE',TELSE,5);
  tokenval := Insert('ENDIF',TENDIF,6);
  tokenval := Insert('REPEAT',TREPEAT,7);
  tokenval := Insert('UNTIL',TUNTIL,6);
  tokenval := Insert('BEGIN',TBEGIN,6);
  tokenval := Insert('END',TEND,4);
  In := Open("fred");               (* Open input file *)
END Analyse.
```

Code generator interface

```
DEFINITION MODULE Code;                (* interface to the code generator *)
FROM Analyse IMPORT TOKEN;
CONST
  NONE = 0 ;                           (* Value associated with operators *)
VAR
  NextLabel : INTEGER;                 (* Unique number for next label *)
  Loaded : BOOLEAN;                    (* Set when the register is loaded *)

PROCEDURE Declare(S: ARRAY OF CHAR);
PROCEDURE Generate(Tok :TOKEN;Value:INTEGER);
END Code.
```

```
IMPLEMENTATION MODULE Code;         (* Module contains the code generator *)
FROM FIO IMPORT File, RdChar, WrStr, WrLn,        (* Modula-2 File I/O *)
                WrInt,WrChar, Create;
FROM Symbol_Table IMPORT Symtab;                  (* To access identifiers *)
VAR
   Out : File ;                                   (* Output file *)
   Last: RECORD                                   (* Last Token *)
           Tok : TOKEN;
           Value: INTEGER;
         END;
   FreeSpace : CARDINAL;                          (* Marks Free data area *)
   POPPED : BOOLEAN;                              (* Last op was a POP *)

PROCEDURE Declare(s: ARRAY OF CHAR);       (* Allocate space for variables *)
BEGIN
  WrStr(Out,s); WrStr(Out," EQ "); WrInt(Out, FreeSpace,0); WrLn(Out);
  FreeSpace := FreeSpace+2,          (* Assuming byte addressing *)
END Declare;

PROCEDURE PrintOp(Operator: ARRAY OF CHAR);       (* Print Operator *)
BEGIN
   WrStr(Out,Operator);
   IF POPPED THEN
      WrStr(Out,' _TMP'); (* last op was a pop so load from TMP *)
      POPPED := FALSE;
   ELSE IF Last.Tok = NUM THEN WrInt(Out,Last.Value,0)  (* A Literal *)
        ELSE  WrStr(Out,Symtab[Last.Value].Entry^);     (* A memory location *)
        END;
   END; WrLn(Out);
END PrintOp;

PROCEDURE Printlabel(X :INTEGER);
BEGIN
  WrStr(Out,'L'); WrInt(Out,X,0); WrStr(Out,':'); WrLn(Out)
END Printlabel;

PROCEDURE Generate(Tok :TOKEN;Value:INTEGER);
(* Generates code for each construct in a simple pseudo-assembler all
operations are of the form: <LABEL:> OPERATOR OPERAND *)
BEGIN
  CASE   Tok OF
    PLUS    : PrintOp(' ADD ')|
    MINUS   : PrintOp(' SUB ')|
    MULT    : PrintOp(' MUL ')|
    DIVI    : PrintOp(' DIV ')|
    BECOMES : PrintOp(' STORE ')|
    EQUALS  : PrintOp(' XOR ')|
    LBRACKET: WrStr(Out,' PUSH ');WrLn(Out)|
    RBRACKET: WrStr(Out,' POP  _TMP');WrLn(Out); POPPED := TRUE|
    ID      : IF NOT Loaded THEN (* start of expression *)
                WrStr(Out,' LOAD '); WrStr(Out,Symtab[Value].Entry^); WrLn(Out)
              END|
    NUM     : IF NOT Loaded THEN (* start of expression *)
                WrStr(Out,' LOAD '); WrInt(Out,Value,0); WrLn(Out)
              END  |
    TLABEL  : Printlabel(Value)|
    GOFALSE : WrStr(Out,'JNZ '); Printlabel(Value)|
    TGOTO   : WrStr(Out,'JMP '); Printlabel(Value)
    ELSE ; (* if you dont know what it is don't translate it! *)

   END;
   Last.Tok := Tok;                              (* Store token and value *)
   Last.Value:= Value;
   Loaded := TRUE;                               (* register is loaded now *)
 END Generate;
 BEGIN
   Out := Create('Out');                         (* Open new output file *)
   FreeSpace := 1000;                            (* Start of Data area *)
   Declare('_TMP');                              (* Set aside temporary storage *)
   NextLabel := 1;                               (* Set first label's value *)
   Loaded   := FALSE;                            (* Accumulator is not loaded *)
 END Code.
```

Symbol table interface

```
DEFINITION MODULE Symbol_Table;      (* interface to the symbol table *)
FROM Analyse IMPORT TOKEN;

CONST
  MAXTAB = 1000;   (* Symbol table size *)
TYPE
  table = RECORD            (* Record used in the Symbol table *)
            t : TOKEN;
            Entry : POINTER TO ARRAY [0..30] OF CHAR;
          END;
VAR
  Symtab : ARRAY [1..MAXTAB] OF table; (* Symbol table *)

(* Looks for identifier in the symbol table returns its position or zero *)
PROCEDURE Lookup(S: ARRAY OF CHAR): INTEGER;

(* Adds a string to the symbol table *)
PROCEDURE Insert(S: ARRAY OF CHAR; TokenType: TOKEN; Size: INTEGER):INTEGER;
END Symbol Table.
```

```
IMPLEMENTATION MODULE Symbol_Table;
FROM Storage IMPORT ALLOCATE;  (* Procedure to allocate space on the heap *)
FROM Str IMPORT Compare, Copy; (* String manipulation routines *)
FROM Analyse IMPORT Error;
FROM Code IMPORT Declare;
VAR
  LastEntry :INTEGER;                  (* Last entry in the symbol table *)

PROCEDURE Lookup(str: ARRAY OF CHAR): INTEGER;
(* Searches symbol table for identifier *)
VAR
  index : INTEGER;
BEGIN
  FOR index := 1 TO LastEntry  DO
    IF Compare(str,Symtab[index].Entry^)=0 THEN RETURN index; END;
  END;
  RETURN 0;
END Lookup;

PROCEDURE Insert(str : ARRAY OF CHAR; tok : TOKEN; Size : INTEGER):INTEGER;
(* Adds Identifier and token type to symbol table *)
BEGIN
  IF MAXTAB = LastEntry THEN
    Error('Symbol table full');
  ELSE
    LastEntry := LastEntry +1;
    Symtab[LastEntry].t := tok;
    ALLOCATE(Symtab[LastEntry].Entry,Size);
    Copy(Symtab[LastEntry].Entry^,str);
    IF tok = ID THEN Declare(str); END;    (* Allocate space for identifier *)
    RETURN LastEntry;
  END;
END Insert;
BEGIN
  LastEntry := 0;                      (* No entries in the Symbol table *)
END Symbol_Table.
```

Main program

```
MODULE COMP;
(* Main module of the compiler - last modified 30-6-88*)
FROM Analyse IMPORT Lex, Error, TOKEN, tokenval;
FROM Code IMPORT Generate, NextLabel, Loaded, NONE;
VAR
  nexttoken : TOKEN;

(* These procedures are used before they are declared *)
PROCEDURE expr; FORWARD;
PROCEDURE statement; FORWARD;

PROCEDURE match(wanted: TOKEN);      (* move to next token *)
BEGIN
  IF wanted=nexttoken THEN           (* If the token is as expected*)
     nexttoken:=Lex()                (* Read in next token *)
  ELSE
    Error('Syntax Error');           (* otherwise output an error message *)
  END;
END match;

PROCEDURE term;                      (* checks terms are well formed *)
VAR
  Temp : INTEGER;         (* Stores unique number for temporary *)
(* Term is one of - a bracketted expression
                  - an identifier
                  - a number                   *)

BEGIN
  CASE nexttoken OF  (* Term is one of                             *)
    LBRACKET :                       (* - a bracketted expression      *)
        match(LBRACKET);
        Generate(LBRACKET,NONE);
        expr;
        match(RBRACKET);
        Generate(RBRACKET,NONE)|
    NUM:                             (* - a number                     *)
        Generate(NUM,tokenval);
        match(NUM)|
    ID :                             (* - an identifier                *)
        Generate(ID,tokenval);
        match(ID)|
    ELSE
      Error('Syntax error');
  END;
END term;

PROCEDURE expr;                      (* Parses expressions *)
VAR Lasttoken : TOKEN;
(* An expression is a term followed by zero or more repetions of
   an operator followed by a term *)
BEGIN
  Loaded := FALSE;                   (* register has to be loaded *)
  term;                              (* match a term *)
  LOOP                               (* followed by zero or more  *)
    CASE (nexttoken) OF              (* repetitions of :-         *)
      PLUS,MINUS,DIVI,MULT,EQUALS:   (* Operator                  *)
        Lasttoken := nexttoken;
        match(nexttoken);
        term;                        (* followed by term          *)

        Generate(Lasttoken,NONE)|    (* generate code for OP after term *)
      ELSE RETURN;                   (* end of exprssion          *)
    END;
  END;
END expr;

PROCEDURE Parse_if;
VAR
   Out,Endif : INTEGER;
BEGIN
  match(TIF);                         (* Find if *)
  expr;                               (* Parse the expression *)
  Out := NextLabel;                   (* get a unique label *)
  INC(NextLabel);
  Generate(GOFALSE,Out);              (* Goto it if the result is false *)
  match(TTHEN);
  statement;                          (* parse a statement *)
  IF nexttoken = TELSE THEN           (* If there is an else clause *)
     match(TELSE);
     Endif := NextLabel;              (* Get another label *)
     INC(NextLabel);
     Generate(TGOTO,Endif);           (* place a goto to the end of the if *)
     Generate(TLABEL,Out);            (* target label for false result *)
     statement;
     Generate(TLABEL,Endif);          (* generate label for end of if *)
  ELSE                                (* No else part *)
     Generate(TLABEL,Out);            (* generate label for false result *)
  END;
  match(TENDIF);
END Parse_if;

PROCEDURE Parse_repeat;
VAR Start : INTEGER;
BEGIN
  match(TREPEAT);
  Start := NextLabel;                 (* generate label for start of loop *)
  INC(NextLabel);
  Generate(TLABEL,Start);
  REPEAT
  statement;                          (* parse statements *)
  UNTIL (nexttoken = TUNTIL) OR (nexttoken=DONE);
  match(TUNTIL);
  expr;                               (* parse expression *)
  Generate(GOFALSE,Start);            (* generate goto start of loop *)
END Parse_repeat;

PROCEDURE Parse_assign;               (* parse assignment statements *)
VAR
  Tmp:INTEGER;
BEGIN
  Tmp := tokenval;                    (* store identifier          *)
  match(ID);
  match(BECOMES);
  expr;                               (* parse expression *)
  Generate(ID,Tmp);                   (* generate code for the assignment *)
  Generate(BECOMES,NONE);             (* after generating the expression  *)
END Parse_assign;

PROCEDURE statement;
BEGIN
 CASE (nexttoken) OF
   TIF :    Parse_if|
   TREPEAT: Parse_repeat|
   ID:      Parse_assign;
 END;
 match(SEMICOLON);  (* All statements are terminated with a semicolon *)
END statement;

PROCEDURE parse;
BEGIN
    nexttoken:= Lex();
    match(TBEGIN);
    WHILE((nexttoken # DONE) AND (nexttoken # TEND)) DO
      statement;
    END;
    match(TEND);
END parse;

BEGIN
  parse;
END COMP.
```

END