RHODES UNIVERSITY

June Examinations - 1997

Computer Science 301 - Paper 2 - Solutions

Section A

A1 Briefly explain the differences between each pair of concepts below:

(a) Self resident compilers and cross compilers
(b) Phrase structure and lexical structure
(c) Scope and extent of identifiers

Self-resident compilers generate code for the architecture of the machine on which they are running, while cross-compilers generate code for a different machine.

Phrase structure defines how tokens (words) are arranged into sentences; lexical structure defines how characters are arranged into tokens.

Scope defines the compile-time area of code in which an identifier can be recognised; extent defines the run-time period for which storage is assigned to, and associated with, an identifier.

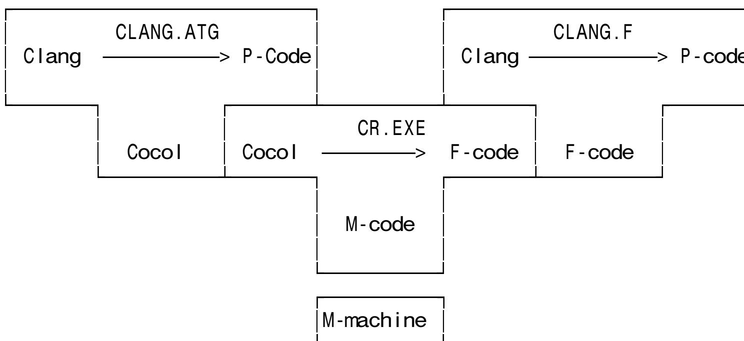
A2 Draw a diagram outlining the relationship between the various phases that occur in compilation, and briefly describe the responsibility of each phase.

A full description of this is to be found in the text book on pages 13-19.

A3 Draw T-diagrams that represent the processes that are used in the compiler generator/compiler/interpreter system that you have developed in the practicals of this course.

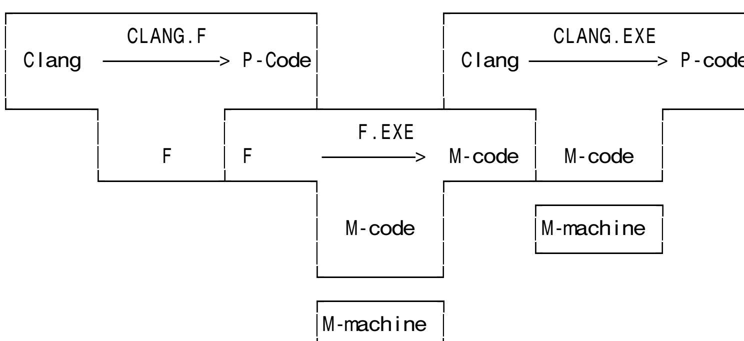
This was appallingly badly done (with a few exceptions). What I was looking for was something like this:

(a) Using Coco to generate the Clang compiler sources:

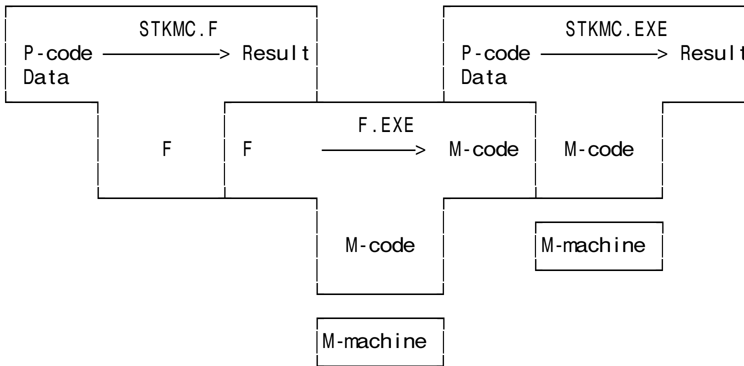


(Here F-Code = Modula-2 or C++ depending on your preference)

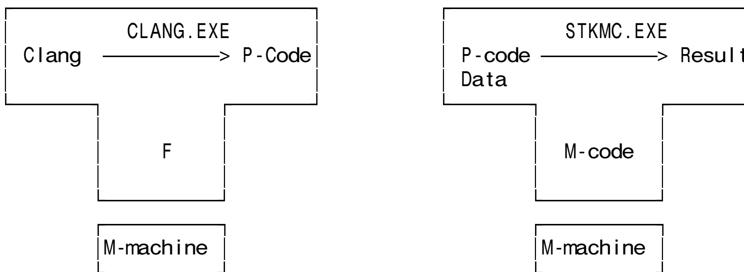
(b) Using BC or M2 to compile the resulting Clang compiler:



(c) Using BC or M2 (Modula or C++) to generate the interpreter:



(d) Chaining compiler and interpreter together:



and so on. What I saw in most cases were simply random T diagrams with almost anything written in each of the arms!

A4 A grammar G may be described by a 4-tuple.

(a) Name, and briefly describe, each component of this 4-tuple.

Text book, page 76 - 77.

(b) In terms of your notation, define precisely (mathematically) what is meant by a language L defined on the grammar G.

Text book, page 78. All we need is to write

$$L(G) = \{ w \mid w \varepsilon T^* ; S \Rightarrow^* w \}$$

(c) Define precisely (mathematically) what conditions must be satisfied by G to ensure that it is a "reduced grammar".

Text book, page 151. A **reduced grammar** is one that does not contain non-terminals that can never be reached from the start symbol, and non-terminals that cannot produce terminal strings.

All that was needed was to say: A grammar is said to be reduced if, for each non-terminal B we can write

$$S \Rightarrow^* \alpha B \beta$$

for some strings α and β, and where

$$B \Rightarrow^* \gamma$$

for some $\gamma \ \varepsilon \ T^*$.

A5 What do you understand by the concepts "ambiguous grammars" and "equivalent grammars"? Illustrate your answer by giving a simple example of an ambiguous grammar, and of an equivalent non-ambiguous grammar.

An ambiguous grammar is one in which there is at least one sentence that has two distinct parse trees. Two grammars are equivalent if they describe exactly the same language.

It is quite easy to write down ambiguous grammars, and there were some quite nice examples shown. But few students came up with equivalent unambiguous ones (ie ones that describe the same language). An easy example would be the one we discussed in class for Roman numerals:

```
Ambiguous:

     Units =    [ "I" ] [ "I" ] [ "I" ]
              | "IV" | "IX"
              | "V" [ "I" ] [ "I" ] [ "I" ]

Unambiguous:

     Units =    [ "I" [ "I" [ "I" ] ] ]
              | "IV" | "IX"
              | "V" [ "I" [ "I" [ "I" ] ] ]
```

A6 Explain what is meant by the FIRST and FOLLOW sets as applied to grammar analysis, and in terms of these state the rules that a grammar must obey for it to be LL(1).

Text book, pages 173 - 176

A7 You should be familiar with the use of the `CASE` *statement in Modula-2. (A few examples of syntactically correct* `CASE` *statements appear in appendix A so as to refresh your memory.)*

(a) Describe the form of a `CASE` *statement in EBNF. You do not need to give productions for Statement or for Expression, but you should give detailed productions for any other non-terminals you introduce.*

Several students came up with solutions that were overly complicated because they had not appreciated that a `CASE` statement is governed by an arbitrary `Expression`, and has "labels" consisting of `ConstExpressions` (which are also syntactically expressions). Thus the general idea of an `Expression` and of a `Statement` means that one can arrive at a rather neat solution as below. The major catch is in finding a way to allow those annoying vertical bars to appear in the correct places!

```
CaseStatement     = "CASE" Expression "OF" Case { "|" Case }
                    [ "ELSE" StatementSequence ] "END" .

Case              = [ CaseLabelList ":" StatementSequence ] .

CaseLabelList     = CaseLabels { "," CaseLabels } .

CaseLabels        = ConstExpression [ ".." ConstExpression ] .

StatementSequence = Statement { ";" Statement } .

ConstExpression   = Expression .
```

(b) Does your set of productions satisfy the LL(1) critera? Explain in detail.

This particular set does. It was, of course, possible to find a set that did not, and several students who did so managed to explain why their productions were not LL(1) compliant. But it was very apparent that very few students really had any clue on how to apply the LL(1) rules.

As the grammar is expressed above, we need to check that for the nullable structures the FIRST and FOLLOW sets have nothing in common (there are no productions with obvious "alternatives", of course).

If we examine

```
CaseStatement     = "CASE" Expression "OF" Case { "|" Case }
                    [ "ELSE" StatementSequence ] "END" .

Nullable portion { "|" Case }
FIRST  = { "|" }
FOLLOW = { "ELSE", "END" }
```

```
Nullable portion  [ "ELSE" StatementSequence ]
FIRST  = { "ELSE" }
FOLLOW = { "END" }
```

Similarly we examine

```
Case = [ CaseLabelList ":" StatementSequence ] .

Nullable portion [ CaseLabelList ":" StatementSequence ]
FIRST  = FIRST(Expression) = { "+", "-", "(", identifier, number }
FOLLOW = { "|", "ELSE", "END" }
```

And we examine

```
CaseLabelList = CaseLabels { "," CaseLabels } .

Nullable portion { "," CaseLabels }
FIRST  = { "," }
FOLLOW = { ":" }
```

And we examine

```
CaseLabels = ConstExpression [ ".." ConstExpression ] .

Nullable portion    [ ".." ConstExpression ]
FIRST  = { ".." }
FOLLOW = { ":", "," }

StatementSequence = Statement { ";" Statement } .

Nullable portion { ";" Statement } .
FIRST  = { ";" }
FOLLOW = { "|", "ELSE", "END" } U FIRST(Statement)
```

(c) *When a Modula-2 compiler translates a* `CASE` *statement it must check not only that the syntax is correct, but also that various static semantic constraints have been met. Identify at least two of these constraints.*

- The type of the selector `Expression` must match the type of the `ConstExpressions`
- The ranges must be properly defined (eg `45 .. 12 :` is invalid)
- The ranges must not overlap (Eg `12, 10 .. 15 :` is invalid
- The ranges must be unique (a "label" may not appear in more than one "arm")
- Strictly there should be a label for each possible value of the `Expression` if there is no `ELSE`. But this is not usually enforced. Instead a runtime check should be provided to catch expressions that at run-time cannot match any of the `ConstExpressions` defined at compile-time.

A8 *Appendix B shows an extract from a simple recursive descent compiler that handles compilation of a simple* `if - then` *statement. Suppose the statement syntax were to be extended to*

IfStatement = "IF" Condition "THEN" StatementSequence
["ELSIF" Condition "THEN" StatementSequence]
["ELSE" StatementSequence]
"END" .

How would the `IfStatement` *routine in the compiler have to be changed to accommodate this?*

Many students clearly had not appreciated the material in the last practical, and simply omitted to generate unconditional branch instructions where they were needed.

Here is a Modula-2 solution, using fairly good code generation.

```
PROCEDURE IfStatement;
(* IfStatement = "IF" Condition "THEN" StatementSequence
                 [ "ELSIF" Condition "THEN" StatementSequence" ]
                 [ "ELSE" StatementSequence" ] "END" . *)
  VAR
    Jump1, Jump2, TestLabel : CGEN.LABELS;
    Elsif : BOOLEAN;
  BEGIN
    Elsif := FALSE;
    SCAN.GetSYM; Condition(SYMSET{ThenSym, DoSym} + Followers);
    CGEN.JumpOnFalse(TestLabel, CGEN.undefined);
```

```
  IF SYM.Sym = ThenSym
    THEN SCAN.GetSYM
    ELSE Error(23); IF SYM.Sym = DoSym THEN SCAN.GetSYM END
  END;
  StatementSequence(SYMSET{ElseSym, ElsIfSym} + Followers);
  CGEN.BackPatch(TestLabel);
  IF (SYM.Sym = ElsIfSym) OR (SYM.Sym = ElseSym)
    THEN
      IF SYM.Sym = ElsIfSym THEN
        CGEN.Jump(Jump1, CGEN.undefined);
        CGEN.BackPatch(TestLabel);
        SCAN.GetSYM; Condition(SYMSET{ThenSym, DoSym} + Followers);
        CGEN.JumpOnFalse(TestLabel, CGEN.undefined);
        IF SYM.Sym = ThenSym
          THEN SCAN.GetSYM
          ELSE Error(23); IF SYM.Sym = DoSym THEN SCAN.GetSYM END
        END;
        StatementSequence(SYMSET{ElseSym} + Followers);
        Elsif := TRUE
      END;
      IF SYM.Sym = ElseSym
        THEN
          SCAN.GetSYM;
          CGEN.Jump(Jump2, CGEN.undefined);
          CGEN.BackPatch(TestLabel);
          StatementSequence(Followers);
          CGEN.BackPatch(Jump2)
        ELSE CGEN.BackPatch(TestLabel)
      END;
      IF Elsif THEN CGEN.BackPatch(Jump1) END
    ELSE CGEN.BackPatch(TestLabel)
  END;
  Accept(EndSym, 24)
END IfStatement;
```

And here is the equivalent C++ version:

```
void PARSER::IfStatement(symset followers)
// IfStatement = "IF" Condition "THEN" StatementSequence "END" .
{ CGEN_labels jump1, jump2, testlabel;
  bool elsif = false;
  GetSym();
  Condition(symset(SCAN_thensym, SCAN_dosym) + followers);
  CGen->jumponfalse(testlabel, CGen->undefined);
  if (SYM.sym == SCAN_thensym)
    GetSym();
  else
    { Report->error(23); if (SYM.sym == SCAN_dosym) GetSym(); }
  StatementSequence(symset(elsifsym, elsesym) + followers);
  CGen->backpatch(testlabel);
  if (SYM.sym == SCAN_elsifsym || SYM.sym = SCAN_elsesym) {
    if SYM.sym = elsifsym {
      CGen->jump(jump1, CGen->undefined);
      CGen->backpatch(testlabel);
      GetSym(); Condition(symset{SCAN_thensym, SCAN_dosym} + followers);
      CGen->jumponfalse(testlabel, CGen->undefined);
      if (SYM.sym == SCAN_thensym)
        GetSym();
      else
        { Report->error(23); if (SYM.sym == SCAN_dosym); GetSym(); }
      StatementSequence(symset(SCAN_elsifsym, SCAN_elsesym) + followers);
      elsif = true;
    }
    if (SYM.sym == SCAN_elsesym) {
      GetSym();
      CGen->jump(jump2, CGen->undefined);
      CGen->backpatch(testlabel);
      StatementSequence(followers);
      CGen->backpatch(jump2);
    }
    else CGen->backpatch(testlabel);
    if (elsif) CGen->backpatch(jump1);
  }
  else CGen->backpatch(testlabel);
  Accept(SCAN_endsym, 24);
}
```

A9 *Appendix C shows a simple recursive Clang program for reading and sorting a list of 6 numbers using a selection sort algorithm.*

(a) *Indicate how a symbol table might appear when compilation had proceeded as far as the point indicated by the comment.*

This was very badly done. I wondered whether anybody would have done the obvious thing - given that the system was available on the machines, simply compile it and run it. But nobody thought of doing that! Most people forgot that the names of the procedures would appear in the table. There are various ways to present the information. One that would have been acceptable would have been

Name	Class	Level	Offset	Size
SORTDEMO	Program			
SELECTIONSORT	Procedure	1	2	2
LIST	Variable	2	6	2
N	Variable	2	8	1
T	Variable	2	9	1
M	Variable	2	10	1
MAXPOS	Function	2	4	2
LIST	Variable	3	6	2
N	Variable	3	8	1
I	Variable	3	9	1
MAX	Variable	3	10	1

(b) *Discuss the run-time appearance of the stack if this program were to have been translated by the compiler studied in the course, at the stage where the* `SelectionSort` *routine had made the first recursive call to itself, and had just called the function* `MaxPos`. *You may assume either of the two possibilities - using a static link, or using a display. (Indicate clearly which you are describing.)*

This was also very badly done. Again, I wondered whether anybody would have done the obvious thing - given that the system was available on the machines, simply compile it and run it after adding a `STACKDUMP` statement. But nobody thought of doing that! Here is a detailed solution. Simpler solutions, in which stack frames were simply drawn as linked lists, were also acceptable.

```
510:    2    Data[0]                Stack frame for SortDemo
509:    3    Data[1]
508:    4    Data[2]
507:    5    Data[3]
506:    6    Data[4]
505:    7    Data[5]
504:    6    I

503:  ???    Return Value           Stack Frame for SelectionSort (first call)
502:  511    Display Copy
501:  511    Dynamic Link
500:  263    Return Address
499:  511    Mark Copy
498:  510    Parameter List = Address of Data[0]
497:    6    Size of List = Size of Data
496:    5    Parameter N
495:    7    Local T
494:    5    Local M

493:  ???    Return Value           Stack Frame for SelectionSort (recursive call)
492:  504    Display Copy
491:  504    Dynamic Link
490:  208    Return Address
489:  504    Mark Copy
488:  510    Parameter List = Address of Data[0]
487:    6    Size of List = Size of Data
486:    4    Parameter N
485:    1    Local T
484:    6    Local M
483:  484    Address of M

482:    0    Return Value           Stack frame for MaxPos
481:  511    Display Copy
480:  494    Dynamic Link
479:  124    Return Address
478:  494    Mark Copy
477:  510    Parameter List = Address of Data[0]
476:    6    Size of List = Size of Data
475:    4    Parameter N
474:  ???    Local I
473:  ???    Local Max

Display 511 494 483 511 511
```

(c) *Discuss briefly the advantages and disadvantages of using the static link as opposed to the display method for storage management.*

See text book, page 378.

Section B

This was really rather disappointing. Nobody at all seemed to appreciate that REAL and INTEGER types in a computer are *totally* different types, and have to be handled separately. I find this almost unbelievable, given that you are third year students and have programmed both in Modula-2 (where the distinction is enforced very rigidly) as well as in C++. There were several attempts made to track type information, but these were nearly all very naive, and often did nothing more than pass a parameter around that was not really used at all.

Here are the sorts of answers I had hoped for:

```
$CN
COMPILER CalcM
(* Four Function Calculator (mixed mode) with simple memory
   P.D. Terry, Rhodes University, 1997 *)

  FROM FileIO IMPORT
    StdOut, WriteLn, WriteString, WriteInt, WriteReal, Compare;
  FROM StringConversions IMPORT
    StrToInt, StrToReal;
  FROM Utils IMPORT
    IntToReal, RealToInt;

  TYPE
    TYPES = ( Real, Integer, Unknown );
    VALUE = RECORD
              CASE Type : TYPES OF
              | Real    : R : REAL;
              | Integer : I : INTEGER;
              | Unknown : (* nothing *)
              END
            END;
   CONST
     MaxMem = 100;
   TYPE
     INDEX = CARDINAL [0 .. MaxMem];
     NAME  = ARRAY [0 .. 25] OF CHAR;
   VAR
     Memory : ARRAY INDEX OF
                RECORD
                  Name  : NAME;
                  Value : VALUE
                END;
     Last : INDEX;

   PROCEDURE Retrieve (N : NAME; VAR V : VALUE);
   (* Use N to retrieve a value V from memory *)
     VAR
       I : INDEX;
     BEGIN
       Memory[0].Name := N (* sentinel *);
       I := Last (* linear search; must succeed *);
       WHILE Compare(N, Memory[I].Name) # 0 DO DEC(I) END;
       IF I # 0 (* we found it *)
         THEN V := Memory[I].Value (* we know its value *)
         ELSE V.Type := Unknown (* it was not there - error *)
       END
     END Retrieve;

   PROCEDURE Store (N : NAME; V : VALUE);
   (* Use N to store a value V in memory *)
     VAR
       I : INDEX;
     BEGIN
       Memory[Last+1].Name := N (* store at end, in case it is a new one *);
       I := 1 (* linear search, must succeed *);
       WHILE Compare(N, Memory[I].Name) # 0 DO INC(I) END;
       Memory[I].Value := V (* we must be able to store the value *);
       IF I > Last THEN (* it is effectively a new variable *)
         INC(Last);
         IF Last = MaxMem THEN (* crude, but effective *)
           WriteString(StdOut, "Memory overflow");
```

```
          HALT
        END
      END
    END Store;

(* ---------------------------------------------------------------- *)

CHARACTERS
  digit  =  "0123456789" .
  letter =  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

IGNORE CHR(9) .. CHR(13)
IGNORE CASE

TOKENS
  integer  = digit { digit } .
  real     = digit { digit } "." { digit } .
  name     = letter { letter | digit } .

PRODUCTIONS
  CalcM
    =                          (. Last := 0 (* initialize memory table *) .)
    { Assignment | Print } "QUIT" .

  Assignment                   (. VAR
                                     Value : VALUE;
                                     Name : NAME; .)
    = name                     (. LexString(Name) .)
      ":=" Expression<Value>   (. Store(Name, Value) .) .

  Print
     = "PRINT" OneExp
       { WEAK "," OneExp }
                               (. WriteLn(StdOut) .) .

  OneExp                       (. VAR Value : VALUE; .)
     = Expression<Value>       (. CASE Value.Type OF
                                    | Integer : WriteInt(StdOut, Value.I, 0);
                                    | Real    : WriteReal(StdOut, Value.R, 0, 4);
                                    | Unknown : WriteString(StdOut, " undefined")
                                  END .) .

  Expression<VAR E : VALUE>    (. VAR T : VALUE; .)
    = (   Term<E>
        | "+" Term<E>
        | "-" Term<E>          (. CASE E.Type OF
                                  | Integer : E.I := - E.I
                                  | Real    : E.R := - E.R
                                  | Unknown : (* nothing *)
                                  END .)
      )
      {   "+" Term<T>          (. CASE E.Type OF
                                  | Integer :
                                      CASE T.Type OF
                                      | Integer : E.I := E.I + T.I
                                      | Real    : E.Type := Real;
                                                  E.R := IntToReal(E.I) + T.R
                                      | Unknown : E.Type := Unknown;
                                      END
                                  | Real :
                                      CASE T.Type OF
                                      | Integer : E.R := E.R + IntToReal(T.I)
                                      | Real    : E.R := E.R + T.R
                                      | Unknown : E.Type := Unknown;
                                      END
                                  | Unknown : (* nothing *)
                                  END .)
        | "-" Term<T>          (. CASE E.Type OF
                                  | Integer :
                                      CASE T.Type OF
                                      | Integer : E.I := E.I - T.I
                                      | Real    : E.Type := Real;
                                                  E.R := IntToReal(E.I) - T.R
                                      | Unknown : E.Type := Unknown;
                                      END
                                  | Real :
                                      CASE T.Type OF
                                      | Integer : E.R := E.R - IntToReal(T.I)
                                      | Real    : E.R := E.R - T.R
                                      | Unknown : E.Type := Unknown;
                                      END
```

```
                                  | Unknown : (* nothing *)
                                  END .)
       } .

  Term<VAR T : VALUE>          (. VAR F : VALUE; .)
    = Factor<T>
       {   "*" Factor<F>        (. CASE T.Type OF
                                  | Integer :
                                     CASE F.Type OF
                                     | Integer : T.I := T.I * F.I
                                     | Real    : T.Type := Real;
                                                 T.R := IntToReal(T.I) * F.R
                                     | Unknown : T.Type := Unknown;
                                     END
                                  | Real :
                                     CASE F.Type OF
                                     | Integer : T.R := T.R * IntToReal(F.I)
                                     | Real    : T.R := T.R * F.R
                                     | Unknown : T.Type := Unknown;
                                     END
                                  | Unknown : (* nothing *)
                                  END .)
         | "/" Factor<F>        (. CASE T.Type OF
                                  | Integer :
                                     CASE F.Type OF
                                     | Integer : T.Type := Real;
                                                 T.R := IntToReal(T.I) / IntToReal(F.I)
                                     | Real    : T.Type := Real;
                                                 T.R := IntToReal(T.I) / F.R
                                     | Unknown : T.Type := Unknown;
                                     END
                                  | Real :
                                     CASE F.Type OF
                                     | Integer : T.R := T.R / IntToReal(F.I)
                                     | Real    : T.R := T.R / F.R
                                     | Unknown : T.Type := Unknown;
                                     END
                                  | Unknown : (* nothing *)
                                  END .)
         | "DIV" Factor<F>      (. CASE T.Type OF
                                  | Integer :
                                     CASE F.Type OF
                                     | Integer : T.I := T.I DIV F.I
                                     | Real    : T.Type := Unknown; SemError(100)
                                     | Unknown : (* nothing *)
                                     END
                                  | Real :
                                     T.Type := Unknown; SemError(100)
                                  | Unknown : (* nothing *)
                                  END .)
       } .

  Factor<VAR F : VALUE>        (. VAR Str : NAME; .)
    = name                     (. LexName(Str);
                                  Retrieve(Str, F);
                                  IF F.Type = Unknown THEN SemError(101) END .)
       | Number<F>
       | "(" Expression<F> ")"
       | "INTEGER" "("
          Expression<F> ")"     (. CASE F.Type OF
                                  | Real :
                                     F.Type := Integer; F.I := RealToInt(F.R)
                                  | Integer : SemError(102)
                                  | Unknown : (* nothing *)
                                  END .)
       | "REAL" "("
          Expression<F> ")"     (. CASE F.Type OF
                                  | Real : SemError(103)
                                  | Integer :
                                     F.Type := Real; F.R := IntToReal(F.I)
                                  | Unknown : (* nothing *)
                                  END .) .

  Number<VAR C : VALUE>        (. VAR
                                    Okay : BOOLEAN;
                                    Str : ARRAY [0 .. 20] OF CHAR; .)
    = integer                  (. LexString(Str); C.Type := Integer;
                                  StrToInt(Str, C.I, Okay) .)
       | real                  (. LexString(Str); C.Type := Real;
                                  StrToReal(Str, C.R, Okay) .) .
END CalcM.
```

```
$CX /* generate compiler, C++ classes */
COMPILER CalcC
/* Four Function Calculator (mixed mode) with simple memory
   P.D. Terry, Rhodes University, 1997 */

#include "misc.h"

/* ------------------------------------------------------------------ */

CHARACTERS
  digit  =  "0123456789" .
  letter =  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

IGNORE CHR(9) .. CHR(13)
IGNORE CASE

TOKENS
  integer  = digit { digit } .
  real     = digit { digit } "." { digit } .
  name     = letter { letter | digit } .

PRODUCTIONS
  CalcC
    =                         (. Last = 0; // initialize memory table  .)
    { Assignment | Print } "QUIT" .

  Assignment
    =                         (. VALUE Value;
                                 char Name[25]; .)
      name                    (. LexString(Name, sizeof(Name) - 1); .)
      ":=" Expression<Value>  (. Store(Name, Value); .) .

  Print
     = "PRINT" OneExp
       { WEAK "," OneExp }
                              (. printf("\n"); .) .

  OneExp
     =                        (. VALUE Value; .)
       Expression<Value>      (. switch (Value.Type) {
                                   case Integer : printf(" %d", Value.I); break;
                                   case Real    : printf(" %f", Value.R); break;
                                   case Unknown : printf(" undefined");  break;
                                 } .) .

  Expression<VALUE &E>
    =                         (. VALUE T; .)
      (    Term<E>
        | "+" Term<E>
        | "-" Term<E>         (. switch (E.Type) {
                                   case Integer : E.I = - E.I; break;
                                   case Real    : E.R = - E.R; break;
                                   case Unknown : break; // nothing
                                 } .)
      )
      {    "+" Term<T>        (. switch (E.Type) {
                                   case Integer :
                                     switch (T.Type) {
                                       case Integer : E.I = E.I + T.I; break;
                                       case Real    : E.Type = Real; E.R = E.I + T.R; break;
                                       case Unknown : E.Type = Unknown; break;
                                     } break;
                                   case Real :
                                     switch (T.Type) {
                                       case Integer : E.R = E.R + T.I; break;
                                       case Real    : E.R = E.R + T.R; break;
                                       case Unknown : E.Type = Unknown; break;
                                     } break;
                                   case Unknown : break; // nothing
                                 } .)
        | "-" Term<T>         (. switch (E.Type) {
                                   case Integer :
                                     switch (T.Type) {
                                       case Integer : E.I = E.I - T.I; break;
                                       case Real    : E.Type = Real; E.R = E.I - T.R; break;
                                       case Unknown : E.Type = Unknown; break;
                                     } break;
                                   case Real :
                                     switch (T.Type) {
                                       case Integer : E.R = E.R - T.I; break;
                                       case Real    : E.R = E.R - T.R; break;
```

```
                                      case Unknown : E.Type = Unknown; break;
                                    } break;
                                  case Unknown : break; // nothing
                                } .)
      } .

  Term<VALUE &T>
    =                           (. VALUE F; .)
      Factor<T>
      {   "*" Factor<F>         (. switch (T.Type) {
                                  case Integer :
                                    switch (F.Type) {
                                      case Integer : T.I = T.I * F.I; break;
                                      case Real    : T.Type = Real; T.R = T.I * F.R; break;
                                      case Unknown : T.Type = Unknown; break;
                                    } break;
                                  case Real :
                                    switch (F.Type) {
                                      case Integer : T.R = T.R * F.I; break;
                                      case Real    : T.R = T.R * F.R; break;
                                      case Unknown : T.Type = Unknown; break;
                                    } break;
                                  case Unknown : break; // nothing
                                } .)
        | "/" Factor<F>         (. switch (T.Type) {
                                  case Integer :
                                    switch (F.Type) {
                                      case Integer :
                                             float t = T.I; float f = F.I;
                                             T.Type = Real; T.R = t / f; break;
                                      case Real    : T.Type = Real; T.R = T.I / F.R; break;
                                      case Unknown : T.Type = Unknown; break;
                                    } break;
                                  case Real :
                                    switch (F.Type) {
                                      case Integer : T.R = T.R / F.I; break;
                                      case Real    : T.R = T.R / F.R; break;
                                      case Unknown : T.Type = Unknown; break;
                                    } break;
                                  case Unknown : break; // nothing
                                } .)
        | "DIV" Factor<F>       (. switch (T.Type) {
                                  case Integer :
                                    switch (F.Type) {
                                      case Integer : T.I = T.I / F.I; break;
                                      case Real    : T.Type = Unknown; SemError(100); break;
                                      case Unknown : break; // nothing
                                    } break;
                                  case Real :
                                    T.Type = Unknown; SemError(100); break;
                                  case Unknown : ; break; /* nothing */
                                } .)
      } .

  Factor<VALUE &F>
    =                           (. char Str[25]; .)
      name                      (. LexName(Str, sizeof(Str) - 1);
                                   Retrieve(Str, F);
                                   if (F.Type == Unknown) SemError(101); .)
      | Number<F>
      | "(" Expression<F> ")"
      | "INTEGER" "("
         Expression<F> ")"      (. switch (F.Type) {
                                  case Real    : F.Type = Integer; F.I = F.R; break;
                                  case Integer : SemError(102); break;
                                  case Unknown : break; // nothing
                                } .)
      | "REAL" "("
         Expression<F> ")"      (. switch (F.Type) {
                                  case Real    : SemError(103); break;
                                  case Integer : F.Type = Real; F.R = F.I; break;
                                  case Unknown : break; // nothing
                                } .) .

  Number<VALUE &C>
    =                           (. char Str[100]; .)
      integer                   (. LexString(Str, sizeof(Str) - 1);
                                   C.Type = Integer; C.I = atoi(Str); .)
      | real                    (. LexString(Str, sizeof(Str) - 1);
                                   C.Type = Real; C.R = atof(Str); .) .
END CalcC.
```

```
// Various common items for calculator

#ifndef MISC_H
#define MISC_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <limits.h>

#define  boolean  int
#define  bool     int
#define  true     1
#define  false    0
#define  TRUE     1
#define  FALSE    0
#define  maxint   INT_MAX

#if __MSDOS__ || MSDOS
#  define  pathsep '\\'
#else
#  define  pathsep '/'
#endif

static void appendextension (char *oldstr, char *ext, char *newstr)
// Changes filename in oldstr from PRIMARY.xxx to PRIMARY.ext in newstr
{ int i;
  char old[256];
  strcpy(old, oldstr);
  i = strlen(old);
  while ((i > 0) && (old[i-1] != '.') && (old[i-1] != pathsep)) i--;
  if ((i > 0) && (old[i-1] == '.')) old[i-1] = 0;
  if (ext[0] == '.') sprintf(newstr,"%s%s", old, ext);
    else sprintf(newstr, "%s.%s", old, ext);
}

typedef enum { Real, Integer, Unknown } TYPES;

struct VALUE {
  TYPES Type;
  float R;
  int I;
};

struct ENTRY {
  char Name[25];
  VALUE Value;
};

static ENTRY Memory[100];
static int Last;

static void Retrieve(char *N, VALUE &V) {
  strcpy(Memory[0].Name,N);
  int i = Last;
  while (strcmp(N, Memory[i].Name)) i--;
  if (i)
    V = Memory[i].Value;
  else
    V.Type = Unknown;
}

static void Store(char *N, VALUE V) {
  strcpy(Memory[Last+1].Name, N);
  int i = 1;
  while (strcmp(N, Memory[i].Name)) i++;
  Memory[i].Value = V;
  if ((i > Last)) {
    Last++;
    if (Last == 100) { printf("Memory Overflow"); exit(1); }
  }
}

#endif /* MISC_H */
```