Computing

University project work

##

Project Report: Pascal Unconventional Desk Calculator.

Name: Alexis Huxley.

Year of Study: 3.

Degree Course: Applied Mathematics.

Project Supervisor: Meurig Beynon.

##

Section 0: Introduction.

The Unconventional Desk Calculator, or udc as it shall henceforth be known, is a conventional desk calculator in that it allows the assignment and interrogation of mathematical expressions but it has additional features. These include function definitions, normal and array structured variables, logical operators such as 'and' and 'or', the if-then-else clause for use inside arithmetic expressions and a range of explanatory error messages. This particular version of the udc has been written in Berkeley Pascal. Because of this the udc's implementation has been complicated by the restricted set of software tools available. Software tools available in a comparable high level language like C, for instance, mean that lexical analysers and parsers can be written with greater ease.
This document has been designed to give an in depth explanation of the use and abilities of the udc (Section 1), to give an explanation of the workings and algorithms involved in the actual program at the same time as explaining what these processes are for and why they are necessary (Section 2), to discuss the novel features, restrictions, and problems incurred in this particular implementation of udc (Section 3), and finally to give Backus Normal form specifications of udc programs and pseudo-code (a translation from the high level language to a not so detailed but easier to read, nearer to English language) for all procedures within the program (Section 4).

Section 1: User Documentation.

1.0: Introduction.

Each subsequent section of the user documentation is intended to explain the results and syntax and give examples for clarification of the commands and concepts involved in that section and for no other section. Larger examples covering several points, in the form of small programs and output results are given at the end of this section.

1.1: Execution and Termination.

The udc is executed by typing 'udc'. The prompt 'UDC> ', visual notification that the program is in a state ready to accept the next command, then appears. Commands are then entered in the form described in the rest of this section. Each command is either a blank line, an interrogation of an expression or an assignment. If the commands for a udc execution are in a file then prompts, though not error messages or interrogation responses, may be suppressed. A udc execution is terminated by pressing Control-D at the beginning of a new line.

1.2: Expression Types.

Expressions can be of the following basic types:
a) Non-negative integers in the range 0 to some maximum value determined by the system.
   e.g. 6, 45
b) Simple variables represented by the 26 letters of the alphabet in small type.
   e.g. a, b, z

c) array variables represented by the 26 letters of the alphabet in capitals followed by a defined array index in brackets in the range 0 to an arbitrary upper index limit which is determined from within the udc program. This upper index limit is here set to 99.
   e.g. A(7), B(98)
d) The undefined value. That is, the value used to represent to the value of variables to which no expression has been assigned or to which an expression with no defined value is assigned. The symbol for this is '@'. All variables are set to this value on udc startup.
e) Complex expressions, made up by combining these simple expressions using the available operator set.

1.3: Command Types.

All udc commands are either blanks, assignments or interrogations, though several commands may be placed on one line or one command may be placed over several lines [1.5].
a) A blank command consists of the carriage return char alone.
   e.g. Udc Sequence 1.

```
UDC> <RETURN>
UDC> <RETURN>
UDC>
```

b) Assignment is to either a simple or array variable. The expression being assigned to the variable may be of any form, and does not have to be defined numerically at the time of assignment. This is the major feature of the udc. That is it allows the assignment of formulae, so that although the values of variables within an assigned expression may change, this does not make the assignment obsolete since the assignment was to the formula representing the expression, and not to the numerical value that that expression held at the time of assignment. Other than error checks, and possible application of the evaluation operator, nothing is done to an assigned formula other than recording it in a table of variable names. A legal assignment results in the expression assigned being recorded with the variable assigned to by way of a variable table which records all successful assignments. An Assignment has the following syntax

```
<variable> = <expression1>
```

   e.g. `a = 8` sets the simple variable a to the value 8.
   `A[5] = @` sets the 5th variable of the array A to the undefined value.
   `b = c + d` sets b to the formula c+d - not the explicit value of c+d.

   There are however certain anomalies in what constitutes a valid assignment command [2.4] caused by the context of certain assignations.
c) A legal interrogation results in the printing of an expression on standard output. The form of the expression printed depends on the type of the expression interrogated. If the interrogated expression is a simple or vector variable then the expression printed is the formula assigned to that variable in the variable table otherwise the expression printed is the expression interrogated. A legal interrogation has the syntax

```
? <expression1>
```

   e.g. `? a` results in the printing of the formula associated with a in the variable table.
   `? a + b` results, rather unusefully, in the printing of the same expression 'a+b'.
   `? 8 - 3` results in the printing of '8-3'.

   So that an interrogation is of some use and expressions can be evaluated, there is the evaluation operator [1.4.6].

1.4: Operators.

1.4.0: Introduction.

The operator set provides a means of combining the simple expressions to make complex and more useful expressions. All operators act on numbers or undefined

values only, since at the time of evaluation the subexpressions within an expression that constitute operands are themselves evaluated first to return either a number or the undefined value. With the exception of the if-then-else operator, if the expression depends on any operand which, when evaluated is the undefined value then the whole expression is undefined.
All explanations of evaluation of operands with a given operator given below are those that would occur when the evaluation is effected, since before then expressions are represented as formulae within the variable table.

1.4.1: Unary Operators.

There are two unary operators.
a) Unary minus has the following syntax
    - <expression1>
   The minus sign denotes a unary minus, as opposed to binary minus (the subtraction operator), if the previous symbol is not a number, a simple variable, the undefined value, a closing brace or a right parenthesis from an expression or array variable. The effect of the unary minus on an expression is to negate any defined evaluation of it.
   e.g. - a will negate the value of a if it is explicit at the time of evaluation.
        - 6 this is not a simple expression but a complex one defined as 'the negation of 6', that is subtly different from 'minus 6'.
b) The logical 'not' has the following syntax
    ! <expression1>
   Its effect on a numeric value is to return 0 if the expression has value not equal to 0 and to return 1 if the value is 0.
   e.g. ! 2 will return 0 on evaluation.
        ! 0 will return 1 on evaluation.
        ! @ will return the undefined value on evaluation.

1.4.2: Binary Operators.

There are fifteen binary operators.
a) The additive operator has the following syntax
    <expression1> + <expression2>
   and the resulting expression is the number defined by sum of the evaluated operands.
   e.g. 1 + 2 will return 3 on evaluation.
        1 + @ will return the undefined value on evaluation.
b) The subtractive operator has the following syntax
    <expression1> - <expression2>
   The minus sign is assumed to represent a binary minus; the subtractive operator if it does not meet the conditions for being a unary minus; that is it is the subtractive operator if the previous symbol is a number, simple variable, the undefined value, or a closing parenthesis or brace, and the resulting expression is the number defined by the difference of the evaluated operands if expression1 is the greater, otherwise it is the number defined by negation of this difference.
   e.g. 1 - 2 will return -1 on evaluation.
c) The multiplicative operator has the following syntax
    <expression1> * <expression2>
   and the resulting expression is the number defined by the product of the evaluated operands.
   e.g. 2 * 3 will return 6 on evaluation.
d) The division operator has the following syntax
    <expression1> / <expression2>
   and results in the number defined by the quotient of the evaluated expression1 divided by the evaluated expression2 unless the evaluated expression2 is zero.
   e.g. 1 / 2 will return 0 on evaluation.
        2 / 1 will return 2 on evaluation.
        6 / @ will return the undefined value on evaluation.
e) The modulus operator has the following syntax
    <expression1> % <expression2>

and returns in the number defined by the remainder of the division of the evaluated expression1 by the evaluated expression2.
e.g. 14 % 6 will return 2 on evaluation.

f) The exponent operator has the following syntax
<expression1> ^ <expression2>
and returns the number defined according to the following function of the evaluated operands

$$e^{\langle expression2\rangle * \ln \langle expression1\rangle}$$

e.g. 2 ^ 4 will return 16 on evaluation.

g) The maximum operator has the following syntax
<expression1> /\ <expression2>
and returns the number defined by the greater of the two evaluated operands.
e.g. 1 /\ 2 will return 2 on evaluation.
-1 /\ -2 will return -1 on evaluation.

h) The minimum operator has the following syntax
<expression1> \/ <expression2>
and returns the number defined by the lesser of the two evaluated operands.
e.g. 1 \/ 2 will return 1 on evaluation.
-1 \/ -2 will return -2 on evaluation.

i) The less-than relational operator has the following syntax
<expression1> < <expression2>
and returns the number 1 if the evaluated expression1 is less than the evaluated expression2, otherwise it returns 0.
e.g. 2 < 2 will return 0 on evaluation.
1 < 2 will return 1 on evaluation.

j) The less-than-or-equal-to relational operator has the following syntax
<expression1> <= <expression2>
and returns the number 1 if the evaluated expression1 is less than or equal to the evaluated expression2, otherwise it returns 0.
e.g. 2 <= 2 will return 1 on evaluation.
2 <= 1 will return 0 on evaluation.

k) The equal-to relational operator has the following syntax
<expression1> == <expression2>
and returns the number 1 if the evaluated expression1 is less than or equal to the evaluated expression2, otherwise it returns 0.
e.g. 2 == 2 will return 1 on evaluation.
2 == 1 will return 0 on evaluation.

l) The greater-than-or-equal-to relational operator has the following syntax
<expression1> >= <expression2>
and returns the number 1 if the evaluated expression1 is less than or equal to the evaluated expression2, otherwise it returns 0.
e.g. 2 >= 3 will return 0 on evaluation.
2 >= 1 will return 1 on evaluation.

m) The greater-than relational operator has the following syntax
<expression1> > <expression2>
and returns the number 1 if the evaluated expression1 is less than or equal to the evaluated expression2, otherwise it returns 0.
e.g. 2 > 2 will return 0 on evaluation.
2 > 1 will return 1 on evaluation.

n) The logical 'and' operator has the following syntax
<expression1> & <expression2>
and returns the number 1 if both evaluated operand expressions are non-zero otherwise it returns 0.
e.g. 2 & 3 will return 1 on evaluation.
2 & 0 will return 0 on evaluation.

o) The logical 'or' operator has the following syntax
<expression1> | <expression2>
and returns the number 1 if at least one of the evaluated operands is non-zero other wise it returns 0.
e.g. 2 | 0 will return 1 on evaluation.
0 | 0 will return 0 on evaluation.

1.4.3: If-then-else.

If-then-else is a the only ternary operator used to build an expression with the following syntax

```
if <expression1> then <expression2> else <expression3>
```

The expression is defined by the evaluation of expression2 if the evaluated expression1 is non-zero regardless of whether expression2 is defined or not, and is defined as the evaluation of expression3 if the evaluation of expression1 is zero regardless of whether expression2 is defined or not. If expression1 is is not numerically defined then the value resulting from the evaluation of the whole if-then-else expression is also undefined regardless of whether expression2 or expression3 are defined.

```
e.g. if 75 then 43 else 44      will when evaluated return 43.
     if 0 then 2 else @         will return @ - the undefined value.
     if 1 then 2 else @         will return 2.
     if @ then 100 else 200     will return @ since the conditional expression
                                is undefined.
```

1.4.4: Operator Precedence.

The unary and binary operators have precedences, which are used to decide the order of evaluation in a unparenthesised expression. For instance the expression

```
1 + 2 * 3
```

could have two meanings. Which of the two possible expressions is being represented is decided by the relative positions of the operators concerned in the operator precedence table. The precedences are given below.

Table 1: Operator Precedences.

```
HIGHEST: unary minus, logical 'not'
         exponentiation
         multiplication, modular division, quotient division
         subtraction, addition
         maximum, minimum
         relational operators
         logical 'and'
         logical 'or'
LOWEST:  if-then-else
```

This gives us an order for evaluation of multi-operator expressions.

```
e.g. '12 + 3 / - 1 /\ 2 & ! 6 | 2 * 4 \/ 0 <= 9'
     1) - 1 ------> -1
        ! 6 ------> 0
        gives '12 + 3 / -1 /\ 2 & 0 | 2 * 4 \/ 0 <= 9'
     2) 3 / -1 ---> -3
        2 * 4 ----> 8
        gives '12 + -3 /\ 2 & 0 | 8 \/ 0 <= 9'
     3) 12 + -3 --> 9
        gives '9 /\ 2 & 0 | 8 \/ 0 <= 9'
     4) 9 /\ 2 ---> 9
        8 \/ 0 ---> 0
        gives '9 & 0 | 0 <= 9'
     5) 0 <= 9 ---> 1
        gives '9 & 0 | 1'
     6) 9 & 0 ----> 0
        gives '0 | 1'
     7) 0 | 1 ----> 1
        gives '1'
```

1.4.5: Parentheses.

Operator precedence supplies us with a default set of expression/operator associations within any expression. However we may wish to override this. For the example taken in [1.4.4] we may wish to evaluate the expression

```
1 + 2 * 3
```

by which we mean to represent the expression given in reverse polish notation as

```
1 2 + 3 *
```

rather than the other possibility which is the default. To do this we use parentheses to force evaluation in an order that does not traverse parentheses until there remains a simple expression within them.
e.g. 1 + 2 * 3 gives 7
(1 + 2) * 3 gives 9
1 + 2 * 3 ^ 4
gives the same as 1 + (2 * (3 ^ 4)) which gives evaluation in the reverse order to ((1 + 2) * 3) ^ 4.

1.4.6: Braces.

Braces are the evaluation operator, without which calculation can not take place. They have the following syntax

```
{ <expression> }
```

The result is an expression, either integer or the undefined value.
e.g. { 2 + 3 } gives 5
a + { 2 + 3 } gives a + 5

1.5: Spaces, Multiple Commands, Split Lines and Comments.

All spaces and tab characters within any entry are ignored.
e.g. 'a+b' is the same as 'a + b'
Several commands may be entered on one line by separating them with semicolons in the following way

```
<command> ; <command>
```

Prompting if enabled [1.5] will be suppressed until all commands in a command line have been completed.
e.g. Udc Sequence 2.

```
UDC> a = b + c ; ? a
b + c
UDC> b = 1 ; c = 2 ; ? { a }
3
```

Conversely, commands may be spread over several lines using a colon to terminate each unfinished line. Any input on a line beyond a colon is ignored. This allows comments within a udc input program to be inserted.
e.g. Udc Sequence 3.

```
UDC> ? { 6 + :
9 }
15
UDC> :a is the width
a = 1010
UDC>
```

1.5: Silent Mode.

Prompting may be suppressed by supplying the udc call with an argument.
e.g. "udc -s"
This gives improved readability of output sent to files.

1.6: Error Types and Reports.

There are ten basic error types and appropriate diagnostic reports that are provided as an aid to tracing errors. When any error is encountered then the command which contains the error is discarded. Error messages, causes and examples are described below.
a) Incomplete expression.
This is caused by an operator having insufficient operands to act upon in an expression made up of binary or unary operators.
e.g. ? { a + b - - }
a = b *

b) Zero division.
   This is caused by an attempt to divide by zero either directly in the quotient operator or in the modulo operator. Attempts to divide by zero will only be detected at the time of evaluation, since an operand may become non-zero before evaluation, due to reassignment of variables that the operand depends on.
   e.g. ? { 1 / (0 /\ -1) }
        a = { 3 % 0 }
c) Circular definition.
   This is caused by attempting to define a variable in terms of itself either directly or through the definitions of other variables.
   e.g. a = a + 1
        b = c ; c = b + 1
        a = b ; b = c ; c = d ; d = e ; e = a
d) Unknown symbol.
   This is caused by there being an illegal symbol in the input.
   e.g. ? { a + # }
        a = A [10]
e) Missing symbol.
   This is caused by an expected symbol not being encountered. There are several symbols that are expected after certain symbol patterns. Below are given these patterns and example errors.
   1) Right parenthesis follows an array indexing expression.
      e.g. A (1 = 0
   2) Right brace follows an expression which is preceded by a left brace.
      e.g. ? { 2 + 3)
   3) Keyword 'then' follows an expression preceded by keyword 'if'.
      e.g. if x == 1 else 3
   4) Keyword 'else' follows an expression preceded by keyword 'then'.
      e.g. if x == 1 then 3
   5) Assignment sign follows a simple variable that starts a command.
      a + b
   6) Assignment sign follows an array variable that starts a command.
      A(1) + 2
f) Invalid expression.
   This is caused by an expression having too many operands.
   e.g. a = b c + d
g) Unexpected symbol.
   This is caused by an encountered symbol being out of context.
   e.g. a = ?
h) Invalid statement.
   This caused by the first symbol in a command not being an interrogation mark, a simple variable or an array variable.
   e.g. if x > 0 then 4 else -4
i) Undefined array index.
   This is caused by an array variable's index being undefined when evaluated.
   e.g. a = @ ; ? A (a)
j) Invalid array reference.
   This is caused by an array variable's index being outside a range of legal indexes determined by a defined constant within the udc source program.
   e.g. ? A (1000)

1.7: Example Programs.

As an aid to understanding the concepts involved in the udc several example programs are given here.

1.7.1: Alternatives to the Logical 'And' Operator.

This program explores four alternatives to the conventional logical 'and' operator. Since the input commands are quite complex they were put in a file and supplied to the udc using the call

```
udc -silent < inputfile > outputfile
```

so that program and results, uninterrupted by prompts, could be kept separate.

Program 1 (with line numbering): Logical 'And' Test.

```
 1  : logical 'and' by alternative definitions
 2  : a and b are the operands
 3  : A(1), A(2), A(3), A(4) are the alternative definitions
 4  : A(0) is normal logical 'and' operator
 5
 6  : function definitions
 7  A(0) = a & b
 8  A(1) = ! ( !a | !b )
 9  A(2) = if a then ( if b then 1 else 0 ) else 0
10  A(3) = ( !( !a ) + !( !b ) ) / 2
11  A(4) = if a*b == 0 then 0 else 1
12
13  : print function definitions
14  ? A(0); ? A(1); ? A(2); ?A(3); ?A(4)
15
16  : tests over all relevant a and b
17  a = 0 ; b = 0
18  ? {A(0)}; ? {A(1)}; ? {A(2)}; ? {A(3)}; ? {A(4)}
19  a = 1
20  ? {A(0)}; ? {A(1)}; ? {A(2)}; ? {A(3)}; ? {A(4)}
21  a = 0 ; b = 1
22  ? {A(0)}; ? {A(1)}; ? {A(2)}; ? {A(3)}; ? {A(4)}
23  a = 1
24  ? {A(0)}; ? {A(1)}; ? {A(2)}; ? {A(3)}; ? {A(4)}
```

This program uses variables in an array to store five different functions that represent the logical 'and' operator. The program consists of the definitions of these five functions in terms of two variables a and b (lines 7-11), the printing of these functions (line 14), and then the evaluation of all five functions with a and b set to all relevant permutations of 1 and 0 (lines 17-24).
The functions are the following
A(0) = a & b
This makes use of the normal logical 'and' operator on the variables a and b, and is included as a control experiment for comparing its results with the results of the other functions.
A(1) = ! (!a | !b)
This function is merely the manipulation of the A(0) function using a bit of boolean algebra.
A(2) = if a then (if b then 1 else 0) else 0
This function is a rather wordy way of checking for when both a and b are set to non-zero values. If the first variable, a, is not 0 then the final result depends on the value of the second variable b.
A(3) = (!(!a) + !(!b)) / 2
This function makes use of two features of the udc. Firstly the logical 'not' of the logical 'not' of a variable is always 1 or 0. Consider the following table.

Table 2: Logical 'Not' of Logical 'Not'.

a	!a	!(!a)
-3	0	1
-2	0	1
-1	0	1
0	1	0
1	0	1
2	0	1
3	0	1

! (!a) is 0 if a is 0 and is 1 otherwise. Secondly, this function makes use of integer division. The sum (!(!a) + (!b)) can result in 3 different values; 0, 1 and 2. Consider the following table.

Table 3: Results from Integer Division on the Values 0 to 2.

x	x/2	x/2 integer
0	0.0	0
1	0.5	0
2	1.0	1

This shows that in integer arithmetic, the only combination of a and b that when 'not-notted', summed and subjected to integer division that gives a non-zero result is when both variables are non-zero - that is, when a AND b are non-zero.
A(4) = if a*b == 0 then 0 else 1
The final definition depends on the product of any two integers is non-zero only if the two integers are non-zero. That is if the product is zero then at least one of the two integers must be zero, so the logical 'and' must also return zero.

Udc Sequence 4.

```
a & b
!( !a | !b )
if a then ( if b then 1 else 0 ) else 0
( !( !a ) + !( !b ) ) / 2
if a * b == 0 then 0 else 1
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1
1
1
1
1
```

1.7.2: An Alternative Multiplication Function.

This function tests on specified values the quick method for multiplying two numbers with a difference of 2 in ones head. That being square the average and subtract 1. If the two numbers have not got a difference of 2 then the value returned is undefined.

Program 2 (with line numbering): Multiplication.

```
 1  : definitions of test values
 2  a = <number>
 3  b = <number>
 4
 5  : c is difference function
 6  c = (a/\b) - (a\/b)
 7
 8  : d is average function if c = 2
 9  d = (a\/b) + 1
10
11  : e is result function
```

```
12  e = if c == 2 then d ^ 2 - 1 else @
13
14  : interrogate result
15  ?{e}
```

This program is split into 5 sections. The first of these defines the values of the variables whose product is to be calculated by this alternative method (lines 1-3). The absolute value of the difference is calculated by subtracting the minimum of the two values from the maximum value (lines 5-6). This line could be replaced by a more easily understandable one

```
6  c = if a >= b then a else b
```

The third section is the defining of the averaging function (lines 8-9). Which if the difference is 2 will be the minimum of the values plus 1. Fourthly the result function is defined as

```
12  if the difference is 2 then e is the average of a and b squared minus
    1 otherwise e is undefined
```

Finally, this value is actually calculated (lines 14-15)
This program demonstrates the use of comments and blank lines to make a program more easily understood. It also shows the way that functions can be defined a long time before calculation.

1.7.3: Complete Udc Session (with inserted comments and line numbering).

Udc Sequence 5.

```
1  $ udc
```

The udc is called with no arguments so prompting is given.

```
2  UDC> ?a
3  @
```

Any previously undefined variable is initially set to the undefined value.

```
4  UDC> a = b + c
```

a is assigned the formula b+c

```
5  UDC> b = 10 ; c = 20
```

The variables that a depends on are set with two commands on one line.

```
6  UDC> ?{a}
7  30
```

An interrogation of the evaluation of a is made.

```
8  UDC> b = a + c
9  circular definition
```

a depends on b so b=(b+c)+c which is a circular definition and so rejected.

```
10  UDC> b = {a} + c
```

However this uses the explicit value of a at the time of this assignment.

```
11  UDC> ?{b}
12  50
```

An interrogation of the evaluation of b is made.

```
13  UDC> b = {b} ^ 2
```

This assigns to b the explicit value of a function of b which is numeric.

```
14  UDC> ?b
```

```
15  50 ^ 2
```

b is here interrogated - note this is not an interrogation of b evaluated

```
16  UDC> ?{b}
17  2500
```

and here the interrogation is of the evaluation of b

```
20  UDC> a = {b / 20 \/ 120}
```

a is set to the lower of the two numbers defined by b/20 and 120.

```
21  UDC> ?{a}
22  120
```

The result is printed.

```
23  c = 15
24  UDC> b = if (a/c) * c==a then 1 else 0
```

This function will return 1 if c is a factor of a.

```
25  UDC> ?{b}
26  1
```

So 15, the value of c, is a factor of 120, stored in a.

```
27  UDC> A(
28  invalid expression
```

Incomplete expression entered so error.

```
29  UDC>
30  UDC>
```

Blank entries.

```
31  UDC> ?A(0)
32  @
```

A previously undefined variable is interrogated.

```
33  UDC> ?A(f)
34  undefined array index
```

Error caused by not knowing which variable in the array is being interrogated.

```
35  UDC> f = 9 ; ?A(f)
36  @
```

So defining the variable representing the index allows us to interrogate A(f).

```
37  UDC> ?b
38  if ( a / c ) * c == a then 1 else 0
```

b is currently assigned to this formula.

```
39  UDC> A(0)=A(1)+A(100)
40  invalid vector reference
```

Attempt to use a vector variable whose index is above the highest legal index.

```
41  UDC> A(0)=A(1)+A(99)
```

In this implementation 99 is the default.

```
42  UDC> if a > 5 then 1 else 0
43  invalid start
```

This statement is neither an assignment nor an interrogation.

```
44  UDC> b = if a > 5 then 1 else 0
45  UDC> ?b
46  if a > 5 then 1 else 0
47  UDC> ?{-(!0)}
48  -1
```

This demonstrates the combination of unary operators.

```
49  UDC> ?----3
50  incomplete expression
```

Although strictly legal, ----3 is a bit confusing without parentheses.

```
51  UDC> ?-(-(-(-3)))
52  3
```

Use of parentheses makes the expression allowable, though pointless.

```
53  UDC> ?a+b
54  a + b
```

Here the input function is being interrogated - very pointless.

```
55  UDC> ?{a+b}
56  -880
```

The same function is interrogated for an explicit value using braces.

```
57  UDC> ^D
```

Exit from udc is made using the CTRL-D symbol at the beginning of a new line.

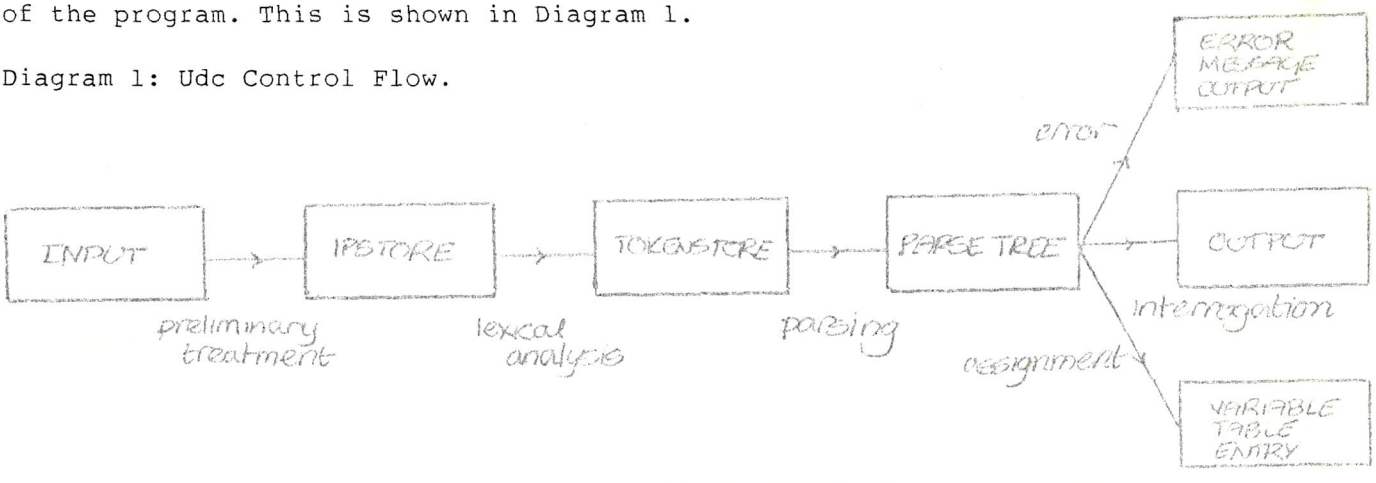
```
58  $
```

Section 2: Program Documentation.

2.0: Introduction.

The computations involved in the running of the udc can be divided into several sections. In this implementation there is a strong notion of processing and inter-process stores. These alternate along the flow of control of the program. This is shown in Diagram 1.

Diagram 1: Udc Control Flow.



Thus is can be seen that every input command creates no new information to be stored, the input goes through several processes and winds up being stored either in the output file or in the variable table for interrogations and

assignments respectively. The input information to a process, the description of the process and the output information to be stored are given in the subsequent sections, though for more detailed descriptions of the mechanics of the processes involved, reference should be made to the pseudocode for these procedures [4].

2.1: Preliminary Treatment.

The first process is to read a single command from the entry from the input file, whether this be a stored file or the standard input file. In doing this, certain irrelevant chars are filtered out - tabs and spaces and certain instruction chars - semicolons and colons are acted upon and then filtered out, remaining commands inputted in the same entry are left in the input buffer or input file depending upon from where input is being sent. Semicolons mark the end of a single command within a line of several commands so this process terminates, colons result in subsequent input being taken from the next line.
e.g. From the input

```
x = :
7  7       7;y=666
```

only the single unspaced command 'x=777' is passed on, on first execution of this procedure, on second execution 'y=666' is passed.
Each char read is either registed as a passable symbol or an unpassable one, it is the passable chars that are transferred one at a time to the command storage array. After which this process terminates.

2.2: Lexical Analysis.

2.2.0: Introduction.

This process lifts the symbols stored in the command storage array one at a time, performs a lexical analysis, converting the command into a representation more easily manipulated by the subsequent parsing process and then stores this representation in the token storage array.

2.2.1: What is Lexical Analysis ?

To understand the concept of lexical analysis we need some preliminary definitions.
Every input symbol can be regarded as a token or as part of a token. Every token is an ordered pair consisting of a lexical value - that is what type of symbol the token represents and a possibly redundant semantic value. Table 4 gives the lexical values and semantic definitions of the possible tokens catered for in this program.

Table 4: Token Definitions.

symbol	lexical value	semantic definition
?	QM	0
=	ASSIG	0
(	LPAREN	0
)	RPAREN	0
{	LBR	0
}	RBR	0
[a-z]	VAR	<alphabetic index>
[A-Z](	VEC	<alphabetic index>
[0-9]*	NUM	<numeric value>
@	UNDEF	0
if	IF	0
then	THEN	0
else	ELSE	0
/\	MAX	0
\/	MIN	0
^	TTPO	0
%	MOD	0

*	MULT	0
/	DIV	0
+	PLUS	0
-	if preceding token's lexical value is NUM, VAR, UNDEF, RBR or RPAREN:	
	MINUS	0
	otherwise:	
	UNMINUS	0
<	LT	0
<=	LE	0
==	EQ	0
>=	GE	0
>	GT	0
&	AND	0
|	OR	0
!	NOT	0
<command end>	EOC	0
<other symbols>	ERR	0

e.g. Consider as an example the following input command

E(3) = 123 + (a/\b)

The lexical and semantic token response to this would be

<VEC,4>
<NUM,3>
<ASSIG,0>
<NUM,123>
<PLUS,0>
<LPAREN,0>
<VAR,0>
<MAX,0>
<VAR,1>
<RPAREN,0>
<EOC,0>

In addition to these tokens, which result from lexical analysis of the input, several other tokens are used by the program. These help in parsing [2.3] the input command. These are not derived by the lexical analysis procedure but as tokens, they are relevant to this section. Table 5 shows what these additional tokens are.

Table 5: Parse-Constructed Tokens.

lexical value	semantic definition	purpose
OPBASE	0	operator to sit at bottom of operator stack, that has low enough precedence as to prevent total reduction of expression stack.
BUENDSYM	0	operator to be placed on top of operator stack, that has low enough precedence to force reduction of expression stack using up all operators on operator stack except the OPBASE operator.
IFACTIONS	0	This is a totally redundant token both for the user and the running of the program. It serves only to label a previously unlabeled node in the parse tree constructed by the parsing procedure, maintaining the aesthetics of the parse tree.

For more details on these additional tokens see the section on parse trees [2.3.2].

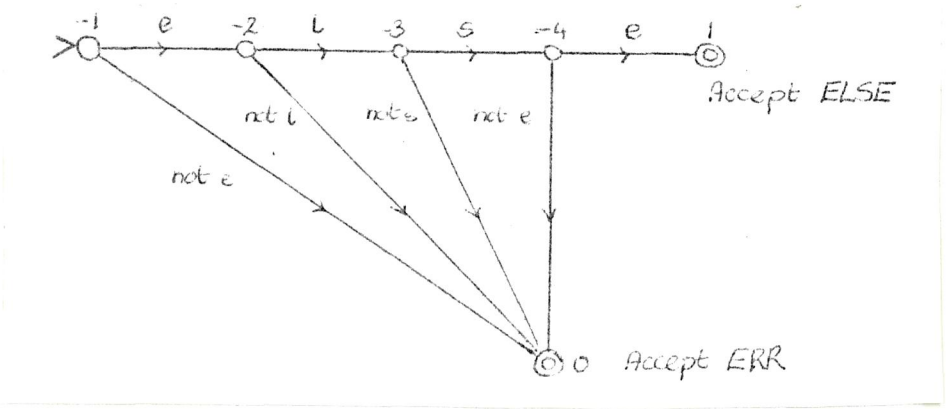
2.2.2: Construction of the State Table.

The lexical analysis state table serves to show which stage of parsing a

stream of characters, to detect a token, has been reached. The state table is the combination of several smaller state tables each of which parses only one token.

e.g. Let us consider the finite automaton that is needed to parse the characters that constitute the else token. It needs to move towards a recognition state as it encounters the letters 'e', 'l', 's' and 'e' consecutively. But if at any stage we encounter an unexpected symbol in the input then we must move into a rejection state and remain there whatever the subsequent input. The automaton we need has instructions as shown in Diagram 2.

Diagram 2: Finite Automaton to Parse the Token 'else'



This can also be represented in the table below. The table shows the new state entered depending on the current state and the next character read off the input store.

Table 6: State Transition Table To Parse The Token 'else'

Current State	Input Characters e	l	s	others
1	<HALT>			
0	<HALT>			
-1	-2	0	0	0
-2	0	-3	0	0
-3	0	0	-4	0
-4	1	0	0	0

In this table, 0 is the start state, 1 is the rejection or recognise error token state, and 2 is the recognise 'else' state.

Having devised mini-lexical analysers for each token, these are then combined making sure that each token recognition state is unique. In the final lexical analysis state transition table [4.3.4] negative states indicate partially parsed tokens, 0 represents the general error state and positive states represent successfully parsed tokens. The possible states range from -6 to +32 and the possible chars range over all ascii characters.

2.3: Parsing

2.3.0: Introduction.

Parsing is the last stage in the udc's manipulation of the input file. It takes tokens one at a time out of the token storage array, performs a parse and either passes out the result of an interrogation, records an assignment in the variable table or alerts the user of an error in the input sequence.

2.3.1: What is a Parser ?

Parsing is the combination of two things. The first of these is in some sense a boolean function of an input command, returning the syntactically directed translation of the input command if it is syntactically correct on the level of the BNF specification for a command [4.1.2] and above this level, checks

that the execution of this command will not cause errors such as division by zero or infinite loops. The second part of parsing is the inclusion of an appropriate instruction set, for execution during the first check.
e.g. Consider a typical command entry

```
? { 23*a + b }
```

This, to anyone familiar with the udc operator set, obviously constitutes a legal command. Thus it must be accepted as such by the parser. Apart from registering it as a legal command, the parser must also perform the following calculation;

what is the value of the function defined by the expression '23*a + b'

An expression is considered legal if a parse tree can be constructed for it.

2.3.2: What is a Parse Tree ?

A parse tree is the program's internal representation of an expression. An expression is represented as a n-ary graph. The value of n depends upon the application but for most it is 2. Each node on the graph represents a token, Those nodes that are leaves denote operands, those nodes that are internal represent operators. For most of the operators this binary tree representation is fine since most of the operators are binary, unary operators result in the binary tree being incomplete at that node. Ternary operators can be represented, but their representation is not elegant. Thus any token can be represented in a tree node that carries three pieces of information: the lexical value of the token held at that node, the semantic value of the token held at that node and thirdly, the token's relationship to other tokens within the expression. This final piece of information is held in two fields of a node's store, each of which points to sub-expressions that the token acts on. Thus each node carries four pieces of information, two of which - the lexical and semantic values will be the same as given in Table 4 above. These fields are labeled 'lex' and 'sem' respectively in the node's information store. The remaining pieces of information are denoted 'lch' and 'rch' for the first and second subexpressions of the token. The particular form that these take are given in the table below where the tokens are divided into four main classes.

Table 7: Token Representation in Tree Nodes.

Token Type	lch	rch
Operand	-	-
Unary operator	pointer to operand	-
Binary operator	pointer to 1st operand	pointer to 2nd operand
if-then-else	pointer to condition	pointer to IFACTIONS node

These constructs are best explained by diagrams of the trees.

Diagram 3a: Token Representation In Tree Nodes: Unary Operators.

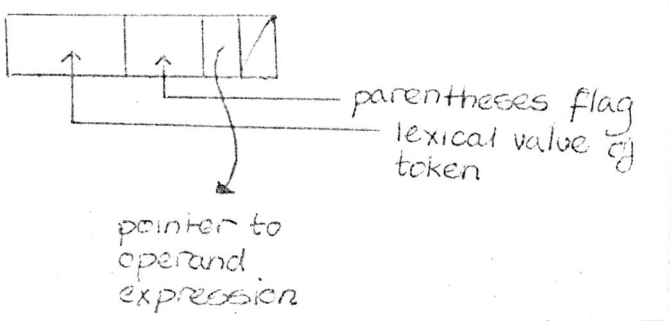


Diagram 3b: Token Representation In Tree Nodes: Binary Operators.

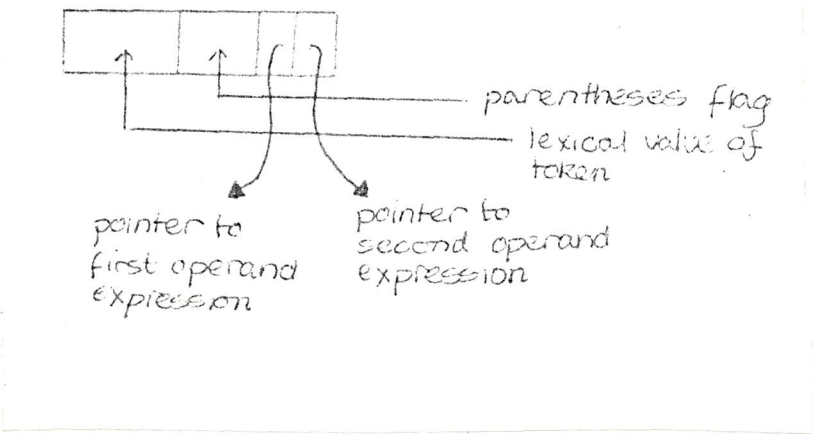


Diagram 3c: Token Representation In Tree Nodes: Ternary Operators.

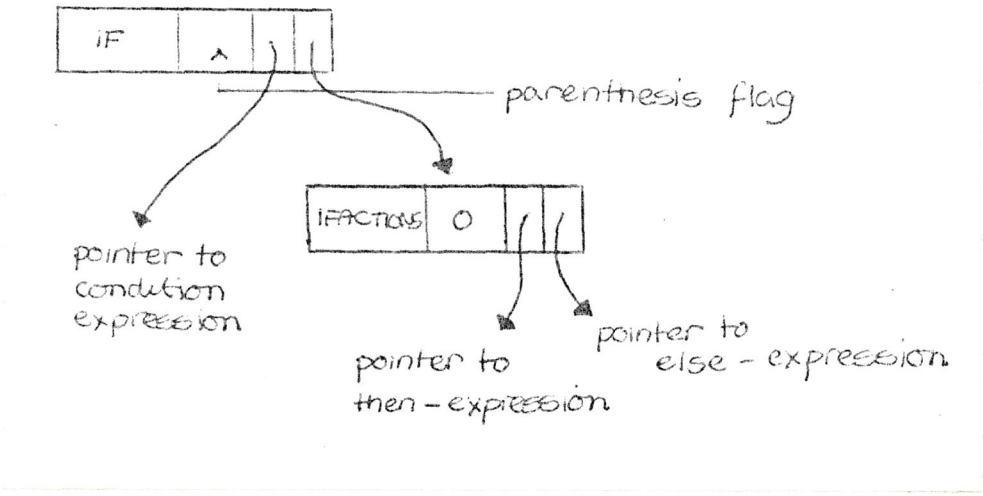
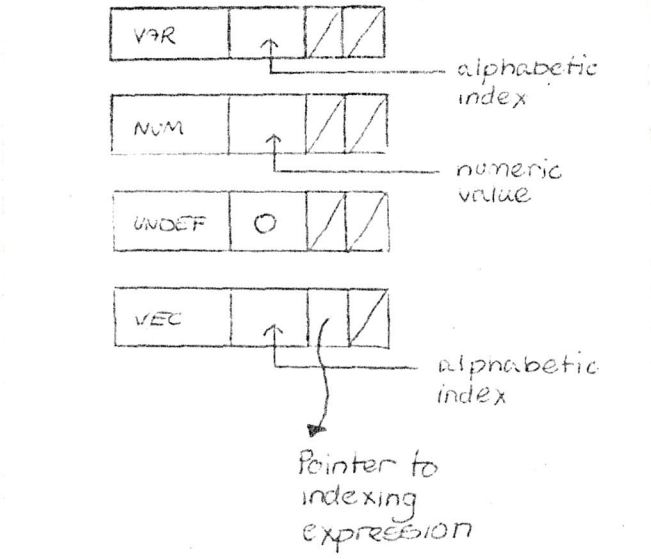


Diagram 3d: Token Representation In Tree Nodes: Operands.



There are a few other tokens that must be catered for in an expression. These are parentheses and braces. Parentheses are not actually stored in the parsing tree but instead when they are encountered then the semantic value of the operator at the expression that the parentheses act on is used as a

parentheses flag; the semantic value of any operator is normally 0 when the flag is set it takes on the value 1. Braces are not represented at all in the tree. A braced expression is evaluated [4.3.12].

e.g. Consider the following expressions and their appropriate tree representations.

Diagram 4a: Binary Tree Representation of -a.

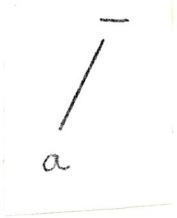


Diagram 4b: Binary Tree Representation of -3*b+c.

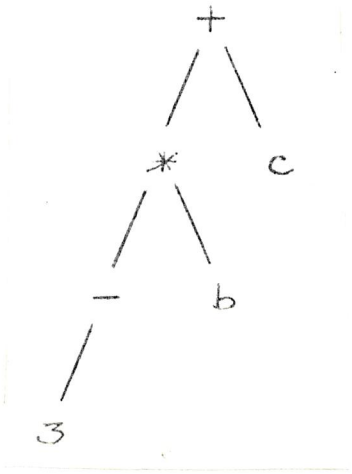
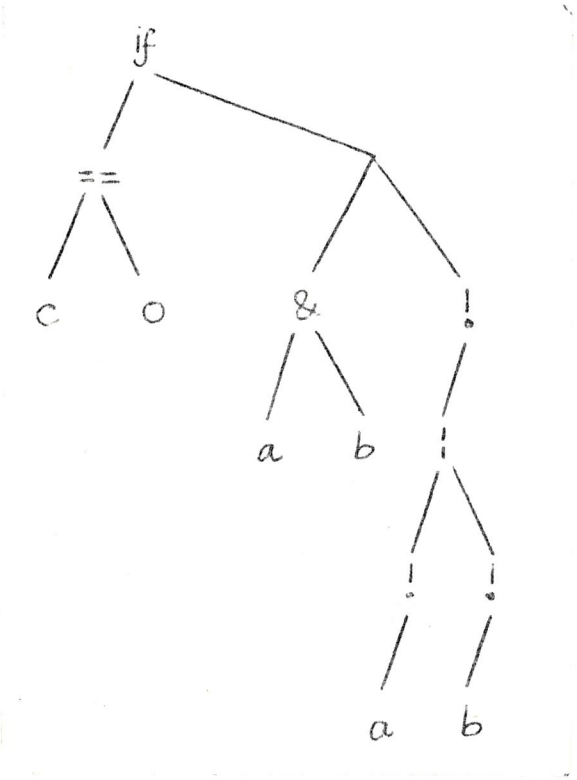


Diagram 4c: Binary Tree Representation of if c==0 then a&b else !(!b|!a).



These parse trees can be constructed by several methods. The most common of these are topdown parsing and bottom up parsing. The choice of parsing technique depends on the application or more particularly, the style of the BNF for the input.

2.3.3: Top Down Parsing Methods.

The basic idea of top down parsing is to construct the parse tree starting at the root and work down attaching nodes derived from the input to the existing nodes of the tree until the input is exhausted and the parse tree complete. Symbols on the input stream are matched to the possibilities defined by the grammar. When only one rules defines the form of the input then as much as possible of the input is matched and the process continues. When more than one rule is found that matches the current part of the input then these are all taken up and all but one are subsequently discarded upon realisation of their inappropriateness.
The sentence definitions involved in the top down parsing process of the udc are determined by the following rules. E represents an expression which will force a bottom up parse of subsequent input when the previous symbol has been passed.

Table 8: Rules for the First Stage Parse.

```
S ::= '?' E
    | <var> '=' E
```

Table 9: Rules for the Top Down Parsing Procedure.

```
S ::= <vec> E ')'
    | '(' E ')'
    | '{' E '}'
    | 'if' E 'then' E 'else' E
```

In both tables it can be seen that the first symbol in the input determines which rule is followed since no two rules have the same first symbols on their right hand sides.

2.3.4: Bottom Up Parsing Methods.

The basic idea of bottom up parsing is the exact reverse of top down parsing, it consists of constructing the parse tree starting at the leaves and working up combining roots of subtrees to make larger trees until there is only one tree - the completed parse tree.
A particular type of bottom up parsing is that designed for operator precedence grammars. In this an expression is considered as a hierarchy of subexpressions. The hierarchical position of each subexpression is determined by the precedence of the operator that binds the subexpressions at that level. This method is akin to saturating an expression with brackets until the order of evaluation is determined by the order of the brackets, operators and operands, rather than their relative precedences, though the precedences are used as rules by which the bracketing is determined. The rules for an operator precedence grammar are much simpler than any other type of parser since only two operations are possible. If the symbol on the input is an operand (possibly derived from a top down parse [4.3.16]) then it placed on an expression stack, if the symbol is an operator then it is placed on the operator stack having first successively combined expressions on the expression stack with operators from the top of the operator stack if they are of higher precedence than the operator on the input. A bottom up parse is completed by the reduction of the expression stack successively using the top operator on the operator stack until there are no operators on the operator stack and only one expression on the expression stack - this being the completed parsed expression.

2.3.5: Hybridisation.

Because of the inclusion in the udc command set of some operators and commands

that are best parsed by top down methods and some that are best parsed by top down methods, this implementation, instead of bending the rules of top down parsing to allow for those features which are parsed more easily by bottom up methods or bending the rules of bottom up parsing to allow for those features which are parsed more easily by top down methods, uses both methods, flipping between the two depending on what the characteristics of the next token on input is. The parser classifies all tokens into classes given in the following table. Depending on which class an encountered token falls into, the program decides either to maintain the current parsing technique or switch to the other technique, calling it as a subroutine. The default parsing technique, that is the one that any expression is first parsed by once the nature of the command is determined, is the bottom up parser.

Table 10: Parser Switching Token Types.

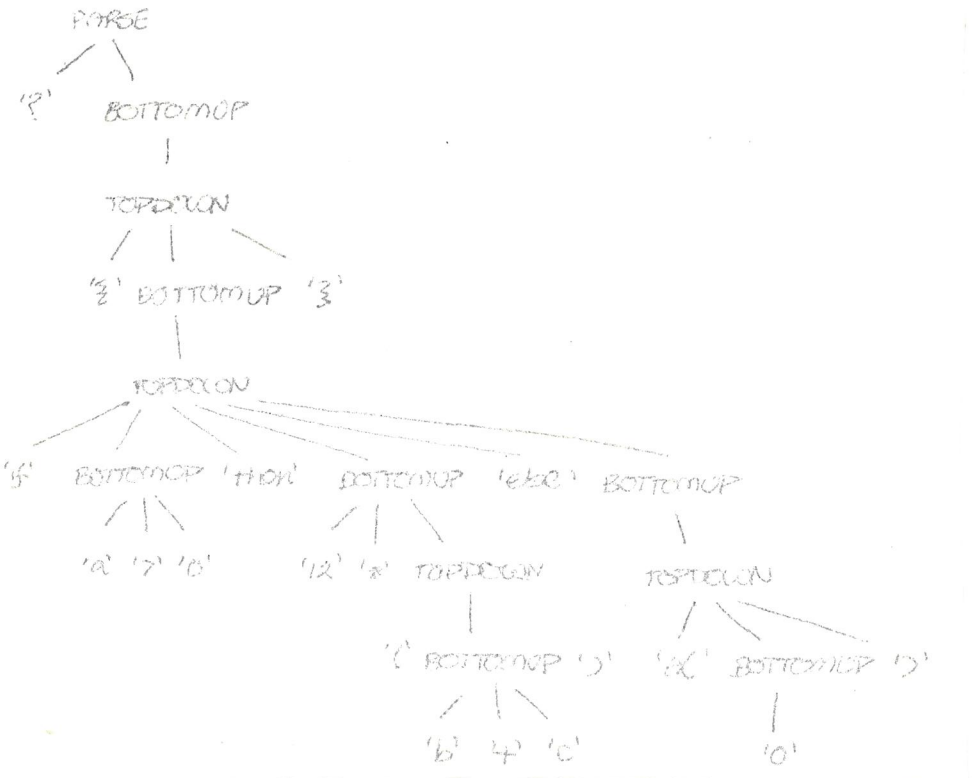
Token class	Elements	Action
TD Starters	VEC, LPAREN, LBR, IF	Start top down child parsing process.
BU Enders	RPAREN, RBR, THEN, EOC, ELSE	Return to parent parsing process.
BU Middlers	NOT, TTPO, MOD, MULT, DIV, PLUS, LT, LE, EQ, GT, MAX, MIN, GE, AND, OR, UNMINUS, MINUS, VAR, NUM, UNDEF	Stay in bottom up parser - no switch.
Inappropriate	ASSIG, QM, ERR	Call error procedure.

e.g. Consider the following expression and how it would be parsed.

?{ if a>0 then 12 * (b+c) else A(0) }

The parsing sequence for this command would involve the nesting of several parsing processes using both parsing procedures. The sequence in which these procedures would be called and returned from can be represented by the diagram below.

Diagram 5: Parsing Calls For '?{ if a>0 then 12 * (b+c) else A(0) }'.



2.3.6: Parsing Instructions.

Other than the tree construction there are several other operations to be done. These together with explanations of what is involved in the process are given below.

a) Evaluation [4.3.12].
   This is the reduction of a section of the parse tree to a single cell containing either the lexical value UNDEF for the undefined value or the lexical value NUM and the semantic value defined by the number. The way that an expression or subexpression is evaluated from the parse tree is by recursively evaluating the subtrees of this subtree. If a subtree is undefined then the parent is also undefined, and if both subtrees are defined numerically then the parent is defined as the combination of these numbers using the operator at the parent node. If a particular node is a variable then it is replaced by the expression tree that the variable represents as defined in the variable table.

When the parse has finished and the root to an expression subtree is returned to the parse procedure [4.3.19] then either an interrogation or an assignment is made.

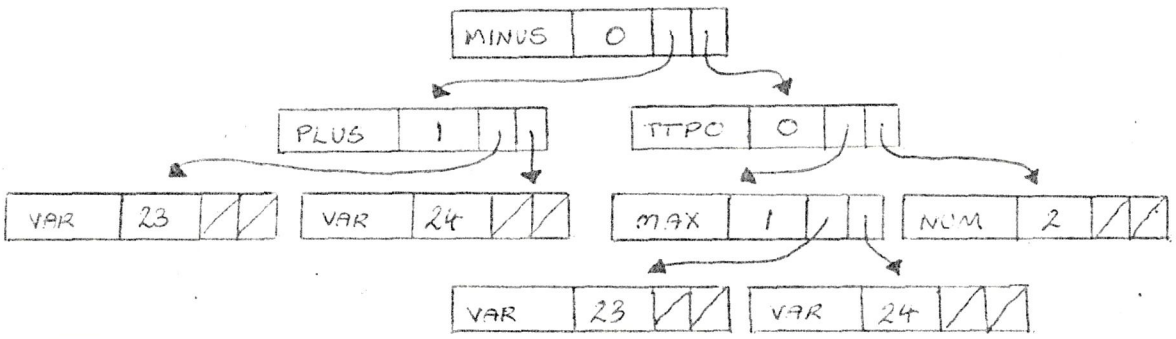
b) Interrogation or Tree Traversal [4.3.10].
   This is the printing of the expression represented by a tree. There are three ways in which a tree could have been printed - postfix, infix, prefix. Each of these involves printing the symbol at a node and recursively doing the same on the left and right subtree should they exist. The difference between these three printing formats comes about from the order in which the tree is traversed. The orders that these actions are carried out in is given for each printing format in the table below. Since infix is the format in which the expression was entered, this was been chosen as the output format.

   Table 11: Different Printing Procedure Procedure Calls.

Format	Order of procedure calls
Prefix:	Prefix(left-subtree);Prefix(right-subtree);PrintNode
Infix:	Infix(left-subtree);PrintNode;Infix(right-subtree)
Postfix:	PrintNode;Postfix(left-subtree);Postfix(right-subtree)

   e.g. Consider the expression '(x+y) - (x/\y)^2' printed in these three formats.

   Diagram 6: Parse Tree for '(x+y) - (x/\y)^2'.



   Prefix output: - + x y ^ /\ x y 2
   Infix output: (x + y) - (x /\ y) ^ 2
   Postfix output: x y + x y /\ 2 ^ -

c) Assignment.
   When an assignment is made then a pointer to the derived expression tree is inserted at the appropriate place in the variable table.

2.4: Error Handling.

There are three main classes of errors, these are listed below.

a) Syntactic errors: these are caused by unknown or unexpected symbols, or commands with missing or surplus symbols. This error has no specific checking routine, though it uses PassSym [4.3.15] and GetToken [4.3.13] to cope with missing expected symbols and unknown symbols respectively. In the event of an error of this type the error [4.3.9] procedure is called.

b) Illegal assignments: caused by circular definitions - that is an attempt to assign to a variable an expression dependent, directly or indirectly, on that same variable. The function SDV [4.3.11] (Self-Dependent Variable) returns the boolean value true if a variable in assignment is self-dependent either because of the assigned expression or the expression assigned to some variable in the expression. The error [4.3.9] handling procedure is called.

c) Division by zero: caused by the second operand to the division or modulus operator being zero. There are checks in the expression evaluation procedure Eval [4.3.12] concerning the value of the second operand to these operators before any attempt to combine the two operands is made. Again the error handling procedure is called.

The error handling procedure after outputting an error message, uses the 'goto' command to drop control from the current position, quickly pass out of all procedures that lead from the first call to the current level, and exit from the parser. Although this method means a loss in the modularity of the program, it does what is required quickly and effectively.

2.5: Initialisation.

When first started up, the udc prompts the user to start entering commands on the assumption that the computer will have executed all the initialisation procedures and be ready to deal with the first command well before the user has finished typing the first command. The initialisation process includes the setting up of the following tables.

a) The lexical analysis state transition table.
This is a table of the new states that the lexical analyser enters based on two pieces of information - the previous state and the current character on the input.

b) The lexical token code table.
Essentially this is a table of the final states of the lexical analyser and the associated lexical token. It facilitates the storage of tokens in a non-numeric coding, making the program much easier to understand.

c) The operator precedence table.
This is a table in which all of the unary and binary operators - that is the stack based operators are assigned a number, positioning their precedences relative to each other.

d) The variable table.
This is the table in which the values of each of the 26 simple variables and 26 vector variables with all their indexes are stored. All variables are initially set to the undefined value.

Section 3: Development.

3.1: Problems and Novelties in Program Development.

a) BNF Adaptation.
The adaptations made to the original BNF specification served two purposes. Firstly it was necessary to introduce arrays of variables, and secondly the changing of the type of expression returned from comparisons such as

```
a < b
```

which previously would have returned true or false but now returns 1 or 0 respectively, was to facilitate bottom up parsing as far as possible. This change also brought with it a greater calculating power, since these expressions can now form part of complex arithmetic expressions.

b) Input from Files.
The silent mode feature was added since when programs that prompt are run into files they normally send prompts without newline characters as well as

the expected results, this can look messy and can make the results hard to read.
e.g. Without prompt suppression the following program

Program 3: Messy Output Program.

```
a=10
b=20
c=a+b
?c ; ?{c}
```

returned to following

Udc Sequence 6a.

```
UDC> UDC> UDC> UDC> a + b
30
UDC>
```

Whereas with prompt suppression achieved by supplying an argument with the udc call, it returned

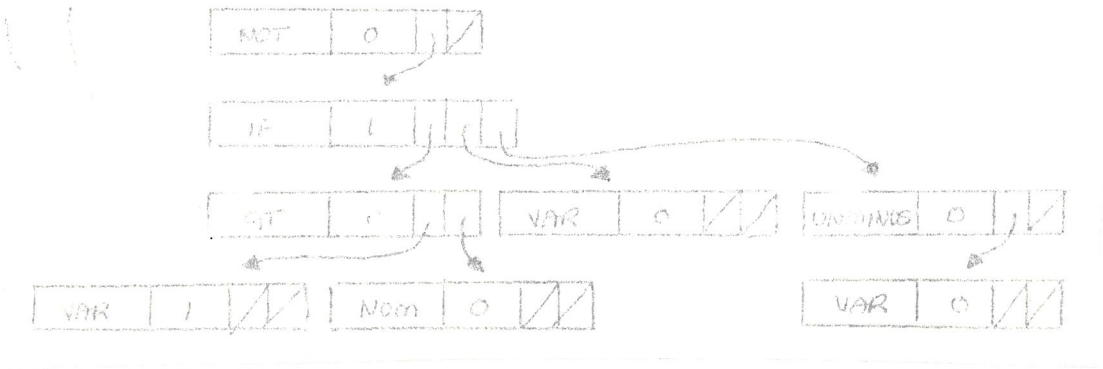
Udc Sequence 6b.

```
a + b
30
```

The disadvantage of implementing this feature in the program is that the function called in CheckArgs [4.3.1], i.e. 'argc', is not common to all versions of Pascal.

c) if-then-else Parse Tree Cell Adaptation.
For clarity within a parse tree, unary, binary and ternary tree cells would seem apt.
e.g. The expression
! (if b>0 then a else -a)
would be represented by the parse tree

Diagram 7: Parse Tree with Unary, Binary and Ternary Nodes.



But this would involve more complicated data structures for containment of the parse tree, so instead the representation of unary and ternary operators was tailored to fit the existing data structure which suited the majority of the operators, which are binary, best. The if-then-else structure can be contained in a subtree spanning two levels. As shown in Diagram 4c. The unary operators are inserted into the tree cell structure as shown in Diagram 4a.

d) Binary or Unary Minus ?
When a minus sign is encountered in the input, there must be rules to determine whether it is the unary or binary operator. The rule decided upon in this program is that if it is preceded by a right brace or parenthesis,

or by a number, variable or the undefined value the the minus sign is taken to be a binary minus. This rule works for all cases except multiple consecutive minus signs which could only be interpreted as unary minuses. Thus

```
-------4
```

is rejected, but

```
-(-(-(-(-(-4))))))
```

is not. This fault was not corrected as multiple minus signs are confusing anyway. This rule also applies to the other unary operator - logical 'not'.

e) Tables or Functions ?
The various tables referenced during execution of the udc, could more neatly have been included as functions. This was tested but proved to be much slower since the entire set of values set up by the function had to be scanned before the right entry was found, so instead tables were set up at the beginning of execution, this while slowing the program down, for the first few moments, resulted in shorter delays between command executions.

f) Compilation Problems.
In the early stages of development of this program the first prompt of the udc session was preceded by a bell-tone, however this was removed later when the source program was first compiled as it caused compilation errors.

3.2: Further Development.

a) Additional Operators.
With the addition of indexed variables, additional operators would be most useful. In particular, if the existing operators could use entire arrays of these indexed variables then they arrays would become much more useful for manipulating data.
e.g. Consider taking the average of the pairs consisting of a fixed number stored in the variable a and the twenty values contained in the array A. Conventionally, this would be performed as follows

Program 4a: Long-winded Calculations.

```
m = ( A(i) + a ) / 2
i = 0 ; ? {m}
i = 1 ; ? {m}
...
i = 20 ; ? {m}
```

However with operators for whole arrays this task would be reduced to

Program 4b: Fast Calculations.

```
M = ( A + a ) / 2
? {M}
```

The expression printing procedure would have to be altered so that it would only print the expressions assigned to an array over a certain index range. The most suitable way of defining this range would seem to be that variables within the array up to the highest indexed and defined variable are printed.
Another useful operator for arrays, not already present, would be a conditional loop variable. Say this be called 'wnz' (While Not Zero), this would have the following syntax

'wnz' '[' <channel number> ']' <expression>
<command sequence>
'end' '<channel number>

The channel number is effectively an evaluable return address for when an 'end' command with the same channel number is encountered.
e.g. Consider the following alternative method to the above problem.

Program 4c: Neater Faster Calculations.

```
b = 0
m = ( A(b) + a ) / 2
```

```
wnz [1] {21-b}
? {m}
b= {b} + 1
end [1]
```

The possibilities with this additional operator are obvious. The channel number would mean easy implementation of nested 'wnz' commands.

b) Real Arithmetic.
Computations in this calculating language would be considerably more applicable if they used real valued variables rather than integer ones. An upgraded version of this program might have real arithmetic over all but a small number of variables, denoted say by a trailing '$' on the variable name. These would typically be used as array indexes.

c) Run-time Input File Change.
A more interactive system would be attained if the input file could be switched from inside a program. This would be most useful for reading in values. Say that this command will be denoted by '#' then it would have the following syntax.

```
<var> '=' '#'
```

This would force subsequent input to be read from the standard input file until a carriage return charter appeared.
e.g. Consider the following program for taking the average of two numbers.

Program 5: Averaging Stdin.

```
a = ( b + c ) / 2
b = # ; c = #
? {a}
```

This would allow greater ease of use and flexibility of programs.
Thus consider the following program which would make use of the previous three additions to the system.

Program 6: All The Future Developments.

```
:program to find the average of any number of entered integers.

:a$ is entered number, initial value is any non-zero
a$ = 1

:t$ is running total of a$
t$ = 0

:i$ is running total number of numbers entered
i$ = -1

:enter numbers until entry is zero
wnz [1] {a$}
a$ = #
t$ = {t$ + a$}
i$ = {i$} + 1
end [1]

:calculate mean into real variable
m = t$ / i$

:evaluation and output
?{m}
```

This program could be stored in a file and if executed on a unix operating system might be executed as follows

Udc Sequence 7.

```
$ udc -silent < average.u
```

```
13
15
17
0
1.42500000000000e+01
$
```

d) Arithmetic Overflow.
There is currently no provision for an arithmetic overflow as might occur if this were typed

```
?{10^1000}
```

Unfortunately for some strange reason this does not occur. Consider the following udc execution extract

Udc Sequence 8.

```
$ udc
UDC> ?{10^1000}
0
UDC> ?{10^10}
1410065407
UDC> ?{10^15}
-1530494977
UDC> ^D
$
```

This extract also shows the gross errors incurred when using the exponentiation operator, caused by the formula used for this calculation since there is no exponentiation operator in Pascal [1.4.2f].

e) Alternative Printing Formats.
The supplying of one of the arguments "-pre" or "-post" could result in the selection of an alternative printing format - either prefix or postfix respectively, [2.3.6b], rather than the default infix.

e) Floyd Productions - An Alternative Parsing Method.
This is a parsing method attributed to R. W. Floyd (1961). It depends on matching parts of the input to the parser with a set of predefined patterns. Rules define how, once a matching pattern is found, the input is to be reduced. By successive application of these rules the input can be reduced to one symbol denoting a successful parse. The process of scanning the set of rules can be speeded up by the parsing procedure also rendering a position in the table of rules from which the search for the next rule begins. This method is flexible and fast but unfortunately the rules are very hard to work out for a given language.

Section 4: Appendices.

4.1.1: Original BNF Specification.

```
<prog>  ::= { <stat> ';' }
<stat>  ::= <var> = <exp>
          | '?' <exp>
          | <empty>
<exp>   ::= <num>
          | <exp> <binop>
          | <exp> '^' <num>
          | <exp> '%' <num>
          | - <exp>
          | 'if' <rel> 'then' <exp> 'else' <exp>
          | '[' <rel> ']'
          | '{' <exp> '}'
          | <var>
          | '@'
<rel>   ::= <exp> <relop> <exp>
          | <rel> '|' <rel>
          | <rel> '&' <rel>
          | '!' <rel>
```

```
<var>   ::= 'a' | 'b' | ... | 'z'
<num>   ::= <dig> { '0' | dig } *
<dig>   ::= '1' | '2' | ... | '9'
<binop> ::= '+'
          | '-'
          | 'S'
          | 'L'
<relop> ::= '<'
          | '<='
          | '=='
          | '>='
          | '>'
```

4.1.2: Adjusted BNF Specification.

```
<prog>  ::= { <stat> ';' }
<stat>  ::= <var> = <exp>
          | '?' <exp>
          | <empty>
<exp>   ::= <var>
          | <num>
          | <exp> <binop> <exp>
          | <unop> <exp>
          | 'if' <exp> then <exp> else <exp>
          | '@'
          | '{' <exp> '}'
          | '(' <exp> ')'
<binop> ::= '^'
          | '*'
          | '%'
          | '/'
          | '-'
          | '+'
          | '/\'
          | '\/'
          | '<`
          | '<='
          | '=='
          | '>='
          | '>'
          | '&'
          | '|'
 <unop> ::= '-'
          | '!'
<var>   ::= <svar>
          | <vec>
<svar>  ::= 'a' | 'b' | ... | 'z'
<vec>   ::= <arr> '(' <exp> ')'
<arr>   ::= 'A' | 'B' | ... | 'Z'
<num>   ::= <dig> { dig } *
<dig>   ::= '0' | '1' | ... | '9'
```

4.3: Pseudocodes For All Procedures.

4.3.1: CheckArgs

```
if there are arguments to the udc call then
    allow prompt suppression
else
    disallow prompt suppression
```

4.3.2: VarTblInit

```
for all simple variables
    make a new tree cell
    set contents to those for undefined value
```

```
    set this variable's entry in the variable table to this cell
for all vector variables
    for all allowable index values
        make a new tree cell
        set contents to those for undefined value
        set this variable's entry in the variable table to this cell
```

4.3.3: MakeOpRankTbl

define operator precedences for all unary and binary operators with unary minus as lowest and logical 'or' as highest followed by an operator to force stack reduction, followed by an irreducible stack base operator

4.3.4: MakeLexStateTbl

define new lexical analyser states from old states and current symbols for all non-error sequences, leaving remaining error sequences as default new state of zero

4.3.5: MakeLexSymTbl

define lexical values for all final states of lexical analyser and for unary minus and end of command characters

4.3.6: GetIp

```
clear input store
reset current char to single space
while the current char is not a semicolon and not at end of line
    current char is storable
    read current char
    if current char is colon then
        skip to read subsequent chars from next line
        current char is not storable
    if current char is semicolon then
        current char is not storable
    if current char is space then
        current char is not storable
    if current char is tab then
        current char is not storable
    if current char is storable
        store current char
if current char is semicolon then
    prepare to suppress prompt
else
    prepare not to suppress prompt
if end of input line then
    skip to read subsequent chars from next line
```

4.3.7: SaveToken (code)

```
store default token semantic value 0
store lexical value defined by the lexical symbol table entry for value code
if code is that for a number token then
    get numeric value of character sequence from token buffer and store
        this as semantic value
if code is that for a simple variable
    store semantic value as the variable's alphabetic position
if code is that for a vector variable
    store semantic value as the variable's alphabetic position
```

4.3.8: LexAnalyse

```
clear token store
reset input store pointer
repeat
```

```
    empty token buffer
    empty char buffer
    set current state to initialisation value
    set default tokencode for error token
    while input store pointer has not reached the end of the input store and
        current state is not the error state
        get next char from input store
        store char at end of char buffer
        get new state from state table based on old state and current char
        if current state is that for a complete token in char buffer then
            record current state in tokencode
            move contents of char buffer to end of token buffer
    if token buffer in empty then
        retreat input store pointer to be able to read in the contents of
            char buffer again
        record error token in tokencode
        advance input store pointer past current error starting char
    else
        retreat input store pointer to be able to read in the contents of
            char buffer again
    if token recorded by tokencode is minus sign and the token store is
        not empty then
        if token at end of token store is variable or number or right
            parenthesis or right brace then
            SaveToken ( number code for binary minus ) [4.3.7]
        else
            SaveToken ( number code for unary minus ) [4.3.7]
    else
        SaveToken ( tokencode ) [4.3.7]
until input store exhausted
SaveToken ( number for end of command ) [4.3.7]

4.3.9: Error ( errortype )

for the value of errortype
    print an appropriate error message

4.3.10: Traverse ( T )

if T points to an empty tree then
    no action
else
    if lexical value of T is in operator set or if and brackets flag set then
        print (
    if lexical value of T is if then
        print if
        Traverse ( pointer to left subtree ) [4.3.10]
        print then
        Traverse ( pointer to left subtree of right subtree ) [4.3.10]
        print else
        Traverse ( pointer to right subtree of right subtree ) [4.3.10]
    if lexical value of T is in binary operator set then
        Traverse ( pointer to left subtree ) [4.3.10]
        case of this operator in
            maximum:
                print /\
            minimum:
                print \/
            exponentiation:
                print ^
            modulus:
                print %
            multiplication:
                print *
            division:
                print /
```

```
            addition:
                print +
            less than:
                print <
            less than or equal to:
                print <=
            equal to:
                print ==
            greater than or equal to:
                print >=
            greater than:
                print >
            logical and:
                print &
            logical or:
                print or
            binary minus:
                print -
        Traverse ( pointer to right subtree ) [4.3.10]
    if lexical value of T is in unary operator set then
        case of this operator in
            unary minus:
                print -
            logical not:
                print !
        Traverse ( pointer to left subtree ) [4.3.10]
    if lexical value of T is number then
        print semantic value of tree cell
    if lexical value of T is simple variable then
        print character appropriate to semantic value of the tree cell
    if lexical value of T is the undefined value then
        print @
    if lexical value of T is vector variable then
        print character appropriate to semantic value of the tree cell
        print (
        Traverse ( pointer to left subtree ) [4.3.10]
        print )
    if lexical value of T is in operator set or if and brackets flag set then
        print )

4.3.11: SDV ( T )

variable self dependency false
if lexical value of T is in binary operator set then
    if SDV ( pointer to left subtree ) [4.3.11] or SDV ( pointer to right
        subtree ) [4.3.12] then
        self variable dependency true
if lexical value of T is in unary operator set then
    if SDV ( pointer to left subtree ) [4.3.11] then
        self variable dependency true
if lexical value of T is the undefined value or number
    then
        variable self dependency status false
if lexical value of T is simple variable
    then
        if this variable's dependency code is the same as that of the variable
            to be assigned to then
            variable self dependency true
        else
            set variable self dependency to SDV ( pointer extending from
                this variables's entry in variable table ) [4.3.11]
if lexical value of T is vector variable then
    if this variable's dependency code is the same as that of the variable
        to be assigned to then
        variable self dependency true
    else
```

```
        set variable self dependency to SDV ( pointer extending from
            this variables's entry in variable table ) [4.3.11]
if lexical value of T is if then
    set variable self dependency to SDV ( pointer to left subtree ) [4.3.11]
        or SDV ( pointer to left subtree of right subtree ) [4.3.11] or SDV
        ( pointer to right subtree of right subtree ) [4.3.11]

4.3.12: Eval ( T )

make new tree cell
if lexical value of T is simple variable then
    new tree cell pointer points to Eval ( pointer to this variable's entry in
        variable table ) [4.3.12]
if lexical value of T is the undefined value then
    new tree cell pointer points to this tree cell
if lexical value of T is number then
    new tree cell pointer points to this tree cell
if lexical value of T is vector variable then
    new tree cell pointer points to Eval ( pointer to this variable's entry in
        variable table ) [4.3.12]
if lexical value of T is in unary operator set then
    get pointer to Eval ( left subtree ) [4.3.12]
    if lexical value of tree cell pointed to by this pointer is the
        undefined value then
        new tree cell pointer points to tree cell with lexical value of
            the undefined value
    else
        if the lexical value of T is
            unary minus:
                new cell's semantic value is minus T's left subtree's
                    semantic value
            logical not:
                new cell's semantic value is 1 if T's left subtree's
                    semantic value is 0 or is 0 otherwise
        other fields of new cell set appropriately to contain a number
if lexical value of T is in binary operator set then
    get pointers to Eval ( left subtree ) and to Eval ( right subtree )
        [4.3.12]
    if either of the lexical value of the tree cells pointed to by this
        pointer is the undefined value then
        new tree cell pointer points to tree cell with lexical value of the
            undefined value
    else
        if the lexical value of T is
            maximum:
                new cell's semantic value is the maximum of T's subtrees'
                    semantic values
            minimum:
                new cell's semantic value is the minimum of T's subtrees'
                    semantic values
            exponentiation:
                new cell's semantic value is T'left subtree's semantic value
                    raised to the power of T's right subtree's semantic value
            modulus:
                if T's right subtree's semantic value is 0 then
                    Error [4.3.9]
                else
                    new cell's semantic value is T's left subtree's semantic
                        value modulus T's right subtree's semantic value
            multiplication:
                new cell's semantic value is the product of T's subtrees'
                    semantic values
            division:
                if the T's right subtree's semantic value is 0 then
                    Error [4.3.9]
                else
```

```
                    new cell's semantic value is the quotient of T's subtrees'
                        semantic values
            addition:
                new cell's semantic value is the sum of T's subtrees' semantic
                    values
            subtraction:
                new cell's semantic value is the difference of T's subtrees'
                    semantic values
            less-than:
                if T's left subtree's semantic value is less than T's right
                    subtree's semantic value then
                    new cell's semantic value is 1
                else
                    new cell's semantic value is 0
            less-than-or-equal-to:
                if T's left subtree's semantic value is less than or equal to
                    T's right subtree's semantic value then
                    new cell's semantic value is 1
                else
                    new cell's semantic value is 0
            equal-to:
                if T's left subtree's semantic value is equal to T's right
                    subtree's semantic value then
                    new cell's semantic value is 1
                else
                    new cell's semantic value is 0
            greater-than-or-equal-to:
                if T's left subtree's semantic value is greater than or equal
                    to T's right subtree's semantic value then
                    new cell's semantic value is 1
                else
                    new cell's semantic value is 0
            greater-than:
                if T's left subtree's semantic value is greater than T's right
                    subtree's semantic value then
                    new cell's semantic value is 1
                else
                    new cell's semantic value is 0
            logical and:
                if T's subtree's semantic values are both non-zero then
                    new cell's semantic value is 1
                else
                    new cell's semantic value is 0
            logical or:
                if either T's subtree's semantic values are non-zero then
                    new cell's semantic value is 1
                else
                    new cell's semantic value is 0
        new cell's other fields set appropriately to contain a number
if T's lexical value is if then
    get pointer to Evaluation ( left subtree ) [4.3.12]
    if this points to an undefined value then
        new tree cell pointer points to tree cell with lexical value of the
        undefined value
    else
        if the semantic value in the tree cell this points to is non-zero then
            get pointer to Evaluation ( T's right subtree's left subtree )
                [4.3.12]
            if this points to an undefined value then
                new tree cell pointer points to tree cell with lexical value
                    of the undefined value
            else
                new tree's semantic value is semantic value of the evaluation
                    of T's right subtree's left subtree
                new cell's other fields set appropriately to contain a number
        else
```

```
            get pointer to Evaluation ( T's right subtree's right subtree )
                [4.3.12]
            if this points to an undefined value then
                new tree cell pointer points to tree cell with lexical value
                    of the undefined value
            else
                new tree's semantic value is semantic value of the evaluation
                    of T's right subtree's right subtree
                new cell's other fields set appropriately to contain a number
return new cell
```

4.3.13: GetToken

```
advance token store pointer forward one
record the semantic and lexical values of the token now being referenced
if it's lexical value is of an error symbol then
    Error [4.3.9]
```

4.3.14: BackSym

```
move token store pointer back one
```

4.3.15: PassSym (passme)

```
GetToken [4.3.13]
if the lexical value of the current token is not the same as passme's then
    Error [4.3.9]
```

4.3.16: TopDown

```
GetToken [4.3.13]
case of its lexical value in
    array variable:
        make new tree cell
        set lexical and semantic values to those of the array variable
        define left subtree as Evaluation [4.3.12] of subsequent expression
            obtained by BottomUp [4.3.18] parsing
        if left subtree is undefined then
            Error [4.3.9]
        define no right subtree
        return pointer to the new cell
    left parenthesis:
        return pointer to subsequent expression obtained by BottomUp [4.3.18]
            parsing
        PassSym ( right parenthesis ) [4.3.15]
    left brace:
        return pointer to Evaluation [4.3.12] of the subsequent expression
            obtained by BottomUp [4.3.18] parsing
    if:
        make new tree cell
        set lexical and semantic values to those of the if token
        set left subtree to pointer to subsequent expression obtained by
            BottomUp [4.3.18] parsing
        PassSym ( then ) [4.3.15]
        make new tree cell attached to right subtree of previous cell
        set lexical and semantic values to those of the if-actions token
        set left subtree to pointer to subsequent expression obtained by
            BottomUp [4.3.18] parsing
        PassSym ( else ) [4.3.15]
        set right subtree to pointer to subsequent expression obtained by
            BottomUp [4.3.18] parsing
        return pointer to if cell
```

4.3.17: Reduce

```
pop operator stack
```

```
make new tree cell
if the popped operator is unary then
    if there are insufficient expression pointers on the expression stack for
        this operator to act upon then
        Error [4.3.9]
    pop expression stack
    insert popped operator's lexical and semantic values in tree cell
    insert popped expression pointer as left subtree and no right subtree
else
    if there are insufficient expression pointers on the expression stack for
        this operator to act upon then
        Error [4.3.9]
    pop expression stack twice
    insert popped operator's lexical and semantic values in tree cell
    insert popped expression pointers as left and right subtree
push pointer to new cell onto expression stack
```

4.3.18: BottomUp

```
reset expression stack
push irreducible token on operator stack
default procedure exit permission false
repeat
    GetToken [4.3.13]
    case of its lexical value in
        operator set:
            while operator rank of this token is greater than that of operator
                on top of operator stack
                Reduce [4.3.17] operator stack
        variable, number or undefined value:
            make a new tree cell
            set lexical and semantic values of this tree cell to those of the
                current token
            define no subtrees
            push pointer to this tree cell on to expression stack
        vector variable, left parenthesis, left brace or if:
            BackSym [4.3.14] to enable rereading of this token by another
                procedure
            push the pointer to expression derived from a top down parse onto
                the expression stack
        right parenthesis, right brace, then, end-of-command,
            else:
                BackSym [4.3.14]
                overwrite current token's lexical value with the
                    full-stack-reduction token of higher precedence than
                    normal operators
                while operator rank of this token is greater than that of
                    operator on top of operator stack
                    Reduce [4.3.17] operator stack
                set procedure exit permission true
        assig or question mark:
            Error [4.3.9]
until procedure exit permission true
if operator stack clear and 1 expression pointer on expression stack then
    return this pointer value
else
    Error [4.3.9]
```

4.3.19: Parse

```
reset token store pointer
GetToken [4.3.13]
if its lexical value is
    simple variable:
        record variable to be assigned to
        PassSym ( assignment sign ) [4.3.16]
```

```
        get pointer to subsequent expression parsed BottomUp [4.3.18]
        make unique code for assigned variable
        if SDV [4.3.11] returns that this expression depends on a variable
            with the same unique code then
            Error [4.3.9]
        else
            store pointer to expression in variable table
    question mark:
        get pointer to subsequent expression parsed BottomUp [4.3.18]
        if this expression is a single simple variable then
            revise pointer to point to the expression pointed to by this
                variable in the variable table
        if this expression is a single vector variable then
            revise pointer to point to the expression pointed to by this
                variable in the variable table
        Traverse [4.3.10] the expression now pointed to
    vector variable:
        record variable to be assigned to
        get explicit value of index using Eval [4.3.12]
        if this value is undefined then
            Error [4.3.9]
        if this value is negative or above defined maximum index then
            Error [4.3.9]
        PassSym ( right parenthesis ) [4.3.16]
        PassSym ( assignment sign ) [4.3.16]
        get pointer to subsequent expression parsed BottomUp [4.3.18]
        make unique code for assigned variable
        if SDV [4.3.11] returns that this expression depends on a variable
            with the same unique code then
            Error [4.3.9]
        else
            store pointer to expression in variable table
    all other symbols:
        Error [4.3.9]
PassSym ( end of command ) [4.3.15]

4.3.20: Init

CheckArgs [4.3.1]
if prompts not being suppressed then
    give prompt
VarTblInit [4.3.2]
MakeLexStateTbl [4.3.4]
MakeLexSymTbl [4.3.5]
define the operator set
define the binary operator set
define the unary operator set

4.3.21: Main

Init [4.3.20]
while input file non-empty
    GetIp [4.3.6]
    if command longer than RETURN then
        LexAnalyse [4.3.8]
        Parse [4.3.19]
    if only 1 command on line and prompts not being suppressed then
        give prompt
if prompts not being suppressed then
    print a blank line to clear CTRL-D char

Section 5: The Pascal Udc Source Code.

program UDC(input, output);
const
    maxchrln = 256;                                {max chars in line}
```

```
    maxsymln = 256;                                  {max symbols in a line}
    maxstkht = 256;                                  {max exp stack height}
    maxref = 99;                                     {max vec ref}
type
    chrbuffer = array [1..maxchrln] of char;         {char buffer}
    lextype = (VEC,                                  {vec var token}
        LPAREN,                                      {left bracket token}
        LBR,                                         {left brace token}
        IF,                                          {if token}
        RPAREN,                                      {right bracket token}
        RBR,                                         {right brace token}
        THEN,                                        {then token}
        ELSE,                                        {else token}
        EOC,                                         {end-of-input token}
        ASSIG,                                       {assignment token}
        QM,                                          {query token}
        MAX,                                         {maximum token}
        MIN,                                         {minimum token}
        TTPO,                                        {'to-the-power-of' token}
        MOD,                                         {modulo token}
        MULT,                                        {multiply token}
        DIV,                                         {division token}
        PLUS,                                        {addition token}
        LT,                                          {less-than token}
        LE,                                          {less-or-equal-to token}
        GT,                                          {greater-than token}
        EQ,                                          {equal-to token}
        GE,                                          {greater-or-equal-to token}
        AND,                                         {logical-and token}
        OR,                                          {logical-or token}
        MINUS,                                       {subtraction token}
        UNMINUS,                                     {unary minus token}
        NOT,                                         {logical-not token}
        VAR,                                         {var token}
        NUM,                                         {num token}
        UNDEF,                                       {undef-value token}
        BUENDSYM,                                    {exp-end dummy token}
        OPBASE,                                      {non-reducible-op token}
        ERR,                                         {unrecognised symbol token}
        IFACTIONS);                                  {then-else clause dummy token}
    lexst = -7..32;                                  {lex analysis table states}
    accst = 0..34;                                   {lex analysis accept states}
    dataptr = ^ datalocation;                        {parse-tree cell struc def}
    datalocation =
        record
            lch, rch: dataptr;                       {cell sub tree ptrs}
            lex: lextype;                            {token's lex value}
            sem: integer                             {token's sem value}
        end;
var
    opset: set of lextype;                           {op set}
    binopset: set of lextype;                        {binary op set}
    unopset: set of lextype;                         {unary op set}
    lexsttbl: array [lexst, char] of lexst;          {lex state table array}
    lexsymtbl: array [accst] of lextype;             {lex symbol table array}
    opranktbl: array [lextype] of integer;           {op prec table array}
    tokenb: chrbuffer;                               {encountered token buffer}
    charb: chrbuffer;                                {incomplete token buffer}
    cpos: integer;                                   {end of char buffer mkr}
    tpos: integer;                                   {end of token buffer mkr}
    x: char;                                         {next char read}
    cs: lexst;                                       {current lex state}
    tokencode: lexst;                                {lex code for token in tokenb}
    tokenstoretop: integer;                          {end of token store mkr}
    ipstore: chrbuffer;                              {command store}
    ipstoretop: integer;                             {end of command store mkr}
```

```
tokenstore: array [1..maxsymln] of          {command store}
    record
        lex: lextype;                       {lex val field}
        sem: integer                        {sem val field}
    end;
tempptr: dataptr;                           {temp ptr}
vartbl: array [0..25] of dataptr;           {var table}
vectbl: array [0..25, 0..maxref] of dataptr;{vec var table}
scomm: boolean;                             {single command line}
sprom: boolean;                             {silent mode}

procedure CheckArgs;
{checks for program call arguments to suppress prompting and nice
printing}

begin
    if argc > 1 then                        {if arguments present}
        sprom := true                       {facilitate silent mode}
    else
        sprom := false                      {facilitate verbose mode}
end; { CheckArgs }

procedure VarTblInit;
{sets all variables and vector variables to point to cells defining
undefined values}

var
    i: char;                                {loop var}
    j: integer;                             {loop var}
    cellptr: dataptr;                       {general cell}
begin
    for i := 'a' to 'z' do begin            {for all vars}
        new(cellptr);                       {make new cell}
        cellptr^.lex := UNDEF;              {set to undef}
        cellptr^.sem := 0;                  {set default sem val}
        cellptr^.lch := nil;                {no left branch}
        cellptr^.rch := nil;                {no right branch}
        vartbl[ord(i) - ord('a')] := cellptr{var point to cell}
    end;
    for i := 'A' to 'Z' do                  {for all vec var}
        for j := 0 to maxref do begin       {for all vec refs}
            new(cellptr);                   {make new cell}
            cellptr^.lex := UNDEF;          {set to undef}
            cellptr^.sem := 0;              {set default sem val}
            cellptr^.lch := nil;            {no left branch}
            cellptr^.rch := nil;            {no right branch}
            vectbl[ord(i) - ord('A'), j] := cellptr
                                            {var points to cell}
        end
end; { VarTblInit }

procedure MakeOpRankTbl;
{set precedences of operators}

begin
    opranktbl[UNMINUS] := 0;                {unary minus strongest bind}
    opranktbl[NOT] := 0;
    opranktbl[TTPO] := 1;
    opranktbl[MULT] := 2;
    opranktbl[MOD] := 2;
    opranktbl[DIV] := 2;
    opranktbl[MINUS] := 3;
    opranktbl[PLUS] := 3;
    opranktbl[MAX] := 4;
    opranktbl[MIN] := 4;
    opranktbl[LE] := 5;
```

```
    opranktbl[LT] := 5;
    opranktbl[EQ] := 5;
    opranktbl[GT] := 5;
    opranktbl[GE] := 5;
    opranktbl[AND] := 6;
    opranktbl[OR] := 7;                    {or weakest bind}
    opranktbl[BUENDSYM] := 8;              {dummy op to force redn}
    opranktbl[OPBASE] := 9                 {dummy op to halt redn}
end; { MakeOpRankTbl }

procedure MakeLexStateTbl;
{set non-error states in lexical analysis transition table, all other
entries are zero, set by default. zero state is error state, negative
state represents incomplete token, positive state represents complete
acceptable token}

begin
    lexsttbl[-1, '!'] := 1;                { f(state,chr)=state }
    lexsttbl[-1, 'A'] := -2;
    lexsttbl[-1, 'B'] := -2;
    lexsttbl[-1, 'C'] := -2;
    lexsttbl[-1, 'D'] := -2;
    lexsttbl[-1, 'E'] := -2;
    lexsttbl[-1, 'F'] := -2;
    lexsttbl[-1, 'G'] := -2;
    lexsttbl[-1, 'H'] := -2;
    lexsttbl[-1, 'I'] := -2;
    lexsttbl[-1, 'J'] := -2;
    lexsttbl[-1, 'K'] := -2;
    lexsttbl[-1, 'L'] := -2;
    lexsttbl[-1, 'M'] := -2;
    lexsttbl[-1, 'N'] := -2;
    lexsttbl[-1, 'O'] := -2;
    lexsttbl[-1, 'P'] := -2;
    lexsttbl[-1, 'Q'] := -2;
    lexsttbl[-1, 'R'] := -2;
    lexsttbl[-1, 'S'] := -2;
    lexsttbl[-1, 'T'] := -2;
    lexsttbl[-1, 'U'] := -2;
    lexsttbl[-1, 'V'] := -2;
    lexsttbl[-1, 'W'] := -2;
    lexsttbl[-1, 'X'] := -2;
    lexsttbl[-1, 'Y'] := -2;
    lexsttbl[-1, 'Z'] := -2;
    lexsttbl[-1, '%'] := 2;
    lexsttbl[-1, '&'] := 3;
    lexsttbl[-1, '('] := 4;
    lexsttbl[-1, ')'] := 5;
    lexsttbl[-1, '*'] := 6;
    lexsttbl[-1, '+'] := 7;
    lexsttbl[-1, '-'] := 8;
    lexsttbl[-1, '/'] := 9;
    lexsttbl[-1, '0'] := 10;
    lexsttbl[-1, '1'] := 10;
    lexsttbl[-1, '2'] := 10;
    lexsttbl[-1, '3'] := 10;
    lexsttbl[-1, '4'] := 10;
    lexsttbl[-1, '5'] := 10;
    lexsttbl[-1, '6'] := 10;
    lexsttbl[-1, '7'] := 10;
    lexsttbl[-1, '8'] := 10;
    lexsttbl[-1, '9'] := 10;
    lexsttbl[-1, '<'] := 11;
    lexsttbl[-1, '='] := 12;
    lexsttbl[-1, '>'] := 13;
    lexsttbl[-1, '?'] := 14;
```

```
    lexsttbl[-1, '@'] := 15;
    lexsttbl[9, '\'] := 16;
    lexsttbl[-1, '\'] := -7;
    lexsttbl[-7, '/'] := 17;
    lexsttbl[-1, '^'] := 18;
    lexsttbl[-1, 'a'] := 19;
    lexsttbl[-1, 'b'] := 19;
    lexsttbl[-1, 'c'] := 19;
    lexsttbl[-1, 'd'] := 19;
    lexsttbl[-1, 'e'] := 29;
    lexsttbl[-1, 'f'] := 19;
    lexsttbl[-1, 'g'] := 19;
    lexsttbl[-1, 'h'] := 19;
    lexsttbl[-1, 'i'] := 27;
    lexsttbl[-1, 'j'] := 19;
    lexsttbl[-1, 'k'] := 19;
    lexsttbl[-1, 'l'] := 19;
    lexsttbl[-1, 'm'] := 19;
    lexsttbl[-1, 'n'] := 19;
    lexsttbl[-1, 'o'] := 19;
    lexsttbl[-1, 'p'] := 19;
    lexsttbl[-1, 'q'] := 19;
    lexsttbl[-1, 'r'] := 19;
    lexsttbl[-1, 's'] := 19;
    lexsttbl[-1, 't'] := 28;
    lexsttbl[-1, 'u'] := 19;
    lexsttbl[-1, 'v'] := 19;
    lexsttbl[-1, 'w'] := 19;
    lexsttbl[-1, 'x'] := 19;
    lexsttbl[-1, 'y'] := 19;
    lexsttbl[-1, 'z'] := 19;
    lexsttbl[-1, '{'] := 20;
    lexsttbl[-1, '|'] := 21;
    lexsttbl[-1, '}'] := 22;
    lexsttbl[-2, '('] := 23;
    lexsttbl[11, '='] := 24;
    lexsttbl[12, '='] := 25;
    lexsttbl[13, '='] := 26;
    lexsttbl[27, 'f'] := 30;
    lexsttbl[28, 'h'] := -3;
    lexsttbl[-3, 'e'] := -4;
    lexsttbl[-4, 'n'] := 31;
    lexsttbl[29, 'l'] := -5;
    lexsttbl[-5, 's'] := -6;
    lexsttbl[-6, 'e'] := 32;
    lexsttbl[10, '0'] := 10;
    lexsttbl[10, '1'] := 10;
    lexsttbl[10, '2'] := 10;
    lexsttbl[10, '3'] := 10;
    lexsttbl[10, '4'] := 10;
    lexsttbl[10, '5'] := 10;
    lexsttbl[10, '6'] := 10;
    lexsttbl[10, '7'] := 10;
    lexsttbl[10, '8'] := 10;
    lexsttbl[10, '9'] := 10
end; { MakeLexStateTbl }

procedure MakeLexSymTbl;
{set up acceptcode to token name converter table}

begin
    lexsymtbl[0] := ERR;                    {accept state 0 is err token}
    lexsymtbl[1] := NOT;
    lexsymtbl[2] := MOD;
    lexsymtbl[3] := AND;
    lexsymtbl[4] := LPAREN;
```

```
    lexsymtbl[5] := RPAREN;
    lexsymtbl[6] := MULT;
    lexsymtbl[7] := PLUS;
    lexsymtbl[8] := MINUS;
    lexsymtbl[9] := DIV;
    lexsymtbl[10] := NUM;
    lexsymtbl[11] := LT;
    lexsymtbl[12] := ASSIG;
    lexsymtbl[13] := GT;
    lexsymtbl[14] := QM;
    lexsymtbl[15] := UNDEF;
    lexsymtbl[16] := MAX;
    lexsymtbl[17] := MIN;
    lexsymtbl[18] := TTPO;
    lexsymtbl[19] := VAR;
    lexsymtbl[27] := VAR;
    lexsymtbl[28] := VAR;
    lexsymtbl[29] := VAR;
    lexsymtbl[20] := LBR;
    lexsymtbl[21] := OR;
    lexsymtbl[22] := RBR;
    lexsymtbl[23] := VEC;
    lexsymtbl[24] := LE;
    lexsymtbl[25] := EQ;
    lexsymtbl[26] := GE;
    lexsymtbl[30] := IF;
    lexsymtbl[31] := THEN;
    lexsymtbl[32] := ELSE;
    lexsymtbl[33] := EOC;
    lexsymtbl[34] := UNMINUS
end; { MakeLexSymTbl }

procedure GetIp;
{procedure to transfer the input to a character store, performing some
preliminary filtering of meta-characters}

var
    cc: char;                                   {current char read}
    passcc: boolean;                            {true if cc is to be stored}
    i: integer;                                 {initn loop var}

begin
    for i := 1 to maxchrln do                   {clear input store}
        ipstore[i] := ' ';
    ipstoretop := 0;                            {init end-of-store mkr}
    cc := ' ';                                  {init cc}
    while (cc <> ';') and not eoln do begin     {on current command}
        passcc := true;                         {default setting}
        read(cc);                               {get cc}
        if cc = ':' then begin                  {if cc=line-connector then-}
            readln;                             {facilitate subsequent-}
            passcc := false                     {reading from next line}
        end;
        if cc = ';' then                        {facilitate storage}
            passcc := false;                    {suppression of command-}
        if cc = ' ' then                        {separators, spaces and-}
            passcc := false;                    {tabs}
        if cc = '    ' then
            passcc := false;
        if passcc = true then begin             {if cc is to be stored}
            ipstoretop := ipstoretop + 1;       {move mkr}
            ipstore[ipstoretop] := cc           {store cc}
        end
    end;
    if cc = ';' then                            {if cc=command separator}
        scomm := false                          {facilitate prompt suppress}
```

```
    else
        scomm := true;                          {facilitate prompt printing}
    if eoln then                                {if at end-of-line then read-}
        readln                                  {subsequent ip from next ln}
end; { GetIp }


procedure SaveToken(code: accst);
{puts lexical and semantic value of current token into store. code is
the last +ve state the lexical analyser was in}

var
    i: integer;                                 {loop var}
begin
    tokenstoretop := tokenstoretop + 1;         {move token store mkr}
    tokenstore[tokenstoretop].sem := 0;         {store default sem val}
    tokenstore[tokenstoretop].lex := lexsymtbl[code];
                                                {store lex val}
    if code = 10 then begin                     {if token is num-}
        i := 0;                                 {calculate numeric val-}
        repeat                                  {from ascii codes of digit-}
            i := i + 1;                         {chars}
            tokenstore[tokenstoretop].sem :=
            tokenstore[tokenstoretop].sem * 10 + ord(tokenb[i]) - ord('0')
        until tokenb[i + 1] = ' '
    end;
    if (code = 19) or (code = 27) or (code = 28) or (code = 29) then
                                                {if token is var}
        tokenstore[tokenstoretop].sem := ord(tokenb[1]) - ord('a');
                                                {store pos of var in alphabet}
    if code = 23 then                           {if token is vec}
        tokenstore[tokenstoretop].sem := ord(tokenb[1]) - ord('A')
                                                {store pos of vec in alphabet}
end; { SaveToken }

procedure LexAnalyse;
{convert string of characters in ipstore to tokens, calculating their
semantic values for particular tokens}

var
    temp: lextype;                              {temp var}
    ipstoreptr: integer;                        {chars from ipstore mkr}
    j: integer;                                 {init loop var}
begin
    tokenstoretop := 0;                         {reset token store ptr}
    ipstoreptr := 0;                            {init ipstore mkr}
    repeat
        for j := 1 to maxchrln do begin         {clear char buffers}
            tokenb[j] := ' ';
            charb[j] := ' '
        end;
        cpos := 0;                              {init end-of-buffer mkrs}
        tpos := 0;
        cs := -1;                               {set initial state}
        tokencode := 0;                         {set initial tokencode}
        while (ipstoreptr < ipstoretop) and (cs <> 0) do begin
                                                {while not eof and not error}
            ipstoreptr := ipstoreptr + 1;       {get next char from ipstore}
            x := ipstore[ipstoreptr];
            cpos := cpos + 1;                   {store x in char buffer}
            charb[cpos] := x;
            cs := lexsttbl[cs, x];              {make state transition}
            if cs > 0 then begin                {if accept state}
                tokencode := cs;                {record state}
                for j := 1 to cpos do           {transfer contents of charb}
                    tokenb[j + tpos] := charb[j];
```

```
                                                {to tokenb}
                tpos := tpos + cpos;
                cpos := 0
            end
        end;
        if tpos = 0 then begin                  {if error state}
            ipstoreptr := ipstoreptr - cpos;
                  {transfer all token excess chars from charb to ipstore}
            tokencode := 0;                     {record error code}
            ipstoreptr := ipstoreptr + 1
        end else
            ipstoreptr := ipstoreptr - cpos;
                  {transfer all token excess chars from charb to ipstore}
        if (tokencode = 8) and (tokenstoretop > 0) then begin
                        {if token is minus and not first token in store}
                              {if previous token is var or undef or num}
            temp := tokenstore[tokenstoretop].lex;
            if (temp = VAR) or (temp = UNDEF) or (temp = NUM) or
            (temp = RPAREN) or (temp = RBR) then
                SaveToken(8)                    {store minus}
            else
                SaveToken(34)                   {store unary minus}
        end else
            SaveToken(tokencode)                {store token}
    until ipstoreptr >= ipstoretop;             {until all chars used}
    SaveToken(33)                               {store end-of-input token}
end; { LexAnalyse }

procedure Parse;
{constructs a parse tree for the expression formed from the tokens in
tokenstore and takes action according to instructions for assignment or
query}

label
    1;                                          {goto for bad parse}
type
    errortype = (BADSYM,                        {token err encountered}
    BADSKIP,                                    {expected symbol not present}
    BADSTACK,
                                {bad stack states for complete expression}
    UNEXPECTED,
                              {token encountered in inappropriate context}
    BADSTART,                                   {invalid first token}
    DIVZERO,                                    {attempted zero division}
    CIRDEF,                                     {var involved in circ def}
    BADEND,                                     {stacks not reducible}
    INVREF,                                     {invalid vec ref}
    UDVECREF);                                  {vector ref undef}
var
    vecrefptr, queryptr: dataptr;               {general ptrs}
    varcode,                                    {var for circ defn check}
    vecvarassig,                                {vec ref}
    varassig: integer;                          {vec var code}
    tokenstoreptr: integer;                     {token store mkr}
    symlex: lextype;                            {lex val of token}
    symsem: integer;                            {sem val of token}

    procedure Error(errorcode: errortype);
    {produces appropriate error messages and prematurely ends parse}

    begin
        case errorcode of
            BADEND:
                writeln('incomplete expression');
            DIVZERO:
                writeln('zero division');
```

```
            CIRDEF:
                writeln('circular definition');
            BADSYM:
                writeln('unknown symbol');
            BADSKIP:
                writeln('missing symbol');
            BADSTACK:
                writeln('invalid expression');
            UNEXPECTED:
                writeln('unexpected symbol');
            BADSTART:
                writeln('invalid statement');
            UDVECREF:
                writeln('undefined array index');
            INVREF:
                writeln('invalid array index')
        end;
        goto 1                              {resume program after parse}
    end; { Error }

    procedure Traverse(T: dataptr);
    {traverses a tree given the root T, printing the expression
    represented in infix}

    begin
        if T = nil then                     {if empty tree}
            null                            {no action (return)}
        else begin
            if ((T^.lex in opset) or (T^.lex = IF)) and (T^.sem = 1) then
                                            {if brackets flag set}
                write('( ');                {print left bracket}
            if T^.lex = IF then begin       {if root is if}
                write('if ');               {print if}
                Traverse(T^.lch);           {traverse condition branch}
                write('then ');             {print then}
                Traverse(T^.rch^.lch);      {traverse then clause}
                write('else ');             {print else}
                Traverse(T^.rch^.rch)       {traverse else clause}
            end;
            if T^.lex in binopset then begin{if root is binary op}
                Traverse(T^.lch);           {traverse left branch}
                case T^.lex of              {print appropriate symbol}
                    MAX:
                        write('/\ ');
                    MIN:
                        write('\/ ');
                    TTPO:
                        write('^ ');
                    MOD:
                        write('% ');
                    MULT:
                        write('* ');
                    DIV:
                        write('/ ');
                    PLUS:
                        write('+ ');
                    LT:
                        write('< ');
                    LE:
                        write('<= ');
                    GT:
                        write('> ');
                    EQ:
                        write('== ');
                    GE:
                        write('>= ');
```

```
                AND:
                    write('& ');
                OR:
                    write('| ');
                MINUS:
                    write('- ')
            end;
            Traverse(T^.rch)            {traverse right branch}
        end;
        if T^.lex in unopset then begin {if root is unary op}
            case T^.lex of              {print appropriate symbol}
                UNMINUS:
                    write('-');
                NOT:
                    write('!')
            end;                        {traverse left branch only}
            Traverse(T^.lch)
        end;
        if T^.lex = NUM then            {if root is num}
            write(T^.sem:0,' ');        {print sem val}
        if T^.lex = VAR then            {if root is var}
            write(chr(ord('a') + T^.sem),' ');
                                        {print letter}
        if T^.lex = UNDEF then          {if root is undef}
            write('@ ');                {print undef symbol}
        if T^.lex = VEC then begin      {if root is vec}
            write(chr(ord('A') + T^.sem), '( ');
                                        {print ( and letter}
            Traverse(T^.lch);           {traverse ref branch}
            write(') ')                 {print closing bracket}
        end;
        if ((T^.lex in opset) or (T^.lex = IF)) and (T^.sem = 1) then
                                        {if brackets flag set}
            write(') ')                 {print right bracket}
    end
end; { Traverse }

function SDV(T: dataptr): boolean;
{traverse a tree given root T to see if any variables encountered
have the same variable code as the 1 whose variable code is set to
varcode}

var
    redexp: boolean;                    {true for circ def}

begin
    redexp := false;                    {default setting}
    if T^.lex in binopset then          {if root is binary op}
        if SDV(T^.lch) or SDV(T^.rch) then
                                {if left or right branch has circ def}
            redexp := true;             {this branch has circ defn}
    if T^.lex in unopset then           {if root is unary op}
        if SDV(T^.lch) then             {if left branch has circ def}
            redexp := true;             {this branch has circ defn}
    if (T^.lex = UNDEF) or (T^.lex = NUM) then
                                        {if root is undef or num}
        redexp := false;                {this branch hasn't circ def}
    if T^.lex = VAR then                {if root is var}
        if varcode = -(T^.sem + 1) then {if var code = varcode}
            redexp := true              {this branch has circ defn}
        else
            redexp := SDV(vartbl[T^.sem]);
                                        {check this var's defn}
    if T^.lex = VEC then                {if root is vec}
        if varcode = T^.sem * (maxref + 1) + T^.lch^.sem then
                                        {if var code = varcode}
```

```
            redexp := true                  {this branch has circ defn}
        else
            redexp := SDV(vectbl[T^.sem, T^.lch^.sem]);
                                            {check this vec's defn}
    if T^.lex = IF then                     {if root is if}
        redexp := SDV(T^.lch) or SDV(T^.rch^.lch) or SDV(T^.rch^.rch);
                                            {check all sub-branches}
    SDV := redexp                           {return result}
end; { SDV }

function Eval(T: dataptr): dataptr;
{by recursive evaluation of subtrees this procedure reduces a tree
whose root is T to a single cell containing a number or an undefined
value}

var
    condptr, thenptr, elseptr, redtreeptr, redlchptr,
    redrchptr: dataptr;                     {general ptrs}

begin
    new(redtreeptr);                        {make new cell}
    if T^.lex = VAR then                    {if root is var}
        redtreeptr := Eval(vartbl[T^.sem]);
                                            {evaluate this var's defn}
    if T^.lex = UNDEF then                  {if root is undef}
        redtreeptr := T;                    {T reduces to this cell}
    if T^.lex = NUM then                    {if root is num}
        redtreeptr := T;                    {T reduces to this cell}
    if T^.lex = VEC then                    {if root is vec}
        redtreeptr := Eval(vectbl[T^.sem, T^.lch^.sem]);
                                            {evaluate this vec's defn}
    if T^.lex in unopset then begin         {if root is unary op}
        redlchptr := Eval(T^.lch);          {evaluate left branch}
        if redlchptr^.lex = UNDEF then begin
                                            {if left branch is undef}
            redtreeptr^.lex := UNDEF;       {T is undef}
            redtreeptr^.sem := 0;
            redtreeptr^.lch := nil;
            redtreeptr^.rch := nil
        end else begin
            case T^.lex of                  {if root is}
                UNMINUS:
                                    {perform appropriate op on sem val}
                    redtreeptr^.sem := -redlchptr^.sem;
                NOT:
                    if redlchptr^.sem = 0 then
                                            {if subtree is 0}
                        redtreeptr^.sem := 1
                                            {T points to 1}
                    else
                        redtreeptr^.sem := 0
                                            {T points to 0}
            end;
            redtreeptr^.lex := NUM;         {set remaining cell features}
            redtreeptr^.lch := nil;
            redtreeptr^.rch := nil
        end
    end;
    if T^.lex in binopset then begin        {if root is binary op}
        redlchptr := Eval(T^.lch);          {evaluate left branch}
        redrchptr := Eval(T^.rch);          {evaluate right branch}
        if (redlchptr^.lex = UNDEF) or (redrchptr^.lex = UNDEF) then
            begin
                        {if either branches reduce to undefined values}
            redtreeptr^.lex := UNDEF;       {T is undef}
            redtreeptr^.sem := 0;           {set default sem val}
```

```
                redtreeptr^.lch := nil;      {no left branch}
                redtreeptr^.rch := nil       {no right branch}
            end else begin
                case T^.lex of               {if root is}
                    MAX:
                           {set T to point to max val of reduced subtrees}
                        if redlchptr^.sem > redrchptr^.sem then
                            redtreeptr^.sem := redlchptr^.sem
                        else
                            redtreeptr^.sem := redrchptr^.sem;
                    MIN:
                           {set T to point to min val of reduced subtrees}
                        if redlchptr^.sem < redrchptr^.sem then
                            redtreeptr^.sem := redlchptr^.sem
                        else
                            redtreeptr^.sem := redrchptr^.sem;
                    TTPO:
                {set T to point to left val to the power of the right val}
                            redtreeptr^.sem := trunc(exp(redrchptr^.sem *
                            ln(redlchptr^.sem)));
                    MOD:
{checking for zero modulus, set T to point to left val modulo right val}
                            if redrchptr^.sem = 0 then
                                Error(DIVZERO)
                            else
                                redtreeptr^.sem := redlchptr^.sem mod
                                redrchptr^.sem;
                    MULT:
                        {set T to point to product of left and right vals}
                        redtreeptr^.sem := redlchptr^.sem *
                        redrchptr^.sem;
                    DIV:
                                {checking for zero division, set T to point to
                                            quotient of left and right vals}
                        if redrchptr^.sem = 0 then
                            Error(DIVZERO)
                        else
                            redtreeptr^.sem := trunc(redlchptr^.sem /
                            redrchptr^.sem);
                    PLUS:
                                {set T to point to sum of left and right vals}
                        redtreeptr^.sem := redlchptr^.sem +
                        redrchptr^.sem;
                    MINUS:
                         {set T to point to difference of left and right vals}
                        redtreeptr^.sem := redlchptr^.sem -
                        redrchptr^.sem;
                    LT:
                          {set T to point to 1 if left val < right val else 0}
                        if redlchptr^.sem < redrchptr^.sem then
                            redtreeptr^.sem := 1
                        else
                            redtreeptr^.sem := 0;
                    LE:
                         {set T to point to 1 if left val <= right val else 0}
                        if redlchptr^.sem <= redrchptr^.sem then
                            redtreeptr^.sem := 1
                        else
                            redtreeptr^.sem := 0;
                    EQ:
                          {set T to point to 1 if left val = right val else 0}
                        if redlchptr^.sem = redrchptr^.sem then
                            redtreeptr^.sem := 1
                        else
                            redtreeptr^.sem := 0;
                    GE:
```

```
                {set T to point to 1 if left val >= right val else 0}
                  if redlchptr^.sem >= redrchptr^.sem then
                      redtreeptr^.sem := 1
                  else
                      redtreeptr^.sem := 0;
              GT:
                {set T to point to 1 if left val > right val else 0}
                  if redlchptr^.sem > redrchptr^.sem then
                      redtreeptr^.sem := 1
                  else
                      redtreeptr^.sem := 0;
              AND:
   {set T to point to 1 if left val and right val non-zero else 0}
                  if (redlchptr^.sem <> 0) and
                  (redrchptr^.sem <> 0) then
                      redtreeptr^.sem := 1
                  else
                      redtreeptr^.sem := 0;
              OR:
    {set T to point to 1 if left val or right val non-zero else 0}
                  if (redlchptr^.sem <> 0) or
                  (redrchptr^.sem <> 0) then
                      redtreeptr^.sem := 1
                  else
                      redtreeptr^.sem := 0
          end;
          redtreeptr^.lex := NUM;      {set remaining cell features}
          redtreeptr^.lch := nil;      {no left branch}
          redtreeptr^.rch := nil       {no right branch}
      end
end;
if T^.lex = IF then begin              {if root is if}
    condptr := Eval(T^.lch);           {evaluate condition branch}
    if condptr^.lex = UNDEF then begin
                                       {if condition branch undef}
        redtreeptr^.lex := UNDEF;      {T is undef}
        redtreeptr^.sem := 0;          {set default sem val}
        redtreeptr^.lch := nil;        {no left branch}
        redtreeptr^.rch := nil         {no right branch}
    end else if condptr^.sem <> 0 then begin
                                       {if condition branch non-zero}
        thenptr := Eval(T^.rch^.lch);{evaluate then clause}
        if thenptr^.lex = UNDEF then begin{if then clause undef}
            redtreeptr^.lex := UNDEF;
                                       {T is undef}
            redtreeptr^.sem := 0;      {set default sem val}
            redtreeptr^.lch := nil;    {no left branch}
            redtreeptr^.rch := nil     {no right branch}
        end else begin
            redtreeptr^.lex := NUM;
                          {set T to point to reduced then clause}
            redtreeptr^.sem := thenptr^.sem;
                                       {set sem val}
            redtreeptr^.lch := nil;    {no left branch}
            redtreeptr^.rch := nil     {no right branch}
        end
    end else begin
        elseptr := Eval(T^.rch^.rch);
                                       {evaluate else clause}
        if elseptr^.lex = UNDEF then begin
                                       {if else clause undef}
            redtreeptr^.lex := UNDEF;
                                       {T id undef}
            redtreeptr^.sem := 0;      {set default sem val}
            redtreeptr^.lch := nil;    {no left branch}
            redtreeptr^.rch := nil     {no right branch}
```

```
            end else begin
                redtreeptr^.lex := NUM;
                                {set T to point to reduced else clause}
                redtreeptr^.sem := elseptr^.sem;
                                        {set sem val}
                redtreeptr^.lch := nil; {no left branch}
                redtreeptr^.rch := nil  {no right branch}
            end
        end
    end;
    Eval := redtreeptr                  {return val}
end; {Eval}

procedure GetToken;
{sets symlex and symsem to lexical and semantic values of next token
in token store}

begin
    tokenstoreptr := tokenstoreptr + 1; {advance store ptr}
    symlex := tokenstore[tokenstoreptr].lex;
                                        {get lex val}
    symsem := tokenstore[tokenstoreptr].sem;
                                        {get sem val}
    if symlex = ERR then                {if err token encountered}
        Error(BADSYM)                   {error}
end; { GetToken }

procedure BackSym;
{moves store pointer back one symbol. this is necessary if there is a
switch in parsing techniques and the subsequent parsing procedure
has to read the symbol again}

begin
    tokenstoreptr := tokenstoreptr - 1  {move store ptr back 1}
end; { BackSym }

procedure PassSym(passme: lextype);
{skips over the next token if it is the same as the token passme}

begin
    GetToken;                           {get next token}
    if symlex <> passme then            {if next token not passme}
        Error(BADSKIP)                  {error}
end; { PassSym }

function BottomUp: dataptr;
forward;
{forward reference for bottom up parsing function}

function TopDown: dataptr;
{returns the root of a forthcoming expression parsed in a top-down
manner}

var
    cellptr, temp: dataptr;             {general ptrs}

begin
    GetToken;                           {get next token}
    case symlex of                      {if token is}
        VEC:
            begin
                new(cellptr);           {make new cell}
                cellptr^.sem := symsem; {set cell fields}
                cellptr^.lex := VEC;
                cellptr^.lch := Eval(BottomUp);
                                {left branch set to subsequent exp}
```

```
                if cellptr^.lch^.lex = UNDEF then{if ref undef}
                    Error(UDVECREF);    {error}
                if (cellptr^.lch^.sem>maxref) or
                (cellptr^.lch^.sem<0) then
                    Error(INVREF);
                cellptr^.rch := nil;    {no right branch}
                PassSym(RPAREN);        {skip over right bracket}
                TopDown := cellptr      {return root of new cell}
            end;
        LPAREN:
            begin
                temp := BottomUp;       {root set to subsequent exp}
                if (temp^.lex in opset) or (temp^.lex = IF) then
                    temp^.sem := 1;     {set brackets flag}
                TopDown := temp;        {return root}
                PassSym(RPAREN)         {skip over right bracket}
            end;
        LBR:
            begin
                TopDown := Eval(BottomUp);
                                {root set to evaluated subsequent exp}
                PassSym(RBR)            {skip over right brace}
            end;
        IF:
            begin
                new(cellptr);           {make new cell}
                cellptr^.lex := IF;     {set lex field}
                cellptr^.lch := BottomUp;
                                        {get condition exp}
                cellptr^.sem := 0;      {set sem field}
                PassSym(THEN);          {skip over then}
                new(cellptr^.rch);      {make actions cell}
                cellptr^.rch^.lex := IFACTIONS;
                                        {set lex field}
                cellptr^.rch^.sem := 0; {set sem field}
                cellptr^.rch^.lch := BottomUp;
                                        {get then exp}
                PassSym(ELSE);          {skip over else}
                cellptr^.rch^.rch := BottomUp;
                                        {get else exp}
                TopDown := cellptr      {return val}
            end;
    end
end; { TopDown }

function BottomUp;
{returns the root of the following expression parsed in a bottom-up
manner}

var
    exitst: boolean;
                            {true if next token suggests end of exp}
    opstack: array [1..maxstkht] of lextype;
                                        {op stack}
    exstack: array [1..maxstkht] of dataptr;
                                        {op stack}
    opstacktop,                         {op stack top mkr}
    exstacktop: integer;                {exp stack top mkr}
    cellptr: dataptr;                   {general ptr}

        procedure Reduce;
        {combines expressions on expression stack with operator on
        operator stack and places the result on the expression stack}

        var
            redexp,                     {ptr to combined exp}
```

```
            expl,                          {ptr to first exp off stack}
            exp2: dataptr;                 {ptr to second exp off stack}
            op: lextype;                   {first op off stack}
        begin
            op := opstack[opstacktop];  {pop op stack}
            opstacktop := opstacktop - 1;
            new(redexp);                   {make new cell}
            if op in unopset then begin {if op is unary op}
                if exstacktop<1 then
                            {if insufficient num of exps on exp stack}
                    Error(BADEND);
                expl := exstack[exstacktop]
                                           ;{pop exp stack}
                exstacktop := exstacktop - 1;
                redexp^.lch := expl;
                             {set left branch to point to popped exp}
                redexp^.rch := nil;        {no right branch}
                redexp^.lex := op;         {set lex field}
                redexp^.sem := 0           {no sem val}
            end else begin
                if exstacktop<2 then
                            {if insufficient num of exps on exp stack}
                    Error(BADEND);
                expl := exstack[exstacktop];
                                           {pop exp stack}
                exstacktop := exstacktop - 1;
                exp2 := exstack[exstacktop];
                                           {pop exp stack}
                exstacktop := exstacktop - 1;
                redexp^.lch := exp2;
                              {set left branch to second popped exp}
                redexp^.rch := expl;
                              {set right branch to first popped exp}
                redexp^.lex := op;         {set lex field}
                redexp^.sem := 0           {no sem val}
            end;
            exstacktop := exstacktop + 1;
                                           {push new exp}
            exstack[exstacktop] := redexp
        end; { Reduce }

begin
    exstacktop := 0;                       {clear exp stack}
    opstack[1] := OPBASE;
                                  {push irreducible token on op stack}
    opstacktop := 1;
    exitst := false;
                                       {default procedure exit status}
    repeat
        GetToken;                          {get next token}
        case symlex of                     {if next token is}
            NOT, TTPO, MOD, MULT, DIV, PLUS, LT,
            LE, EQ, GT, MAX, MIN, GE, AND,
            OR, UNMINUS, MINUS:            {op list}
                begin
                    while opranktbl[symlex] >=
                    opranktbl[opstack[opstacktop]] do
                             {while of higher prec than op on opstack}
                        Reduce;            {reduce stack}
                    opstacktop := opstacktop + 1;
                                           {push next token}
                    opstack[opstacktop] := symlex
                end;
            VAR, NUM, UNDEF:
                begin
                    new(cellptr);          {make new cell}
```

```
                    cellptr^.lex := symlex;
                                        {insert token lex val}
                    cellptr^.sem := symsem;
                                        {insert token sem val}
                    cellptr^.lch := nil;{no left branch}
                    cellptr^.rch := nil;{no right branch}
                    exstacktop := exstacktop + 1;
                    exstack[exstacktop] := cellptr
                                        {push this cell}
                end;
            VEC, LPAREN, LBR, IF:       {top down left delimiters}
                begin
                    BackSym;            {move back 1 token}
                    exstacktop := exstacktop + 1;
                    exstack[exstacktop] := TopDown
                                        {push subsequent top-down exp}
                end;
            RPAREN, RBR, THEN, EOC, ELSE:{bottom up right delimiters}
                begin
                    BackSym;            {move back 1 token}
                    symlex:=BUENDSYM;
                                  {facilitate total stack reduction}
                    while opranktbl[symlex] >=
                    opranktbl[opstack[opstacktop]] do
                        Reduce;         {do total stack reduction}
                    exitst := true      {facilitate proc finish}
                end;
            ASSIG, QM:                  {inappropriate symbols}
                Error(UNEXPECTED)       {error}
        end
    until exitst = true;                {until proc finish facilitate}
    if (exstacktop = 1) and (opstacktop = 1) then
                                        {if stack in good state}
        BottomUp := exstack[exstacktop]
                                  {return ptr to final expression}
    else Error(BADSTACK);               {error}
end; { BottomUp }

begin
    tokenstoreptr := 0;                 {init token store ptr}
    GetToken;                           {get first token}
    case symlex of                      {if first token}
        VAR:
            begin
                varassig := symsem;     {record var to be assigned to}
                PassSym(ASSIG);         {skip over assignment sign}
                tempptr := BottomUp;    {get exp assigned}
                varcode := -(varassig + 1); {compute var code}
                if SDV(tempptr) then    {if circ def}
                    Error(CIRDEF)       {error}
                else
                    vartbl[varassig] := tempptr
                                        {make assignment}
            end;
        QM:
            begin
                queryptr := BottomUp;   {get exp queried}
                if queryptr^.lex = VAR then {if exp is single var}
                    queryptr := vartbl[queryptr^.sem];
                                  {reset expt that assigned to the var}
                if queryptr^.lex = VEC then {if exp is single vec}
                    queryptr := vectbl[queryptr^.sem, queryptr^.lch^.sem];
                                  {reset exp to that assigned to the vec}
                Traverse(queryptr);     {traverse the given exp}
                writeln
            end;
```

```
            VEC:
                begin
                    vecvarassig := symsem;      {record vec to be assigned to}
                    vecrefptr := Eval(BottomUp);{get ref}
                    if vecrefptr^.lex = UNDEF then
                                                {if not numeric ref}
                        Error(UDVECREF);        {error}
                    if (vecrefptr^.sem>maxref) or (vecrefptr^.sem<0) then
                        Error(INVREF);
                    PassSym(RPAREN);            {skip over right bracket}
                    PassSym(ASSIG);             {skip over assignment sign}
                    tempptr := BottomUp;        {get exp assigned}
                    varcode := vecvarassig * (maxref + 1) + vecrefptr^.sem;
                                                {compute var code}
                    if SDV(tempptr) then        {if circ def}
                        Error(CIRDEF)           {error}
                    else
                        vectbl[vecvarassig, vecrefptr^.sem] := tempptr
                                                {make assignment}
                end;
            LPAREN, LBR, IF, RPAREN, RBR, THEN, ELSE,
            EOC, ASSIG, MAX, MIN, TTPO, MOD, MULT,
            DIV, PLUS, LT, LE, EQ, GT, GE,
            AND, OR, MINUS, NUM, NOT, UNDEF:    {inappropriate first tokens}
                Error(BADSTART)                 {error}
        end;
        PassSym(EOC);                           {skip over end-of-input}
    1:
        null                                    {landing place for bad parse}
    end; { Parse }

    procedure Init;
    {does all initialisation processes}

    begin
        CheckArgs;                              {check for call arguments}
        if sprom = false then                   {if nor silent mode}
            write('UDC> ');                     {give prompt}
        VarTblInit;                             {set up var tables}
        MakeLexStateTbl;
                                    {set up lex analysis transition table}
        MakeLexSymTbl;                          {set up token code table}
        MakeOpRankTbl;                          {define op precedences}
        opset := [MAX..NOT];                    {define op set}
        binopset := [MAX..MINUS];               {define binary ops}
        unopset := [UNMINUS..NOT]               {define unary ops}
    end; { Init }

begin
    Init;                                       {do initn}
    while not eof do begin                      {while there is input}
        GetIp;                                  {get command line}
        if ipstoretop > 0 then begin            {if not blank}
            LexAnalyse;                         {perform lex analysis}
            Parse                               {perform parse}
        end;
        if (scomm = true) and (sprom = false) then
                                    {if single command and not silent mode}
            write('UDC> ')                      {give prompt}
    end;
    if sprom = false then                       {if not silent mode}
        writeln
                                {produce blank line to clear the escape symbol}
end.

################################################################################
```