Demonstration Programs for CTL and μ-Calculus Symbolic Model Checking

by

Martin Richards

mr@uk.ac.cam.cl

http://www.cl.cam.ac.uk/users/mr/

Computer Laboratory
University of Cambridge

September 15, 2005

Abstract

This paper presents very simple implementations of Symbolic Model Checkers for both Computational Tree Logic (CTL) and μ-calculus. They are intended to be educational rather than practical. The first program discovers, for a given non-deterministic finite state machine (NFSM), the states for which a given CTL formula holds. The second program does the same job for μ-calculus formulae.

For simplicity the number of states in the NFSM has been limited to 32 and a bit pattern representation is used to represent the boolean functions involved. It would be easy to extend both programs to use ordered binary decision diagrams more normally used in symbolic model checking.

The programs include lexical and syntax analysers for the formulae, the model checking algorithms and drivers to exercise them with respect to various simple machines. The programs is implemented in MCPL. A brief summary of MCPL is given at the end.

Keywords

Symbolic model checking, Computational Tree Logic, μ-calculus, finite state machines, boolean functions, bit patterns, MCPL.

Contents

1 Introduction

This report describes two programs to illustrate how symbolic model checkers works, one uses Computational Tree Logic (CTL) and the other uses μ-calculus. Symbolic model checking normally relies on the use of ordered binary decision diagrams (OBDDs) to allow substantial problems to be tested, but, for simplicity, these are not used here. Instead, the boolean functions are represented directly using bit patterns of length 32, and the transition relations for the non deterministic finite state machines (NFSMs) are encoded by a 32×32 bit matrices. In this implementation, the size of the NFSMs are thus limited to 32 states, but this is sufficient to illustrate the capabilities of these two logics.

The first program presented is essentially an implementation, in MCPL[Ric97], of the algorithm described in Symbolic Model Checking by McMillan[McM93], and the second is based on a paper by Berezin, Clarke, Jha and Marrero[SBM96]. The MCPL code should be comprehensible without previous knowledge of the language, but a brief summary of the language is given at the end.

2 CTL Model Checking

Given a non deterministic finite state machine(NFSM), we can identify its states using binary integers encoded by a sequence of Booleans $(v_1, v_2, \ldots v_n)$. We can imagine properties that are satisfied by some states and not others. Such properties can be defined by functions of type $\{0,1\}^n \rightarrow \{0,1\}$. These functions can be specified by propositional formulae involving the variables $(v_1, v_2, \ldots v_n)$ and the operators $\neg, \wedge, \vee, \Rightarrow$, and $\Leftrightarrow$. But more interesting properties depend also on the transitions of the NFSM; for example: is there a path from the given state to one in which $v_1 \wedge v_2$ is true? Many such properties can be described using CTL[CE81] described in the next section.

A Symbolic model checker is a program to determine for which states a given formula holds with respect to a given NFSM. Often we wish to check that the formula holds for all states. In general, the cost of the algorithm grows exponentially with n, but, by cunning encoding and the use of OBDDs, significant problems can often be solved in reasonable time for even quite large values of n.

2.1 Computational Tree Logic

A CTL formula defines a function of type $\{0,1\}^n \rightarrow \{0,1\}$, whose argument variables $(v_1, v_2, \ldots v_n)$ identify a state in the given NFSM, and whose result indicates whether the formula is satisfied at this state.

For this demonstration, only five variables (`a`, `b`, `c`, `d`, `e`) are allowed, limiting the number of NFSM states to 32. CTL formulae are formed as follows:

- `a`, `b`, `c`, `d`, `e`, `T`, and `F` are formulae
- assuming *f* and *g* are formulae then so are:

`(`*f*`)`,	`~`*f*,	*f*`=`*g*,
f`&`*g*,	*f*`|`*g*,	*f*`->`*g*,
`AX` *f*,	`AF` *f*,	`AG` *f*,
`EX` *f*,	`EF` *f*,	`EG` *f*,
`A(`*f* `U` *g*`)`,	`E(`*f* `U` *g*`)`	

2.2 Semantics of CTL

A CTL formula is evaluated with respect to a current state, represented by a 5-tuple of truth values (`a`, `b`, `c`, `d`, `e`), and a given NFSM. The meaning of a CTL formula depends on its syntactic form as follows:

`a` is true if and only if `a` is true in the 5-tuple representing the current state. The other simple variables forms are defined similarly.

`F` is false for all states, and `T` is true for all states.

`(` *f* `)` is true if and only if *f* is satisfied in the current state.

`~` *f* is true if and only if *f* is false in the current state.

f `=` *g* is true if and only if both operands have the same value in the current state.

f `&` *g* is true if and only if both operands are satisfied in the current state.

f `|` *g* is true if and only if one or both operands are satisfied in the current state.

f `->` *g* is equivalent to `~`*f*`|`*g*.

`AX` *f* is true if and only if *f* is true for every immediate successor state.

`AF` *f* is true if and only if *f* is true in the current state, or `AF` *f* is true for every immediate successor state of which there must be at least one.

`AG` *f* is true if and only if *f* is true in the current state and `AG` *f* is true for every immediate successor state.

`EX` *f* is true if and only if *f* is true for at least one immediate successor state.

`EF` *f* is true if and only if *f* is true in the current state, or `EF` *f* is true for at least one immediate successor state.

`EG` *f* is true if and only if *f* is true in the current state and `EG` *f* is true for at least one immediate successor state.

`A(`f `U` g`)` is true if and only if g is true in the current state, or f is true in the current state and `A(`f `U` g`)` is true for every immediate successor state of which there must be at least one.

`E(`f `U` g`)` is true if and only if g is true in the current state, or f is true in the current state and `E(`f `U` g`)` is true for at least one immediate successor state. Note that the semantics given above permit the NFSM to contain states that have no successors.

2.3 Syntax Analysis

The program given in Section 2.5 parses and evaluates various CTL formulae. The structure of the parse tree is as follows:

T	$\rightarrow$	`[Atom,`	*bits*`]`		—	`a`, `b`, `c`, `d`, `e`, `T`, `F`
		`[Not,`	T_f`]`		—	`~`f
		`[Eq,`	T_f,	T_g`]`	—	f`=`g
		`[And,`	T_f,	T_g`]`	—	f`&`g
		`[Or,`	T_f,	T_g`]`	—	f`|`g
		`[Imp,`	T_f,	T_g`]`	—	f`->`g
		`[EX,`	T_f`]`		—	`EX` f
		`[EF,`	T_f`]`		—	`EF` f
		`[EG,`	T_f`]`		—	`EG` f
		`[AX,`	T_f`]`		—	`AX` f
		`[AF,`	T_f`]`		—	`AF` f
		`[AG,`	T_f`]`		—	`AG` f
		`[EU,`	T_f,	T_g`]`	—	`E(`f `U` g`)`
		`[AU,`	T_f,	T_g`]`	—	`A(`f `U` g`)`

where

bits is a bit pattern representing a subset of S, and
T_f and T_g represent the parse trees for f and g.

Parsing is done by recursive descent using the functions:

- `exp` to parse expressions of a given precedence, and
- `prim` to parse primary expressions.

They both read lexical tokens using `lex`. The implementation of `lex` is particularly simple since tokens are longer than two characters. The next two characters are held in `ch` and `nch` and both are used in the MATCH statement that forms the body of `lex`. Note that a zero byte marks the end of an MCPL string.

The parse tree can be printed using `prtree`; for instance: the call `prtree(parse "AG (a&b->c) -> A(d U ~e)")` generates the following output:

```
Imp
*-AG
! *-Imp
!   *-And
!   ! *-a
!   ! *-b
!   *-c
*-AU
  *-d
  *-Not
    *-e
```

2.4 The CTL Model Checking Algorithm

A boolean function of five boolean variables (`a`, `b`, `c`, `d`, `e`) is represented by a bit pattern whose i^{th} bit holds the result where $i = \mathtt{a} + 2\mathtt{b} + 4\mathtt{c} + 8\mathtt{d} + 16\mathtt{e}$ (with true represented by 1 and false by 0). Thus, the bit pattern `#xFFFFFFFF` represents the function that always yields true, and `#x00000000` represents the function that always yields false. These are given manifest names `True` and `False`, respectively. The function: `f(a,b,c,d,e)=a` is represented by the pattern `#xAAAAAAAA` which is given the manifest name `Abits`. The name `Bbits`, `Cbits`, `Dbits` and `Ebits` are similarly defined. Notice that a function such as: `f(a,b,c,d,e)=a&b` is represented by `Abits&Bbits`.

The function `eval` computes the bit pattern representation of the boolean function corresponding to a given CTL formula. For the atomic formulae (`a` to `e`, `T` and `F`), the result is respectively `Abits` to `Ebits`, `True` and `False`. For the propositional operators, (`~`, `=`, `&`, `|` and `->`), the result is obtained by applying the operator to the operand value(s).

The value of `EX` f is obtained by applying `evalEX` to the bit pattern representing f, where `evalEX` is defined as follows:

```
FUN evalEX : w =>
  LET res = 0
  LET p   = preds
  WHILE w DO { IF w&1 DO res |:= !p
               w >>:= 1
               p+++
             }
  RETURN res
```

The NFSM is represented by the vector `preds` whose i^{th} element is the bit pattern giving the set of predecessors of state i. The argument `w` is the bit pattern representing f (i.e. the set of states for which f is satisfied), and the result is obtained by or-ing together the elements of `preds` corresponding to the states identified in `w`. So, if bit i of `w` is set, then element i of `preds` is or-ed into the result.

The expression `AX` f is also evaluated using `evalEX` using the observation that: `AX` f = `~EX ~`f. The value of `E(`f `U` g`)` is obtained by applying `evalEU` to the bit patterns for f and g. The definition of `evalEU` is as follows:

```
FUN evalEU : f, g =>  // Computes: E(f U g)
  LET y = g
  { LET a = g | f & evalEX y
    IF a=y RETURN y
    y := a
  } REPEAT
```

The correctness of the definition of `evalEU` depends on the observation that: `E(`f `U` g`)` = g `|` f `& EX E(`f `U` g`)`. The evaluation of formulae with leading operators `EF` and `AG` rely on the observations that: `EF` f = `E(T U` f`)` and `AG` f = `~E(T U ~`f`)`. The value of `A(`f `U` g`)` is obtained by applying `evalAU` to the bit patterns for f and g. The definition of `evalAU` is as follows:

```
FUN evalAU : f, g =>  // Computes: A(f U g)
  LET succs = evalEX True
  LET y = g
  { LET a = g | f & succs & ~evalEX(~y)
    IF a=y RETURN y
    y := a
  } REPEAT
```

The correctness of the definition of `evalAU` depends on the observation that: `A(`f `U` g`)` = g `|` f `& EX T & ~EX ~A(`f `U` g`)`. Note that the term `EX T` is true for any state that has one or more successors. `AF` f is computed using `evalAU` based on the fact that: `AF` f = `A(T U` f`)`, and finally, the value of `EG` f is obtained by applying `evalEG` to the bit pattern representing f, where `evalEG` is defined as follows:

```
FUN evalEG : f =>     // Computes: EG f
  LET nosuccs = ~evalEX True
  LET y = f
  { LET a = f & (nosuccs | evalEX y)
    IF a=y RETURN y
    y := a
  } REPEAT
```

The correctness of the definition of `evalEG` depends on the observation that: `EG` f = f `& (~EX T | EX EG` f`)`. Note that the term `~EX T` is satisfied for any state that has no successors.

It is easy to show, using monotonicity, that all the above computations terminate. The algorithm can be simplified slightly if every state is known to have at least one successor.

2.5 The CTL Model Checker Program

```
GET "mcpl.h"

MANIFEST
  Id, Atom, Not, And, Or, Imp, Eq,          // Tokens
  EX, EF, EG, EU, E, AX, AF, AG, AU, A, U,
  Lparen, Rparen, Eof,

  E_syntax=100, E_space, E_eval,            // Exceptions

                           // Atomic boolean functions
  True= #xFFFFFFFF,        // f(a,b,c,d,e) = T
  False=#x00000000,        // f(a,b,c,d,e) = F

  Abits=#xAAAAAAAA,        // f(a,b,c,d,e) = a
  Bbits=#xCCCCCCCC,        // f(a,b,c,d,e) = b
  Cbits=#xF0F0F0F0,        // f(a,b,c,d,e) = c
  Dbits=#xFF00FF00,        // f(a,b,c,d,e) = d
  Ebits=#xFFFF0000         // f(a,b,c,d,e) = e
```

```
//********** Model checking algorithm ************************

// The transition relation will be represented by the vector preds
// preds!i will be the bit pattern representing the set of immediate
//         predecessors of state i

STATIC preds = VEC #b11111  // Initialised later.

FUN eval
: [Atom, bits] =>   bits
: [Not, f]     => ~ eval f
: [And, f,  g] =>   eval f  &  eval g
: [Or,  f,  g] =>   eval f  |  eval g
: [Imp, f,  g] => ~ eval f  |  eval g
: [Eq,  f,  g] => ~(eval f XOR eval g)
: [EX,  f]     =>   evalEX(  eval f)
: [AX,  f]     => ~ evalEX(~ eval f)
: [EF,  f]     =>   evalEU(    True,   eval f)
: [AG,  f]     => ~ evalEU(    True, ~ eval f)
: [AF,  f]     =>   evalAU(    True,   eval f)
: [EU,  f,  g] =>   evalEU(  eval f,   eval g)
: [EG,  f]     =>   evalEG(  eval f)
: [AU,  f,  g] =>   evalAU(  eval f,   eval g)
:              =>   RAISE E_eval

FUN evalEX : w =>     // Computes: EX w
  LET res = 0
  LET p   = preds
  WHILE w DO { IF w&1 DO res |:= !p
               w >>:= 1
               p+++
             }
  RETURN res

FUN evalEU : f, g =>  // Computes: E(f U g)
  LET y = g
  { LET a = g | f & evalEX y
    IF a=y RETURN y
    y := a
  } REPEAT

FUN evalAU : f, g =>  // Computes: A(f U g)
  LET succs = evalEX True
  LET y = g
  { LET a = g | f & succs & ~ evalEX(~y)
    IF a=y RETURN y
    y := a
  } REPEAT

FUN evalEG : f =>     // Computes: EG f
  LET nosuccs = ~ evalEX True
  LET y = f
  { LET a = f & (nosuccs | evalEX y)
    IF a=y RETURN y
    y := a
  } REPEAT

//********** End of Model checking algorithm *****************
```

```
/****************** Syntax Analyser ************************

STATIC  str, strp, ch, nch, token, lexval

FUN rch : => ch, nch := nch, %strp
             IF nch DO strp++

FUN lex_init
: formula => str := formula; strp := formula; rch(); rch()

FUN lex : => MATCH (ch, nch)
: ' ' | '\n' => rch(); lex()     // Ignore white space
:  0         => token := Eof     // End of file
: 'a'        => token := Id;     lexval := Abits; rch()
: 'b'        => token := Id;     lexval := Bbits; rch()
: 'c'        => token := Id;     lexval := Cbits; rch()
: 'd'        => token := Id;     lexval := Dbits; rch()
: 'e'        => token := Id;     lexval := Ebits; rch()
: 'T'        => token := Id;     lexval := True;  rch()
: 'F'        => token := Id;     lexval := False; rch()
: '('        => token := Lparen;                  rch()
: ')'        => token := Rparen;                  rch()
: '~'        => token := Not;                     rch()
: '='        => token := Eq;                      rch()
: '&'        => token := And;                     rch()
: '|'        => token := Or;                      rch()
: '-', '>'   => token := Imp;             rch(); rch()
: 'A', 'X'   => token := AX;              rch(); rch()
: 'A', 'F'   => token := AF;              rch(); rch()
: 'A', 'G'   => token := AG;              rch(); rch()
: 'E', 'X'   => token := EX;              rch(); rch()
: 'E', 'F'   => token := EF;              rch(); rch()
: 'E', 'G'   => token := EG;              rch(); rch()
: 'A'        => token := A;                       rch()
: 'E'        => token := E;                       rch()
: 'U'        => token := U;                       rch()
:            => RAISE E_syntax
```

```
FUN parse : formula => lex_init formula;
                       LET tree = nexp 0
                       chkfor Eof
                       RETURN tree

FUN chkfor : tok => UNLESS token=tok RAISE E_syntax
                    lex()

FUN prim : => MATCH token
  : Id         => LET a = lexval; lex(); RETURN mk2(Atom, a)

  : Lparen     => LET a = nexp 0; chkfor Rparen; RETURN a

  : Not | AX | AF | AG | EX | EF | EG
               => LET op = token; RETURN mk2(op, nexp 5)

  : A | E      => LET op = token=A -> AU, EU
                  lex()
                  chkfor Lparen
                  LET a = exp 0
                  chkfor U
                  LET b = exp 0
                  chkfor Rparen
                  RETURN mk3(op, a, b)

  :            => RAISE E_syntax

FUN nexp : n => lex(); exp n

FUN  exp : n => LET a = prim()
                MATCH (token, n)
                : Eq,  <4 => a := mk3(Eq,  a, nexp 4)
                : And, <3 => a := mk3(And, a, nexp 3)
                : Or,  <2 => a := mk3(Or,  a, nexp 2)
                : Imp, <1 => a := mk3(Imp, a, nexp 1)
                :         => RETURN a
                . REPEAT
```

```
//******************** Space Allocation *****************

STATIC  spacev, spacep

FUN mk_init : upb    => spacev := getvec upb
                        UNLESS spacev RAISE E_space
                        spacep := @ spacev!upb

FUN mk_close :       => freevec spacev

FUN mk1 : x          => !---spacep := x; spacep
FUN mk2 : x, y       => mk1 y; mk1 x
FUN mk3 : x, y, z    => mk1 z; mk1 y; mk1 x

//************** Print tree function *********************

STATIC prlinev = VEC 50

FUN prtree
: 0,     ?,        ? => writef "Nil"
: ?, depth,   =depth => writef "Etc"
: x, depth, maxdepth =>
  LET upb = 1
  MATCH x
  : [Atom, =Abits]  => writef "a";           RETURN
  : [Atom, =Bbits]  => writef "b";           RETURN
  : [Atom, =Cbits]  => writef "c";           RETURN
  : [Atom, =Dbits]  => writef "d";           RETURN
  : [Atom, =Ebits]  => writef "e";           RETURN
  : [Atom, =True]   => writef "T";           RETURN
  : [Atom, =False]  => writef "F";           RETURN
  : [Not, f]        => writes "Not"
  : [Eq,  f,  g]    => writes "Eq";        upb := 2
  : [And, f,  g]    => writes "And";       upb := 2
  : [Or,  f,  g]    => writes "Or";        upb := 2
  : [Imp, f,  g]    => writes "Imp";       upb := 2
  : [EX,  f]        => writes "EX"
  : [EU,  f,  g]    => writes "EU";        upb := 2
  : [EG,  f]        => writes "EG"
  : [EF,  f]        => writes "EF"
  : [AX,  f]        => writes "AX"
  : [AG,  f]        => writes "AG"
  : [AU,  f,  g]    => writes "AU";        upb := 2
  :                 => writes "Unknown";   upb := 0
  .
  FOR i = 1 TO upb DO { newline()
                        FOR j=0 TO depth-1 DO writes( prlinev!j )
                        writes("*-")
                        prlinev!depth := i=upb-> "  ", "! "
                        prtree(x!i, depth+1, maxdepth)
                       }
```

```
//******************** Main Program *************************

FUN try : e =>
  { mk_init 100_000
    writef("\n%s\n", e)
    LET exp = parse e
//  prtree(exp, 0, 20)
    LET res = eval exp
    FOR v = #b00000 TO #b11111 DO
    { UNLESS v MOD 8 DO newline()
      writef("%5b %c ", v, res&1=0->' ', 'Y')
      res >>:= 1
    }
    newline()
  } HANDLE : E_syntax => writef("Bad Syntax\n%s\n", str)
                         FOR i = str TO strp-4 DO wrch ' '
                         writes "^\n"
           : E_space  => writef "Insufficient space\n"
           : E_eval   => writef "Error in eval\n"
           .
  mk_close()

FUN start : =>
  init_nfsm_5Dcube()

  try "d&e->a&b&c"
  try "EX a & EX b & EX c & EX d & EX e"
  try "EX EX (a&b&c&d&e)"
  try "EG ~EX EX (a&b&c&d&e)"
  try "EX ~(a|b|c|d|e)"

  init_nfsm_glasses()

  try "~a&~b&~c -> AF ~(d|e)"
  try "AF ~(d|e)"
  try "AG ~(a&b&c)"
  try "AX F"

  init_nfsm_async()

  try "d&~c -> AX AX A( ~d U c)"
  try "d&~c -> A(d|~c U c)"
  try "EG ~(a&b&c&d)"
  try "EX EX EX EX EX EX (a&b&c&d)"

  RETURN 0
```

```
FUN edge : v1, v2 => rpredsv2 XOR:= 1<<v1 // Add/remove an edge

FUN init_nfsm_5Dcube : =>
  writef "\n5D Cube\n"
  FOR v = #b00000 TO #b11111 DO preds!v := 0

  FOR v = #b00000 TO #b11111 DO // Form a 5D cube with all edges
  { edge(v, v XOR #b00001)
    edge(v, v XOR #b00010)
    edge(v, v XOR #b00100)
    edge(v, v XOR #b01000)
    edge(v, v XOR #b10000)
  }
  edge(#b11111, #b00000)        // But, add one more edge
  edge(#b11000, #b11100)        // and  remove one edge

FUN init_nfsm_glasses : =>
  writef "\nThe Glasses Game\n"
  FOR v = #b00000 TO #b11111 DO preds!v := 0

  // A state is represented by two octal digits #gm
  // where g=0 means all glasses are the same way up
  //       g=1 means one glass is the wrong way up
  //       g=2 means two adjacent glasses are the wrong way up
  //       g=3 means two opposite glasses are the wrong way up
  // and   m=0..7 is the move number.

  move2x 0; move2a 1; move2x 2; move1 3; move2x 4; move2a 5; move2x 6

FUN move1  : i => edge(#10+i,#01+i) // Turn one glass over
                  edge(#10+i,#21+i)
                  edge(#10+i,#31+i)
                  edge(#20+i,#11+i)
                  edge(#30+i,#11+i)

FUN move2x : i => edge(#10+i,#11+i) // Turn two opposite glasses over
                  edge(#20+i,#21+i)
                  edge(#30+i,#01+i)

FUN move2a : i => edge(#10+i,#11+i) // Turn two adjacent glasses over
                  edge(#20+i,#01+i)
                  edge(#20+i,#31+i)
                  edge(#30+i,#21+i)

FUN init_nfsm_async : =>
  writef "\nAn Asynchronous Circuit\n"
  FOR v = #b00000 TO #b11111 DO preds!v := 0
  edge( 2, 0); edge( 2, 1); edge( 2, 3); edge( 0, 1)
  edge( 3, 1); edge( 7, 6); edge( 7, 4); edge( 7, 5)
  edge( 6, 4); edge( 5, 4); edge(13,15); edge(13,14)
  edge(13,12); edge(15,14); edge(12,14); edge( 8, 9)
  edge( 8,11); edge( 8,10); edge( 9,11); edge(10,11)
  edge( 1, 5); edge( 3, 5); edge( 3, 7); edge( 4,12)
  edge( 5,12); edge( 5,13); edge(14,10); edge(12,10)
  edge(12, 8); edge(10, 2); edge(10, 3); edge(11, 3)
```

2.6 The Output from the CTL Checker

The program exercises the model checker on three simple non deterministic finite state machines. The first is essentially a five dimensional cube whose vertices have coordinates `edcba`. From each vertex there are five outgoing edges to vertices that differ in only one coordinate variable. For this demonstration the edge `11000->11100` has been removed and the edge `11111->00000` has been added. These changes show up in some of the tests below.

```
5D Cube

d&e->a&b&c

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000   11001   11010   11011   11100   11101   11110   11111 Y

EX a & EX b & EX c & EX d & EX e

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000   11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

EX EX (a&b&c&d&e)

00000   00001   00010   00011   00100   00101   00110   00111 Y
01000   01001   01010   01011 Y 01100   01101 Y 01110 Y 01111
10000   10001   10010   10011 Y 10100   10101 Y 10110 Y 10111
11000   11001 Y 11010 Y 11011   11100 Y 11101   11110   11111 Y

EG ~EX EX (a&b&c&d&e)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111
01000 Y 01001 Y 01010 Y 01011   01100 Y 01101   01110   01111
10000 Y 10001 Y 10010 Y 10011   10100 Y 10101   10110   10111
11000 Y 11001   11010   11011   11100   11101   11110   11111

EX ~(a|b|c|d|e)

00000   00001 Y 00010 Y 00011   00100 Y 00101   00110   00111
01000 Y 01001   01010   01011   01100   01101   01110   01111
10000 Y 10001   10010   10011   10100   10101   10110   10111
11000   11001   11010   11011   11100   11101   11110   11111 Y
```

The second example and its solution was suggested by Stewart and VanInwegen[SV97]. It is based on a game concerned with four empty glasses at the corners of a square tray. Initially some of the glasses may be upside-down. A player can cause (M1) one glass, or (M2A) two adjacent glasses, or (M2X) two opposite glasses to be turned over, but there is the complication that the tray is out of the sight of the player and may be rotated at any time and so the player cannot specify precisely which glasses are turned over. The game stops when all the glasses are the same way up. The move sequence M2X-M2A-M2X-M1-M2X-M2A-M2X guarantees that the game terminates in no more than 7 steps. An NFSM for this game with 32 states is shown in figure 1.

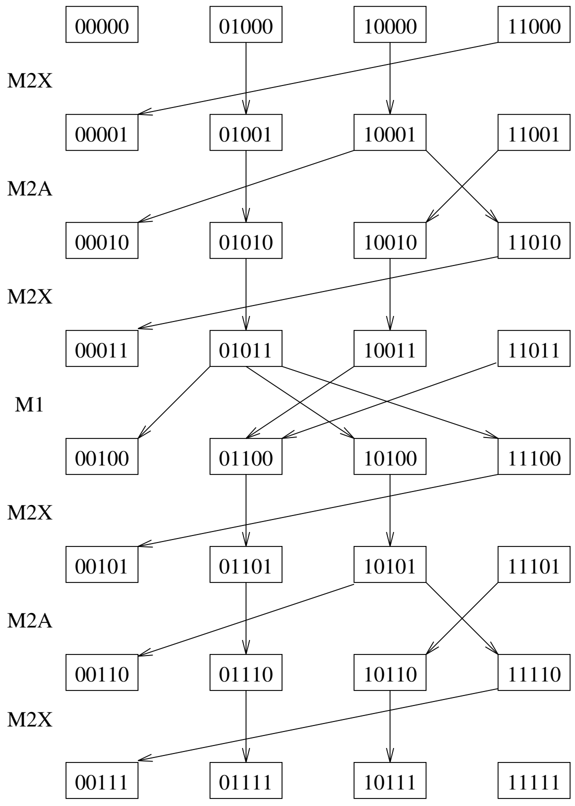

Figure 1: The Glasses Game NFSM

The state of the tray is represented by `ed=00` for all glasses the same way up, `ed=01` for one oriented differently from the other three, `ed=10` for two adjacent glasses oriented differently from the other two, and `ed=11` for two opposite glasses oriented differently from the other two, and the number of steps taken so far is represented by `cba`. Thus, the possible initial states are: `00000`, `01000`, `10000` and `11000`, and the final states have the form: `00XXX`.

That every initial state leads to a final state can be encoded in CTL as: `~a&~b&~c -> AF ~(d|e)`. The evaluation of this and other formulae are shown below:

```
The Glasses Game

~a&~b&~c -> AF ~(d|e)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000 Y 11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

AF ~(d|e)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100   01101   01110   01111
10000 Y 10001 Y 10010   10011   10100 Y 10101 Y 10110   10111
11000 Y 11001   11010 Y 11011   11100 Y 11101   11110 Y 11111

AG ~(a&b&c)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111
01000   01001   01010   01011   01100   01101   01110   01111
10000 Y 10001 Y 10010   10011   10100   10101   10110   10111
11000 Y 11001   11010 Y 11011   11100 Y 11101   11110   11111

AX F

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000   01001   01010   01011   01100   01101   01110   01111 Y
10000   10001   10010   10011   10100   10101   10110   10111 Y
11000   11001   11010   11011   11100   11101   11110   11111 Y
```

As a final example, a simple asynchronous circuit is tested. There are four signals held in a four bit word `dcba`. The circuit is designed so that the signal values change according to the following rules:

```
a := ~c
b := d
c := a&~d + c&(a|~d)
d := c&~b + d&(c|~b)
```

but the assignments have random delays and so the number of signals that change at any transition is not deterministic. The possible transitions, represented as an NFSM, are shown in figure 2. Some tests with this circuit then follow.

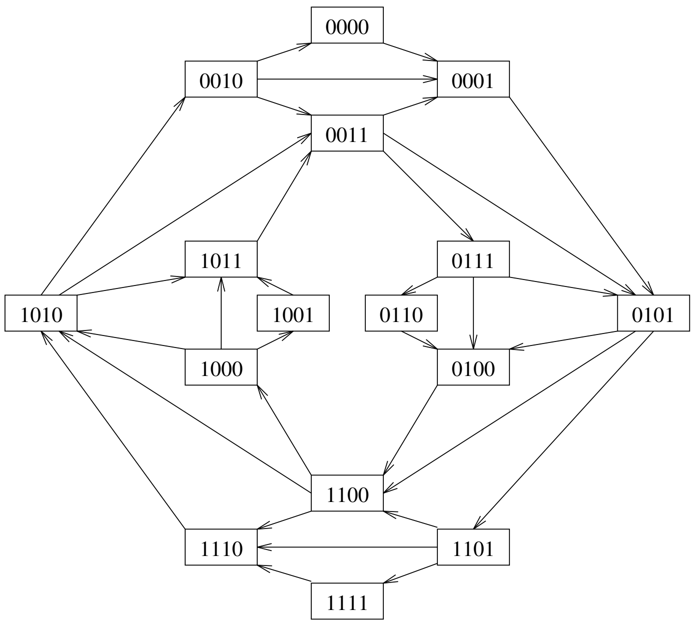

Figure 2: The Asynchronous Circuit NFSM

```
An Asynchronous Circuit

d&~c -> AX AX A( ~d U c)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000   01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000 Y 11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

d&~c -> A(d|~c U c)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000   11001   11010   11011   11100 Y 11101 Y 11110 Y 11111 Y

EG ~(a&b&c&d)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000 Y 11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111

EX EX EX EX EX EX (a&b&c&d)

00000   00001   00010   00011   00100 Y 00101 Y 00110   00111
01000 Y 01001 Y 01010 Y 01011   01100 Y 01101 Y 01110 Y 01111 Y
10000   10001   10010   10011   10100   10101   10110   10111
11000   11001   11010   11011   11100   11101   11110   11111
```

3 A μ-Calculus Model Checker

Another language for describing properties of transition systems is the μ-calculus, and, as with CTL, it can be used in a model checker. Both the version of μ-calculus and the checking algorithm presented here are based on the paper by Berezin et al.[SBM96]. The μ-Calculus extends CTL by, firstly, giving the non deterministic machine labels (called actions) on its transitions and, secondly, having explicit constructs for both the least and greatest fixed point operators. The resulting logic is more powerful than CTL but still simple enough to form the basis of a symbolic model checker.

3.1 A syntax for μ-calculus

Formulae in μ-calculus are formed as follows:

- `a`, `b`, `c`, `d`, `e`, `T`, `F`, `x`, `y` and `z` are formulae
- assuming f and g are formulae then so are:

`(`f`)`,	`~`f,	f`=`g,
f`&`g,	f`|`g,	f`->`g,
`<p>`f,	`<q>`f,	`<r>`f,
`[p]`f,	`[q]`f,	`[r]`f,
`Mx.`f,	`My.`f,	`Mz.`f,
`Nx.`f,	`Ny.`f,	`Nz.`f

As an example, the following is a syntactically correct formula:

```
Ny.(<r>Mx.(<r>x | y&(a&b&c&d)))
```

which, with the semantics given below, will evaluate to give the set of states for which there exist a path, using `r`-transitions, that visits either state 01111 or 11111 infinitely often. This is an example of a property that cannot be stated in CTL.

3.2 Semantics of μ-calculus

A μ-calculus formula is evaluated with respect to an NSFM and an environment to yield a subset of the states of the NFSM for which the formula is said to be satisfied. We will assume that the set of states is S and we will assume that any state (s, say) in S can be identified by a 5-bit binary integer *edcba*.

We will use $s \xrightarrow{\mathtt{p}} t$ to mean that there is a transition in the NFSM from s to t labelled `p`, and we define $s \xrightarrow{\mathtt{q}} t$ and $s \xrightarrow{\mathtt{r}} t$ similarly.

An environment (e, say) is a mapping from the relational variables `x`, `y` and `z` to subsets of the S. We use $e(\mathtt{x})$, $e(\mathtt{y})$ and $e(\mathtt{z})$ to denote these subsets, and we use $e[X/\mathtt{x}]$ to denote an environment identical to e except that $e(\mathtt{x}) = X$. The expressions: $e[Y/\mathtt{y}]$ and $e[Z/\mathtt{z}]$ are defined similarly.

We will use $[\![f]\!]\ e$ to denote the subset of S for which the formula f is satisfied in the environment e. It is defined recursively as follows:

$$
\begin{array}{ll}
[\![\mathtt{a}]\!]\ e & = \{\ s \epsilon S \mid s = edcba \text{ and } a = 1\ \} \\
 & \text{and similarly for } [\![\mathtt{b}]\!], [\![\mathtt{c}]\!], [\![\mathtt{d}]\!] \text{ and } [\![\mathtt{e}]\!] \\
[\![\mathtt{T}]\!]\ e & = S \\
[\![\mathtt{F}]\!]\ e & = \phi \\
[\![\mathtt{x}]\!]\ e & = e(\mathtt{x}) \\
 & \text{and similarly for } [\![\mathtt{y}]\!] \text{ and } [\![\mathtt{z}]\!] \\
[\![\mathtt{(}f\mathtt{)}]\!]\ e & = [\![f]\!]\ e \\
[\![\mathtt{\char`~}f]\!]\ e & = S \ - \ [\![f]\!]\ e \\
[\![f\mathtt{\&}g]\!]\ e & = [\![f]\!]\ e \ \cap \ [\![g]\!]\ e \\
[\![f\mathtt{|}g]\!]\ e & = [\![f]\!]\ e \ \cup \ [\![g]\!]\ e \\
[\![f\mathtt{->}g]\!]\ e & = (S \ - \ [\![f]\!]\ e) \ \cup \ [\![g]\!]\ e \\
[\![f\mathtt{=}g]\!]\ e & = [\![f\mathtt{->}g]\!]\ e \ \cap \ [\![g\mathtt{->}f]\!]\ e \\
[\![\mathtt{<p>}f]\!]\ e & = \{\ s \mid \exists t.\ (s \xrightarrow{\mathtt{p}} t) \wedge (t \epsilon [\![f]\!]\ e)\ \} \\
 & \text{and similarly for } [\![\mathtt{<q>}f]\!] \text{ and } [\![\mathtt{<r>}f]\!] \\
[\![\mathtt{[p]}f]\!]\ e & = \{\ s \mid \forall t.\ (s \xrightarrow{\mathtt{p}} t) \Rightarrow (t \epsilon [\![f]\!]\ e)\ \} \\
 & \text{and similarly for } [\![\mathtt{[q]}f]\!] \text{ and } [\![\mathtt{[r]}f]\!] \\
[\![\mathtt{Mx.}f]\!]\ e & = \bigcup_i \tau^i(\phi) \ \text{ where } \ \tau(X) = [\![f]\!]\ (e[X/\mathtt{x}]) \\
 & \text{and similarly for } [\![\mathtt{My.}f]\!] \text{ and } [\![\mathtt{Mz.}f]\!] \\
[\![\mathtt{Nx.}f]\!]\ e & = \bigcap_i \tau^i(S) \ \text{ where } \ \tau(X) = [\![f]\!]\ (e[X/\mathtt{x}]) \\
 & \text{and similarly for } [\![\mathtt{Ny.}f]\!] \text{ and } [\![\mathtt{Nz.}f]\!]
\end{array}
$$

To ensure that the fixed point operators (`M` and `N`) are properly defined, the relational variables `x`, `y` and `z` may only occur under an even number of negations, after replacing f`->`g by `~`f`|`g, and f`=`g by (`~`f`|`g)`&`(`~`g`|`f).

3.3 Syntax Analysis

The program given in Section 3.5 parses and evaluates various μ-calculus expressions. The structure of the parse tree is as follows:

T	$\rightarrow$	`[Atom,`	*bits*`]`		—	`a`, `b`, `c`, `d`, `e`, `T`, `F`
		`[Var,`	v`]`		—	`x`, `y`, `z`
		`[Not,`	T_f`]`		—	`~`f
		`[Eq,`	T_f,	T_g`]`	—	f`=`g
		`[And,`	T_f,	T_g`]`	—	f`&`g
		`[Or,`	T_f,	T_g`]`	—	f`|`g
		`[Imp,`	T_f,	T_g`]`	—	f`->`g
		`[EX,`	a,	T_f`]`	—	`<p>`f, `<q>`f, `<r>`f
		`[AX,`	a,	T_f`]`	—	`[p]`f, `[q]`f, `[r]`f
		`[Mu,`	v,	T_f`]`	—	`Mx.`f, `My.`f, `Mz.`f
		`[Nu,`	v,	T_f`]`	—	`Nx.`f, `Ny.`f, `Nz.`f

where

bits is a bit pattern representing a subset of S,
$v = 0$, 1 or 2 representing variable `x`, `y` or `z`, respectively,
$a = 0$, 1 or 2 representing variable `p`, `q` or `r`, respectively, and
T_f and T_g represent the parse trees for f and g.

The lexical analyser `lex` and syntax analyser `parse` are similar to those used in the CTL checker described above. However, in this implementation a check is made that the formula is well formed with respect to negation of the relational variables `x`, `y` and `z`. This is done by the call of `wff` in `parse` just after the formula has been parsed.

The parse tree can be printed using the function `prtree`. For instance, the output generated by `prtree` for the formula:

```
Ny.(<q>T & [q]Mx.(<q>T & [q]x | y&(b&c&d)))
```

is as follows:

```
Ny
*-And
  *-<q>
  ! *-T
  *-[q]
    *-Mx
      *-Or
        *-And
        ! *-<q>
        ! ! *-T
        ! *-[q]
        !   *-x
        *-And
          *-y
          *-And
            *-And
            ! *-b
            ! *-c
            *-d
```

3.4 The Model Checking Algorithm

The model checking algorithm is rather simpler than the CTL version, even though there are now three 32×32 bits matrices, representing the `p-`, `q-` and `r-`transitions of the NFSM. These are held in `preds!0`, `preds!1`, `preds!2`, respectively. NFSM edges are added or removed by the function `edge` defined as follows:

```
FUN edge
: lab, s, t => preds!lab!t XOR:= 1<<s // Add/remove an edge
```

The edge label `lab` equals 0, 1 or 2 to represent `p`, `q` or `r`, respectively. The expression `preds!1!t`, for example, yields a bit pattern representing the set of `q`-predecessors of state `t`.

The function `eval` takes two arguments. The first is the tree representation of the formula to evaluate and the second is a vector (`e`, say) of three elements representing the environment. The values of `x`, `y` and `z` in this environment are `e!0`, `e!1` and `e!2`, respectively. There are match items to deal with each possible operator in the tree. The operators `Atom`, `Var`, `Not`, `And`, `Or`, `Imp` and `Eq` are straightforward and need no further explanation.

Formulae of the form `<x>`f, `<y>`f or `<z>`f are evaluated by the by the match item:

```
: [EX, lab, f], e =>   evalEX(lab, eval(f,e))
```

which first evaluate f in the current environment and then invokes `evalEX` to compute the set of `lab`-predecessors of all states for which f is satisfied. The definition of `evalEX` is as follows:

```
FUN evalEX : lab, w =>
  LET res = 0
  LET p   = preds!lab
  WHILE w DO { IF w&1 DO res |:= !p
               w >>:= 1
               p+++
             }
  RETURN res
```

The evaluation of formulae of the form `[x]`f, `[y]`f or `[z]`f also use `evalEX`, based on the observation that `[`v`]`f = `~<`v`>~`f.

Formulae involving the fixed point operators are evaluated by iteration until a convergent result is obtained. For instance, a formula of the form `M`v`.`f is evaluated by the call: `fix(False,`v`,`T_f`,e)` where v is 0, 1 or 2 representing `x`, `y` or `z`, and T_f is the tree representation of formula f. The definition of `fix` is as follows:

```
FUN fix : t, v, f, e[x,y,z] =>  // Compute: [[Mv.f]] e, if t=False
                                // or       [[Nv.f]] e, if t=True
  LET env = [x,y,z]             // Make a copy of e
  { env!v := t                  // env = e[t/v]
    LET a = eval(f, env)
    IF a=t RETURN t             // Return when converged
    t := a
  } REPEAT
```

The environment is copied so that its vth element can be updated on each interation, without disturbing the outer environment `e`. The successive approximations are held in `t` starting from a given initial value. When two successive approximations are equal the result is returned. If `t` is initially `False` the iteration yields the minimal fixed point, and if `True` the maximal fixed point is found. The iteration will converge if f monotonic, and in most cases the convergence will be rapid.

3.5 The μ-Calculus Model Checker Program

```
GET "mcpl.h"

MANIFEST
Id, Atom, Var, Lab, Not, And, Or, Imp, Eq,      // Tokens
Mu, Nu, EX, AX,
Dot, Rbra, Rket, Sbra, Sket, Abra, Aket, Eof,

E_syntax=100, E_space, E_eval, E_wff,           // Exceptions

                         // Atomic boolean functions
True= #xFFFFFFFF,        // f(a,b,c,d,e) = T
False=#x00000000,        // f(a,b,c,d,e) = F

Abits=#xAAAAAAAA,        // f(a,b,c,d,e) = a
Bbits=#xCCCCCCCC,        // f(a,b,c,d,e) = b
Cbits=#xF0F0F0F0,        // f(a,b,c,d,e) = c
Dbits=#xFF00FF00,        // f(a,b,c,d,e) = d
Ebits=#xFFFF0000         // f(a,b,c,d,e) = e
```

```
//********** Model checking algorithm ************************

// The transition relation will be represented by the vector preds
// preds!lab!i will be the bit pattern representing the set of
//             immediate lab-predecessors of state i

STATIC preds = [ VEC #b11111,  // p-predecessors   ) all
                 VEC #b11111,  // q-predecessors   ) initialised
                 VEC #b11111   // r-predecessors   ) later
               ]

FUN eval
: [Atom, bits   ], e =>   bits
: [Var,     v   ], e =>   e!v        // v = 0, 1 or 2
: [Not,     f   ], e => ~ eval(f,e)
: [And,     f, g], e =>   eval(f,e)  &  eval(g,e)
: [Or,      f, g], e =>   eval(f,e)  |  eval(g,e)
: [Imp,     f, g], e => ~ eval(f,e)  |  eval(g,e)
: [Eq,      f, g], e => ~(eval(f,e) XOR eval(g,e))
: [EX,    lab, f], e =>   evalEX(lab,   eval(f,e))
: [AX,    lab, f], e => ~ evalEX(lab, ~ eval(f,e))
: [Mu,      v, f], e =>   fix(False, v, f, e)
: [Nu,      v, f], e =>   fix(True,  v, f, e)
:                    =>   RAISE E_eval

FUN evalEX : lab, w => // the lab-predecessors of all states in w
  LET res = 0
  LET p   = preds!lab
  WHILE w DO { IF w&1 DO res |:= !p
               w >>:= 1
               p+++
             }
  RETURN res

FUN fix : t, v, f, e[x,y,z] =>  // Compute: [[Mv.f]] e, if t=False
                                // or       [[Nv.f]] e, if t=True
  LET env = [x,y,z]             // Make a copy of e
  { env!v := t                  // env = e[t/v]
    LET a = eval(f, env)
    IF a=t RETURN t             // Return when converged
    t := a
  } REPEAT

//********** End of Model checking algorithm *****************
```

```
/******************** Syntax Analyser ***********************

STATIC  str, strp, ch, nch, token, lexval

FUN rch : => ch, nch := nch, %strp
             IF nch DO strp++

FUN lex_init : s => str := s; strp := str; rch(); rch()

FUN lex : => MATCH (ch, nch)
: ' ' | '\n' => rch(); lex()     // Ignore white space
:  0         => token := Eof     // End of file
: 'a'        => token := Id;     lexval := Abits;  rch()
: 'b'        => token := Id;     lexval := Bbits;  rch()
: 'c'        => token := Id;     lexval := Cbits;  rch()
: 'd'        => token := Id;     lexval := Dbits;  rch()
: 'e'        => token := Id;     lexval := Ebits;  rch()
: 'T'        => token := Id;     lexval := True;   rch()
: 'F'        => token := Id;     lexval := False;  rch()
: 'x'..'z'   => token := Var;    lexval := ch-'x'; rch()
: 'p'..'r'   => token := Lab;    lexval := ch-'p'; rch()
: '('        => token := Rbra;                     rch()
: ')'        => token := Rket;                     rch()
: '<'        => token := Abra;                     rch()
: '>'        => token := Aket;                     rch()
: '['        => token := Sbra;                     rch()
: ']'        => token := Sket;                     rch()
: '~'        => token := Not;                      rch()
: '='        => token := Eq;                       rch()
: '&'        => token := And;                      rch()
: '|'        => token := Or;                       rch()
: '-', '>'   => token := Imp;               rch(); rch()
: 'M'        => token := Mu;                       rch()
: 'N'        => token := Nu;                       rch()
: '.'        => token := Dot;                      rch()
:            => RAISE E_syntax


FUN wff     // Check that every relational variable is declared
            // before use and under an even number of NOTs.
            // Convention concerning e and variable v:
            //   e!v = 0  v not declared
            //       = 1  v declared and under an even number of NOTs
            //       =-1  v declared and under an odd  number of NOTs
: [Atom, bits],    e        => RETURN
: [Var,     v],    e        => UNLESS e!v=1 RAISE E_wff
: [Not,     f],    e[x,y,z] => wff(f, [-x,-y,-z])
: [And|Or,  f, g], e        => wff(f, e);     wff(g, e)
: [Imp,     f, g], e[x,y,z] => wff(f, [-x,-y,-z]); wff(g, e)
: [Eq,      f, g], e        => wff([And, [Or, [Not, f], g],
                                         [Or, [Not, g], f]], e)
: [EX|AX, lab, f], e        => wff(f, e)
: [Mu|Nu,   v, f], e[x,y,z] => LET env = [x,y,z]
                               env!v := 1
                               wff(f, env)
:                           => RAISE E_wff
```

```
FUN parse : str => lex_init str;
                   LET tree = nexp 0
                   chkfor Eof
                   wff(tree, [0,0,0])
                   RETURN tree

FUN chkfor : tok => UNLESS token=tok RAISE E_syntax
                    lex()

FUN prim : =>
  LET op = token

  MATCH op
  : Id       => LET a = lexval; lex(); RETURN mk2(Atom, a)
  : Var      => LET a = lexval; lex(); RETURN mk2(Var,  a)
  : Rbra     => LET a = nexp 0; chkfor Rket; RETURN a
  : Not      => mk2(op, nexp 5)

  : Abra     => lex()                   // <p>T,  <q>T  or <r>T
                LET a = lexval
                chkfor Lab
                chkfor Aket
                RETURN mk3(EX, a, exp 5)

  : Sbra     => lex()                   // [p]T,  [q]T  or [r]T
                LET a = lexval
                chkfor Lab
                chkfor Sket
                RETURN mk3(AX, a, exp 5)

  : Mu | Nu  => lex()     // Mx.T, My.T, Mz.T, Nx.T, Ny.T or Nz.T
                LET a = lexval
                chkfor Var
                chkfor Dot
                RETURN mk3(op, a, exp 0)

  :          => RAISE E_syntax

FUN nexp : n => lex(); exp n

FUN  exp : n => LET a = prim()
                MATCH (token, n)
                : Eq,  <4 => a := mk3(Eq,  a, nexp 4)
                : And, <3 => a := mk3(And, a, nexp 3)
                : Or,  <2 => a := mk3(Or,  a, nexp 2)
                : Imp, <1 => a := mk3(Imp, a, nexp 1)
                :         => RETURN a
                . REPEAT
```

```
//******************** Space Allocation *****************

STATIC  spacev, spacep

FUN mk_init : upb    => spacev := getvec upb
                        UNLESS spacev RAISE E_space
                        spacep := @ spacev!upb

FUN mk_close :       => freevec spacev

FUN mk1 : x          => !---spacep := x; spacep
FUN mk2 : x, y       => mk1 y; mk1 x
FUN mk3 : x, y, z    => mk1 z; mk1 y; mk1 x

//************** Print tree function *********************

STATIC prlinev = VEC 50

FUN prtree
: 0,     ?,        ? => writef "Nil"
: ?, depth,   =depth => writef "Etc"
: x, depth, maxdepth =>
  LET upb = 1
  MATCH x
  : [Atom, =Abits]  => writef "a";              RETURN
  : [Atom, =Bbits]  => writef "b";              RETURN
  : [Atom, =Cbits]  => writef "c";              RETURN
  : [Atom, =Dbits]  => writef "d";              RETURN
  : [Atom, =Ebits]  => writef "e";              RETURN
  : [Atom, =True]   => writef "T";              RETURN
  : [Atom, =False]  => writef "F";              RETURN
  : [Var, v]        => writef("%c", 'x'+v);     RETURN
  : [Not, f]        => writes "Not"
  : [Eq,  f, g]     => writes "Eq";             upb := 2
  : [And, f, g]     => writes "And";            upb := 2
  : [Or,  f, g]     => writes "Or";             upb := 2
  : [Imp, f, g]     => writes "Imp";            upb := 2
  : [EX,lab, f]     => writef("<%c>", 'p'+lab); x+++
  : [AX,lab, f]     => writef("[%c]", 'p'+lab); x+++
  : [Mu,  v, f]     => writef("M%c",  'x'+v);   x+++
  : [Nu,  v, f]     => writef("N%c",  'x'+v);   x+++
  :                 => writes "Unknown";        upb := 0
  .
  FOR i = 1 TO upb DO { newline()
                        FOR j=0 TO depth-1 DO writes( prlinev!j )
                        writes("*-")
                        prlinev!depth := i=upb-> "  ", "! "
                        prtree(x!i, depth+1, maxdepth)
                       }
```

```
//******************** Main Program ************************

FUN try : e =>
  { mk_init 100_000
    writef("\n%s\n", e)
    LET exp = parse e
    prtree(exp, 0, 20)
    LET res = eval(exp, [0,0,0])
    FOR v = #b00000 TO #b11111 DO
    { UNLESS v MOD 8 DO newline()
      writef("%5b %c ", v, res&1=0->' ', 'Y')
      res >>:= 1
    }
    newline()
  } HANDLE : E_syntax => writef("Bad Syntax\n%s\n", formula)
                         FOR i = 0 TO strp-formula-3 DO wrch ' '
                         writes "^\n"
           : E_space  => writef "Insufficient space\n"
           : E_wff    => writef "Expression not well formed\n"
           : E_eval   => writef "Error in eval\n"
           .
  mk_close()

FUN start : =>
  init_nfsm()

  writes "\nTest the 5D cube -- using p-edges\n"

  writes "\nCTL: d&e->a&b&c"
  try "d&e->a&b&c"

  writes "\nCTL: EX a & EX b & EX c & EX d & EX e"
  try "<p>a & <p>b & <p>c & <p>d & <p>e"

  writes "\nCTL: EX EX (a&b&c&d&e)"
  try "<p><p> (a&b&c&d&e)"

  writes "\nCTL: EG ~EX EX (a&b&c&d&e)"
  try "Nx. (~<p><p>(a&b&c&d&e)) & <p>x"

  writes "\nCTL: EX ~(a|b|c|d|e)"
  try "<p> ~(a|b|c|d|e)"

  writes "\nTest the Glasses game -- using q-edges\n"

  writes "\nCTL: ~a&~b&~c -> AF ~(d|e)"
  try "~a&~b&~c -> ~Nx.((d|e) & (<r>x | [q]F))"

  writes "\nCTL: AF ~(d|e)"
  try "~Nx.((d|e) & (<q>x | [q]F))"

  writes "\nCTL: AG ~(a&b&c)"
  try "~Mx.(a&b&c | <q>x)"

  writes "\nCTL: AX F"
  try "[q]F"
  writes "\nTest the asynchronous circuit -- using r-edges\n"

  writes "\nCTL: d&~c -> AX AX A( ~d U c)"
  try "d&~c -> [r][r] Mx.(c | ~d & [r]x & ~[r]F)"

  writes "\nCTL: d&~c -> A(d|~c U c)"
  try "d&~c -> Mx.(c | (d|~c) & [r]x & ~[r]F)"

  writes "\nCTL: EG ~(a&b&c&d)"
  try "Nx.(~(a&b&c&d)) & (<r>x | [r]F)"

  writes "\nCTL: EX EX EX EX EX EX (a&b&c&d)"
  try "<r><r><r><r><r><r> (a&b&c&d)"

  writes "\nExists a path visiting state 10000 infinitely often"
  try "Ny.(<r>Mx.(<r>x | y&(e&~d&~c&~b&~a)))"

  writes "\nIn all paths (b&c&d) occurs infinitely often"
  try "Ny.(<r>T & [r]Mx.(<r>T & [r]x | y&(b&c&d)))"

  writes "\nThere is a path of length 6n to state 01111 or 11111"
  try "My.((a&b&c&d) | <r><r><r><r><r><r>y)"

  writes "\nStates having paths: rqprqp..."
  try "Nx.(x & <r><q><p>x)"

  writes "\nStates having paths: pqrr... to state 11111"
  try "<p><q>My.(<r>y|a&b&c&d&e)"

  writes "\nStates having qr-paths to 11111 that must pass through 01011"
  try "  (Mx. a&b&c&d&e | <r>x | <q>x)             \n\
      \&~(Mx. a&b&c&d&e | <r>x | <q>x & (e|~d|c|~b|~a))"

  RETURN 0

FUN edge
: lab, v1, v2 => preds!lab!v2 XOR:= 1<<v1 // Add/remove an edge

FUN init_nfsm : =>
  FOR v = #b00000 TO #b11111 DO { preds!0!v := 0 // p-preds
                                  preds!1!v := 0 // q-preds
                                  preds!2!v := 0 // r-preds
                                }

                                // Using p-edges,
  FOR v = #b00000 TO #b11111 DO // form a 5D cube with all edges
  { edge(0, v, v XOR #b00001)
    edge(0, v, v XOR #b00010)
    edge(0, v, v XOR #b00100)
    edge(0, v, v XOR #b01000)
    edge(0, v, v XOR #b10000)
  }
  edge(0, #b11111, #b00000)     // But, add one more edge
  edge(0, #b11000, #b11100)     // and  remove one edge
  // The Glasses Game        -- using q-edges
  // A state is represented by two oct digits #gm
  // where g=0 means all glasses are the same way up
  //       g=1 means one glass is the wrong way up
  //       g=2 means two adjacent glasses are the wrong way up
  //       g=3 means two opposite glasses are the wrong way up
  // and   m=0..7 is the move number.

  move2x 0; move2a 1; move2x 2; move1 3; move2x 4; move2a 5; move2x 6

  // An asynchronous circuit -- using r-edges
  edge(2, 2, 0); edge(2, 2, 1); edge(2, 2, 3); edge(2, 0, 1)
  edge(2, 3, 1); edge(2, 7, 6); edge(2, 7, 4); edge(2, 7, 5)
  edge(2, 6, 4); edge(2, 5, 4); edge(2,13,15); edge(2,13,14)
  edge(2,13,12); edge(2,15,14); edge(2,12,14); edge(2, 8, 9)
  edge(2, 8,11); edge(2, 8,10); edge(2, 9,11); edge(2,10,11)
  edge(2, 1, 5); edge(2, 3, 5); edge(2, 3, 7); edge(2, 4,12)
  edge(2, 5,12); edge(2, 5,13); edge(2,14,10); edge(2,12,10)
  edge(2,12, 8); edge(2,10, 2); edge(2,10, 3); edge(2,11, 3)

  // Add a few more edges to show off the power of mu-calculus
  edge(2,16,17); edge(2,17,18); edge(2,18,19) // a path from 16->22
  edge(2,19,20); edge(2,20,21); edge(2,21,22)

  edge(2,22,23); edge(2,23,25); edge(2,25,27); edge(2,27,29)
                                edge(2,25,16) // loop back to 16
  edge(2,22,24); edge(2,24,26); edge(2,26,28); edge(2,28,30)

  edge(2,29,30); edge(2,30,31); edge(2,31,29) // a 3 edge loop

FUN move1  : i => edge(1, #10+i,#01+i) // Turn one glass over
                  edge(1, #10+i,#21+i)
                  edge(1, #10+i,#31+i)
                  edge(1, #20+i,#11+i)
                  edge(1, #30+i,#11+i)

FUN move2x : i => edge(1, #10+i,#11+i) // Turn two opposite glasses over
                  edge(1, #20+i,#21+i)
                  edge(1, #30+i,#01+i)

FUN move2a : i => edge(1, #10+i,#11+i) // Turn two adjacent glasses over
                  edge(1, #20+i,#01+i)
                  edge(1, #20+i,#31+i)
                  edge(1, #30+i,#21+i)
```

3.6 Output from the μ-Calculus Checker

The first few examples use the 5D cube example given above to illustrate that CTL formulae can be translated into μ-calculus.

```
Test the 5D cube -- using p-edges

CTL: d&e->a&b&c
d&e->a&b&c

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000   11001   11010   11011   11100   11101   11110   11111 Y

CTL: EX a & EX b & EX c & EX d & EX e
<p>a & <p>b & <p>c & <p>d & <p>e

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000   11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

CTL: EX EX (a&b&c&d&e)
<p><p> (a&b&c&d&e)

00000   00001   00010   00011   00100   00101   00110   00111 Y
01000   01001   01010   01011 Y 01100   01101 Y 01110 Y 01111
10000   10001   10010   10011 Y 10100   10101 Y 10110 Y 10111
11000   11001 Y 11010 Y 11011   11100 Y 11101   11110   11111 Y

CTL: EG ~EX EX (a&b&c&d&e)
Nx. (~<p><p>(a&b&c&d&e)) & <p>x

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111
01000 Y 01001 Y 01010 Y 01011   01100 Y 01101   01110   01111
10000 Y 10001 Y 10010 Y 10011   10100 Y 10101   10110   10111
11000 Y 11001   11010   11011   11100   11101   11110   11111

CTL: EX ~(a|b|c|d|e)
<p> ~(a|b|c|d|e)

00000   00001 Y 00010 Y 00011   00100 Y 00101   00110   00111
01000 Y 01001   01010   01011   01100   01101   01110   01111
10000 Y 10001   10010   10011   10100   10101   10110   10111
11000   11001   11010   11011   11100   11101   11110   11111 Y
```

The second example uses q-edges to represent the glasses game given above. Again it shows that the CTL formulae can be translated into μ-calculus.

```
Test the Glasses game -- using q-edges

CTL: ~a&~b&~c -> AF ~(d|e)
~a&~b&~c -> ~Nx.((d|e) & (<q>x | [q]F))

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000 Y 11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

CTL: AF ~(d|e)
~Nx.((d|e) & (<q>x | [q]F))

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100   01101   01110   01111
10000 Y 10001 Y 10010   10011   10100 Y 10101 Y 10110   10111
11000 Y 11001   11010 Y 11011   11100 Y 11101   11110 Y 11111

CTL: AG ~(a&b&c)
~Mx.(a&b&c | <q>x)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111
01000   01001   01010   01011   01100   01101   01110   01111
10000 Y 10001 Y 10010   10011   10100   10101   10110   10111
11000 Y 11001   11010 Y 11011   11100 Y 11101   11110   11111

CTL: AX F
[q]F

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000   01001   01010   01011   01100   01101   01110   01111 Y
10000   10001   10010   10011   10100   10101   10110   10111 Y
11000   11001   11010   11011   11100   11101   11110   11111 Y
```

The tests that follow use `r`-edges the NFSM given in figure 2 augmented by the set of extra edges shown in figure 3.

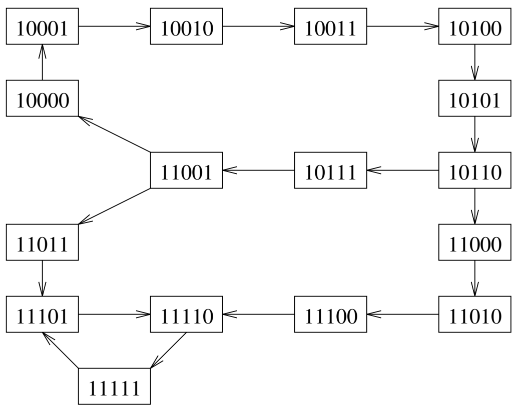

Figure 3: Extra `r`-edges

```
Test the asynchronous circuit -- using r-edges

CTL: d&~c -> AX AX A( ~d U c)
d&~c -> [r][r] Mx.(c | ~d & [r]x & ~[r]F)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000   01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000 Y 11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

CTL: d&~c -> A(d|~c U c)
d&~c -> Mx.(c | (d|~c) & [r]x & ~[r]F)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000 Y 11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

CTL: EG ~(a&b&c&d)
Nx.(~(a&b&c&d)) & (<r>x | [r]F)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000   11001 Y 11010   11011   11100   11101   11110   11111

CTL: EX EX EX EX EX EX (a&b&c&d)
<r><r><r><r><r><r> (a&b&c&d)

00000   00001   00010   00011   00100 Y 00101 Y 00110   00111
01000 Y 01001 Y 01010 Y 01011   01100 Y 01101 Y 01110 Y 01111 Y
10000   10001   10010   10011   10100   10101 Y 10110 Y 10111
11000   11001   11010 Y 11011 Y 11100   11101   11110   11111 Y
```

We now give some μ-calculus examples that cannot be specified in CTL.

```
Exists a path visiting state 10000 infinitely often
Ny.(<r>Mx.(<r>x | y&(e&~d&~c&~b&~a)))

00000   00001   00010   00011   00100   00101   00110   00111
01000   01001   01010   01011   01100   01101   01110   01111
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000   11001 Y 11010   11011   11100   11101   11110   11111

In all paths (b&c&d) occurs infinitely often
Ny.(<r>T & [r]Mx.(<r>T & [r]x | y&(b&c&d)))

00000   00001   00010   00011   00100   00101   00110   00111
01000   01001   01010   01011   01100   01101   01110   01111
10000   10001   10010   10011   10100   10101   10110   10111
11000 Y 11001   11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

There is a path of length 6n to state 01111 or 11111
My.((a&b&c&d) | <r><r><r><r><r><r>y)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001   10010 Y 10011 Y 10100   10101 Y 10110 Y 10111
11000   11001 Y 11010 Y 11011 Y 11100   11101   11110   11111 Y
```

Finally, there are three examples that check properties involving simultaneously the `p`, `q` and `r` edges of the combined NFSM.

```
States having paths: rqprqp...
Nx.(x & <r><q><p>x)

00000   00001   00010   00011   00100 Y 00101 Y 00110   00111
01000 Y 01001 Y 01010 Y 01011   01100 Y 01101 Y 01110 Y 01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000 Y 11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110   11111 Y

States having paths: pqrr... to state 11111
<p><q>My.(<r>y|a&b&c&d&e)

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111
01000   01001 Y 01010 Y 01011   01100   01101 Y 01110   01111 Y
10000 Y 10001 Y 10010 Y 10011 Y 10100 Y 10101 Y 10110 Y 10111 Y
11000 Y 11001 Y 11010 Y 11011 Y 11100 Y 11101 Y 11110 Y 11111 Y

States having qr-paths to 11111 that must pass through 01011
  (Mx. a&b&c&d&e | <r>x | <q>x)
&~(Mx. a&b&c&d&e | <r>x | <q>x & (e|~d|c|~b|~a))

00000 Y 00001 Y 00010 Y 00011 Y 00100 Y 00101 Y 00110 Y 00111 Y
01000 Y 01001 Y 01010 Y 01011 Y 01100 Y 01101 Y 01110 Y 01111 Y
10000   10001   10010   10011   10100   10101   10110   10111
11000   11001   11010   11011   11100   11101   11110   11111
```

A Summary of MCPL

In the syntactic forms given below

E	denotes an expression,
K	denotes a constant expression,
A	denotes an argument list,
P	denotes a pattern,
N	denotes a variable name,
M	denotes a manifest name.

A.1 Outermost level declarations

These are the only constructs that can occur at the outermost level of the program.

`MODULE` *N*

This directive must occur first, if present.

`GET` *string*

Insert the file named *string* at this point.

`FUN` *N* `:` *P* `=>` *E* `:..:` *P* `=>` *E* `.`

The main procedure has name: `start`. Functions may only be defined at the outermost level, hence they have no dynamic free variables.

`EXTERNAL` *N* `:` *string* `,..,` *N* `:` *string*

The "`:` *string*"s may be omitted.

`MANIFEST` *M* `=` *K* `,..,` *M* `=` *K*

The "`=` *K*"s are optional. When omitted the next available integer is used.

`STATIC` *N* `=` *K* `,..,` *N* `=` *K*

The "`=` *K*"s are optional, and when omitted the corresponding variable is not initialised. The *K*s may include strings, tables and functions.

`GLOBAL` *N* `:` *K* `,..,` *N* `:` *K*

The "`:` *K*"s may be omitted, and when omitted the next available integer is used.

A.2 Expressions

N Eg: `abc v1 a s_err`

These are used for variable and function names. They start with lower case letters.

M Eg: `Abc B1 A S_for`

These are used for manifest constant names. They start with upper case letters.

inumb Eg: `1234 #x7F_0001 #377 #b_0111_1111_0000`

`?`

This yields an undefined value.

`TRUE` `FALSE`

These are constants equal to `-1` and `0`, respectively.

char Eg: `'A' '\n' 'XYZ'`

The characters are packed into a word as consecutive bytes. The rightmost character being placed in the least significant byte position. Such constants can be thought of as base 256 integers.

string Eg: `"abc" "Hello\n"`

Strings are zero terminated for compatibility with C.

`TABLE [` *E* `,..,` *E* `]`

This yields an initialised static vector. The elements of the vector are not necessarily re-initialised on each evaluation of the table, particularly if the initial values are constants.

`[` *E* `,..,` *E* `]`

This yields an initialised local vector. The space allocated in current scope.

`VEC` *K*

This yields an uninitialised local word vector with subscripts from 0 to *K*. The space is allocated on entry to the current scope.

`CVEC` *K*

This yields an uninitialised local byte vector with subscripts from 0 to *K*. The space is allocated on entry to the current scope.

`(` *E* `)`

Parentheses are used to group an expression that normally yields a result.

`{` *E* `}`

Braces are used to group an expression that normally has no result.

EA

This is a function call. To avoid syntactic ambiguity, *A* must be a name (*N* or *M*), a constant (*inumb*, `?`, `TRUE`, `FALSE`, *char* or *string*), or it must start with `(` or `[`.

`@` *E*

This returns the address of *E*. *E* must be either a variable name (*N*) or a subscripted expression (*E*`!`*E*, *E*`%`*E*, `!`*E* or `%`*E*).

E `!` *E* `!` *E*

This is the word subscription operator. The left operand is a pointer to the zeroth element of a word vector and the right hand operand is an integer subscript. The form `!`*E* is equivalent to *E*`!0`.

E `%` *E* `%` *E*

This is the byte subscription operator. The left operand is a pointer to the zeroth element of a byte vector and the right hand operand is an integer subscript. The form `%`*E* is equivalent to *E*`%0`.

`++` *E* `+++` *E* `--` *E* `---` *E*

Pre increment or decrement by 1 or `Bpw` (bytes per word).

E `++` *E* `+++` *E* `--` *E* `---`

Post increment or decrement by 1 or `Bpw`.

`~` *E* `+` *E* `-` *E* `ABS` *E*

These are monadic operators for bitwise NOT, plus, minus and absolute value, respectively.

E `<<` *E* *E* `>>` *E*

These are logical left and right shift operators, respectively.

E `*` *E* *E* `/` *E* *E* `MOD` *E* *E* `&` *E*

These are dyadic operators for multplication, division, remainder after division, and bitwise AND, respectively.

E `+` *E* *E* `-` *E* *E* `|` *E*

These are dyadic operators for addition, subtraction, and bitwise OR, respectively.

E `XOR` *X*

This returns the bitwise exclusive OR of its operands.

E relop E relop . . . *E*

where *relop* is any of `=`, ∼`=`, `<`, `<=`, `>` or `>=`. It return `TRUE` only if all the individual relations are satisfied. Each *E* is evaluated atmost once.

`NOT` *E* *E* `AND` *E* *E* `OR` *E*

These are the truth value operators.

E `->` *E*, *E*

This is the conditional expression construct.

E , . . , *E* `:=` *E* , . . , *E* *E* , . . , *E* `ALL:=` *E*

This is the simultaneous assignment operator. All the expressions are evaluated then all the assignments done.

E `,..,` *E op*`:=` *E* `,..,` *E*

Where *op*`:=` can be any of the following: `>>:=`, `<<:=`, `*:=`, `/:=`, `MOD:=`, `&:=`, `+:=`, `-:=`, or `XOR:=`.

`RAISE` *A*

This transfers control to the the currently active HANDLE. Up to three arguments can be passed.

`TEST` *E* `THEN` *E* `ELSE` *E*

`IF` *E* `DO` *E*

`UNLESS` *E* `DO` *E*

These are the conditional commands. They are less binding than assignment and typically do not yield results.

`WHILE` *E* `DO` *E*

`UNTIL` *E* `DO` *E*

E `REPEATWHILE` *E*

E `REPEATUNTIL` *E*

E `REPEAT`

`FOR` *N* `=` *E* `TO` *E* `BY` *K* `DO` *E*

`FOR` *N* `=` *E* `TO` *E* `DO` *E*

`FOR` *N* `=` *E* `BY` *K* `DO` *E*

`FOR` *N* `=` *E* `DO` *E*

These are the repetitive commands. The FOR command introduces a new scope for locals, and *N* is a new variable within this scope.

`VALOF` *E*

This introduces a new scope for locals and defines the context for RESULT commands within *E*.

`MATCH` *A* `:` *P* `=>` *E* `:..:` *P* `=>` *E* `.`

`EVERY` *A* `:` *P* `=>` *E* `:..:` *P* `=>` *E* `.`

E `HANDLE :` *P* `=>` *E* `:..:` *P* `=>` *E* `.`

In each of these construct, the dot `(.)` is optional. The arguments (*A*) are matched against the patterns (*P*), and control passed to the first expression whose patterns match. For the EVERY construct, all guarded expressions whose patterns match are evaluated. The HANDLE construct defines the context for RAISE commands. A RAISE command will supply the arguments to be matched by HANDLE.

`RESULT` *E* `RESULT`

Return from current `VALOF` expression with a value. `RESULT` with no argument is equivalent to `RESULT ?`.

`EXIT` E `EXIT`

Return from the current function or MATCH, EVERY or HANDLE construct with a given value. `EXIT` with no argument is equivalent to `EXIT ?`.

`RETURN` E `RETURN`

Return from current function with a value. `RETURN` with no argument is equivalent to `RETURN ?`.

`BREAK` `LOOP`

Respectively, exit from, or loop in the current repetitive expression.

E `;..;` E

Evaluate expressions from left to right. The result is the value of the last expression. Any semicolon at the end of a line is optional.

`LET` N `=` E `,..,` N `=` E

This construct declares and possibly initialises some new local variables. The allocation of space for them is done on entry to the current scope. New local scopes are introduced by `FUN`, `MATCH`, `EVERY`, `HANDLE`, `=>`, `VALOF`, and `FOR`. The "`=`E"s are optional, but, if present, cause the corresponding variable to be initialised when the LET contruct is reached.

A.3 Constant expressions

These are used in MANIFEST, STATIC and GLOBAL declarations, in VEC expressions, in the step length of FOR commands, and in patterns.

The syntax of constant expressions is the same as that of ordinary expressions except that only constructs that can be evaluated at compile time are permitted. These are:

M, *inumb*, `?`, `TRUE`, `FALSE`, *char*,
`(` K `)`, `{` K `}`,
`~` K, `+` K, `-` K, `ABS` K,
K `<<` K, K `>>` K,
K `*` K, K `/` K, K `MOD` K, K `&` K,
K `+` K, K `-` K, K `|` K,
K `XOR` K,
K *relop* K *relop* `...` K,
`NOT` K, K `AND` K, K `OR` K,
K `->` K, K
`TEST` K `THEN` K `ELSE` K

A.4 Patterns

Patterns are used in function definitions, MATCH, EVERY and HANDLE constructs. Patterns are matched against parameter values held in consecutive storage locations. Pattern matching is applied from left to right, except that any assignments are done at the end and only if the entire match was successful.

N

The current location is given name N.

`?`

This will always match the current location.

K

The value in the current location must equal K.

K`..`K

The value in the current location must greater than or equal to the first K and less than or equal to the second K.

`(` P `)`

Parentheses are used for grouping.

P `,..,` P

The current location and adjacent ones are matched by the corresponding Ps.

`[` P `,..,` P `]`

The value of the current location is a pointer to consective locations matched by the Ps.

PP

The value in the current location is matched by both Ps.

P `OR` P

One or other pattern must match. The patterns must only be constants (K) or ranges (K`..`K).

`<` E `<=` E `>` E `>=` E `=` E `~=` E

The value of the current location must be `<`E, `<=`E, etc.

`:=` E

If the entire match is successful, the current location is updated with the value of E.

op`:=` E

If the entire match is successful, the current location is modified by the specified operation with E.

A.5 Arguments

Arguments are used in function calls and in MATCH, EVERY, GOTO and RAISE commands. They cause a number of expressions to be evaluated and placed in consecutive locations ready to be matched by one or more patterns.

E
`( )`
`(` *E* , . . . , *E* `)`

An argument list is either an expression, or a, possibly empty, list of expressions separated by commas and enclosed in parentheses. The argument in a function call must start with `(` or `[`, or be a name, a constant, `?`, `TRUE` or `FALSE`.

References

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[McM93] K.L McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.

[Ric97] M. Richards. *MCPL Programming Manual.* Technical Report No 431, Cambridge University Computer Laboratory, July 1997.

[SBM96] S. Jha S. Berezin, E. Clarke and W. Marrero. Model checking algorithms for the μ-calculus. Technical report, Carnegie Mellon University, September 1996. CMU-CS-96-180.

[SV97] D.J. Stewart and M. VanInwegen. Private communication. Cambridge University Computer Laboratory, 1997.