```
>> Let a[1] := x /\ y;
```

```
>> Let a[2] := a[1] /\ z;
```

Following are some of the statements available in TLV-BASIC:

- `Let` *var* `:=` *exp* — Assign the value of expression *exp* to variable *var*.
- `Proc` *proc-name* $(par_1, \ldots, par_n)$; S `End` — Define a procedure *proc-name* with parameters $par_1, \ldots, par_n$ and body S. Paramteres are transferred by value.
- `While (`*exp*`)` S `End` — Repeatedly execute statement S until *exp* becomes 0.
- `If (`*exp*`)` S_1 $\left[\texttt{ else } S_2\right]$ `End` — If *exp* evaluates to a non-zero value, execute statement S_1. Otherwise, execute statement S_2. The `else` clause is optional.
- `Call` *proc-name* $(par_1, \ldots, par_n)$ — Invoke procedure *proc-name* with the given actual parameters.

The last two statements are the main commands that are used in an interactive mode.

```
Let n := 3;

To prepare;
  Let mux := TRUE;
  Let i := n;
  While (i)
    Let j := i - 1;
    While (j)
      Let mux := mux & !(proc[i].loc = 3 & proc[j].loc = 3);
      Let j := j - 1;
    End -- end loop on j
    Let i := i - 1;
  End -- end loop on i
End -- end procedure

Run prepare
```

Figure 2.1: File `mux-sem.pf`: Proof Script of mutual exclusion for general n.

```
-- Procedure which checks if the paramter p_ is an invariant.
Proc binv(p_);

  -- Check B1.

  -- Find counter example for first premise of binv.
  Let counter_example := ! ( _i -> p_ );

  -- If the obdd of counter_example is anything but the 0 obdd leaf
  -- then we found a counter example.
  If ( counter_example )
    Print "binv FAILED in premise B1","\n",counter_example ;
    Return ;
  Else
    Print "B1 PASSED","\n" ;
  End


  -- Check B2.

  -- Assign # of transitions in the system to an index variable k_.
  Let k_ := _tn;

  -- Loop which executes until k_ = 0.
  While(k_)

    -- Find counter example for B2 and the current transition.
    Let counter_example := _trans[k_] & p & !next(p);



    -- If the obdd of counter_example is anything but the 0 obdd leaf
    -- then we found a counter example.
    If ( counter_example )
      Print "binv FAILED in premise B2 for transition ",k_,
            "\n",counter_example ;
      Return ;
    End

    Let k_ := k_ - 1;
  End

  Print "B2 PASSED","\n" ;
End
```

- _tn — The total number of transitions.

Transition i ($1 \leq i \leq$ _tn) resides in three arrays:

- _t[i] — The sequential component. Defines the value of the state variables in the next state as a function of the state variable and combinational variables of the current state.
- _d[i] — The combinational component. Defines the value of the combinational variables as a function of the state variables.
- _pres[i] — Preserve all other variables which are not assigned to in this transition.

The actual transition is `_t[i] & _d[i] & _pres[i]`. They are kept separate for performance considerations.

- _i — The total initial condition
- _j[i] — Array of just conditions
- _jn — The number of items in array _j
- _cp[i] — Array of compassionate subcondition
- _cq[i] — Array of compassionate subcondition
- _cn — The number of items in arrays _cp,_cq

All arrays start from index 1.
Usually there is only one system in a single SMV file. The following variables are relevant when there is more than one system:

- _sn — The number of systems in the file
- _tn[i] — The number of transitions in system i.
- _i[i] — The initial condition of system i.
- _jn[i] — The number of justice conditions in system i.
- _cn[i] — The number of compassion conditions in system i.
- _id[i] — All variables in system i are equal to their primed versions.
- _vars[i] — All variables in system i (unprimed).

```
k : 0..3 kind of xx;
```

Then for all systems, an array item with the set of variables of this kind will be created, even if that system has no variables of that kind. For example, if the file containing the decaration above has three systems then array items xx[1], xx[2] and xx[3] will be formed, and one of them will contain the variable k defined above.
In addition, the following "constants" are also supplied:

- _id — All variables are equal to their primed version.
- _vars — All variables (unprimed).
- _def — All variables which are defined variables.

Logical operators

operator	TLV
$\wedge$	&, /\
$\vee$	| , \/
$\neg$	!
$\rightarrow$	->
$\leftrightarrow$	<->

The bdds which are a result of the expressions cannot have leaves with a value which differs fr
0/1.

Numeric operators

Comparitive : = != < > <= >=
Computational : + - * / mod

Value sets

These expressions are used to specify sets of values which a variable may be equal to.

The set constant has the following form:

```
{ val1, val2, ... , valn }
```

Set operators

- expr1 in expr2

  This is the inclusion operator which checks for membership in a set expr2.

- expr1 notin expr2

  The negation of the "in" operator.

- expr1 union expr2

  Returns the union of two value sets. If either argument is a number of a symbolic value instead of a set it is coerced to a singleton set.

Variable sets

Variable sets are used mainly for quantification over sets of variables, by the following functions:

- expr1 forsome expr2
  expr1 forall expr2

  Returns forsome and forall functions where expr2 contains the variables to be quantified. expr2 should be a variable set.

The predefined variable "_vars" is a bdd which represents the set of all program variables. The following functions manipulate sets of variables.

A common pitfall arises when one wants to ask whether an expression is a contradication or empty set (i.e. it is the OBDD leaf 0). A naive attempt to ask this can be done as:

```
# Wrong
If ( k = 0 )
  ...
End
```

Note that the value of the expression `k = 0` will be true if the bdd representing k has some path leading to a zero leaf. But what we want is an expression which will be true if all paths lead to the zero leaf.

This can be done by using the "value" function. "value" always returns some leaf of its bdd argument. It tries to return a non-zero leaf, so it will only return 0 if its parameter is 0. So the right way to ask whether k is a contradicition is:

```
If ( value(k) = 0 )
  ...
End
```

Alternatively you could add an else statement and move the desired code there. For example:

```
If ( k )
  # Some code which does absolutely nothing
  Let k := k;
Else
  # The code you want to perform in case k is a contradiction.
  ...
End
```