2. A Programming Notation (ch. 2 & 6)

Special Notation

- Quantified expressions assume the form

```
〈 operator  variables  :  range  ::  expression 〉
```

  - operator is generally associative and commutative (plus, minus, min, max, and, or, set, sequence, etc.)
  - if the range is empty the result is the unit element for that operator

```
〈 + k : 1≤k≤N :: X(k) 〉

        X(1)   X(2)   …   X(N-1)    X(N)
        X(1) + X(2) + … + X(N-1) +  X(N)

〈 Σ k : 1≤k≤N :: X(k) 〉

〈 ∧ k : 1≤k≤N :: X(k)=0 〉
〈 ∀ k : 1≤k≤N :: X(k)=0 〉

〈 ∨ k : 1≤k≤N :: X(k)=0 〉
〈 ∃ k : 1≤k≤N :: X(k)=0 〉

〈 ∪ k : 1≤k≤N :: {X(k)} 〉
〈 set k : 1≤k≤N :: X(k) 〉

〈 seq k : 1≤k≤N :: X(k) 〉
```

  - [] can be used to create lists of definitions

    `〈 [] k : 1≤k<N :: sorted(k) = X(k) ≤ X(k+1) 〉`

  - [] can be used to create sets of statements

    `〈 [] k : 1≤k<N :: X(k),X(k+1) := X(k+1),X(k) if ¬sorted(k) 〉`

  - || can be used to construct complex single statements

    `〈 || k : 1≤k<N :: X(k) := X(k+1) || X(N) := 0 〉`

Program structure

```
Program program name
    declare   variable declarations
    always    macro definitions
    initially initial values
    assign    assignment statements
end
```

- Example: matrix transposition

Declare

- Pascal-like declarations
- Typical types: integer, boolean, array, set, sequence

```
declare
    N : integer
[]  M : array[1..N, 1..N] of integer
[]  B : array[1..N, 1..N] of boolean
```

Always

- It defines transparent variables
  - macro-like definitions (referential transparency)
  - simplify reasoning (act as invariants)
  - are efficient to implement
- The definition is given in terms of a set of equations
- The definition must be non-circular under some ordering of the equations

```
always
    〈 [] i : 1≤i≤N :: rok(i) = 〈 ∧ k : 1≤k≤N :: B[i,k] 〉 〉
```

- Note that:
  - the [] acts as a separator and allows for reordering
  - transparent variables must appear once and only once on the left
  - i and k are quantified variables, not part of the program state and not available outside the quantification scope

Initially

- It defines the initial values of the variables
- It follows the same form and rules as the always section

```
initially
    N = 10
[]  〈 [] i,j : 1≤i≤N ∧ 1≤j≤N :: B[i,j] = false 〉
[]  〈 [] i,j : 1≤i≤N ∧ 1≤j≤N :: M[i,j] = f(i,j) 〉
```

- Note:
  - the absence of input/output
  - N and M do not need to be initialized
  - it is reasonable to state that N>1

```
initially
    N > 1
[]  B = false
```

Assign

- Set of conditional multiple-assignment statements
- The set is fixed
- All statements are deterministic
- Weak fairness — each statement is executed infinitely often

```
assign
    i,j,B[1,1] := 1,1,true  if ¬B[1,1]
[]  i,j := i+1,1            if rok(i) ∧ i<N
[]  j := j+1                if B[i,j] ∧ j<N
[]  M[i,j],M[j,i],B[i,j],B[j,i]
        := M[j,i],M[i,j],true,true if ¬B[i,j]
```

  - Note that i and j may start with any legal values and should have be declared

```
declare
    i,j : integer
```

- What if we were to use …

```
    i := i+1  []  i := i-1  []  j := j+1  []  j := j-1

        no fixpoint;
        no sure coverage of M – some (i,j) may not be generated

    i := i+1  ||  i := i-1  ||  j := j+1  ||  j := j-1

        illegal; || creates a single statement!

    i,j := i+1,j  if ¬B[i+1,j]  ~  i-1,j  if ¬B[i-1,j] ... etc.

        nondeterministic; no initial values; may get stuck

    i,j := i+1,j  if ¬B[i+1,j]  []  i-1,j  if ¬B[i-1,j] ... etc.

        deterministic; no initial values; may get stuck
```

- Enumerated assignments — building a set of statements

```
〈 [] i,j : 1≤i,j≤N ::
    M[i,j],M[j,i],B[i,j],B[j,i]
        := M[j,i],M[i,j],true,true  if ¬B[i,j] 〉
```

- Quantified assignments — building a large statement

```
〈 || i,j : 1≤i,j≤N ::
    M[i,j],M[j,i],B[i,j],B[j,i]
        := M[j,i],M[i,j],true,true if ¬B[i,j] 〉
or
    〈 || i,j : 1≤i,j≤N ::
        M[i,j],M[j,i],B := M[j,i],M[i,j],true if ¬B 〉
```

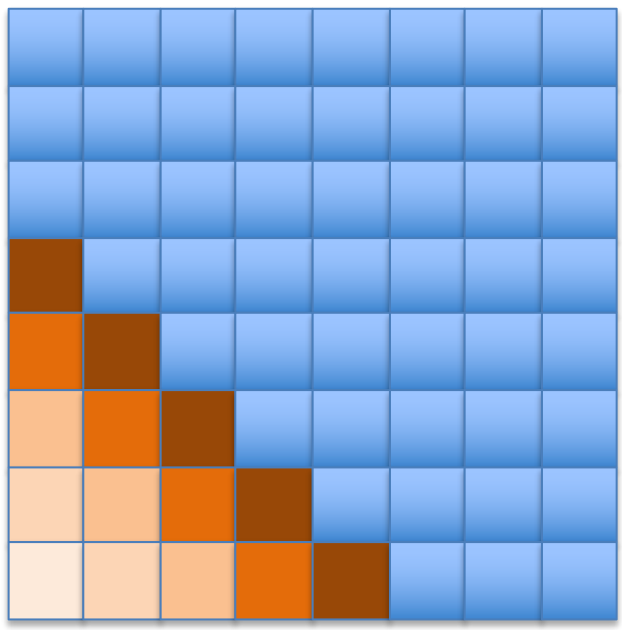
- Note:
  - box versus parallel bar — building multiple statements versus a single statement
  - box cannot be used under the scope of a parallel bar

- What if initially k=1+1 and we have

```
        k := k+1 if k<N+N
    ∥
        ⟨ ∥ i,j : 1≤i≤N ∧ 1≤j≤N ∧ i+j=k::
                M[i,j],M[j,i] := M[j,i],M[i,j] ⟩

        ok; fixed number of statements, each of variable size;
        replacing the ∥ is not possible
```

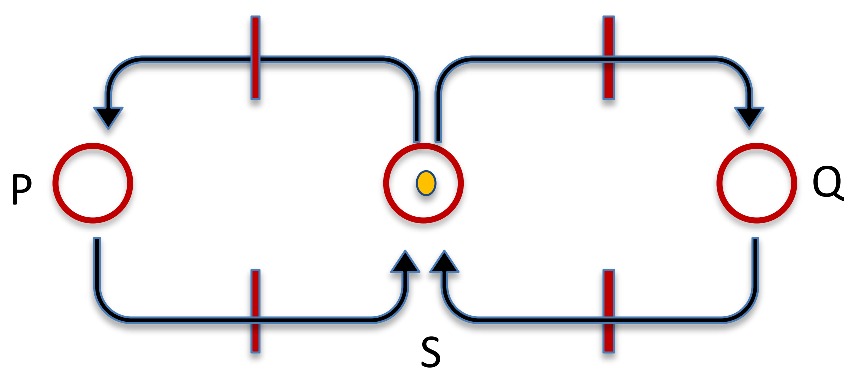
- Whole diagonals are processed one at a time in order



Sample programs

Petri Net simulation

- Mutual exclusion logic



```
Program  Mutex
   declare   p,q,s : boolean
   initially p,q,s = 0,0,1
   assign
      p,s := 1,0  if s = 1
   [] q,s := 1,0  if s = 1
   [] p,s := 0,1  if p = 1
   [] q,s := 0,1  if q = 1
end
```

Comparing two ascending sequences

- Given two ascending sequences of numbers, determine if they represent the same set
- Assumptions

```
f[0] = g[0]
f[N] = g[N]
f[N] > f[N-1]
g[N] > g[N-1]
⟨ ∀ i : 0≤i<N :: f[i] ≤ f[i+1] ⟩
⟨ ∀ i : 0≤i<N :: g[i] ≤ g[i+1] ⟩

Program  Compare
   declare   u,v : integer
   initially u,v = 0,0
   assign
      u := u+1  if u<N ∧ f[u]=f[u+1]
   [] v := v+1  if v<N ∧ g[v]=g[v+1]
   [] u,v := u+1,v+1  if v<N ∧ u<N ∧ f[u+1]=g[v+1]
end
```

- Key invariant property

```
0≤u≤N ∧ 0≤v≤N ∧
⟨ set i : 0≤i≤u :: f[i] ⟩ = ⟨ set i : 0≤i≤v :: g[i] ⟩
```

Maximum of a set

- Find the largest integer in a set represented as an array A[0..N-1]
- Assumptions

```
N = 2 * M

Program  Maximum
   assign
   ⟨ ‖ i : 0≤i<M :: A[i] := max(A[2*i], A[2*i+1]) ⟩
end
```

- *O(log N)* steps on a synchronous machine
- The basic idea is to create a tree-like computation

0	1	2	3	4	5	6	7	8	9
0/1	2/3	4/5	6/7	8/9					
0/1/2/3	4/5/6/7	8/9/...							
0/1/2/3/4/5/6/7	8/9/...								
0/1/2/3/4/5/6/7/8/9/...									

Saddle point of a matrix

- The problem requires one to detect only the existence of a saddle point (not its location)
- The solution is in the style of the equational programming paradigm
- A[u,v] is a saddle point if it has
  - the lowest value along the column — denoted by X[v]
  - the largest value along the row — denoted by Y[u]
- In general, any matrix element satisfies the property X[v]≤A[u,v]≤Y[u]
  i.e., higher than the min on column, and lower than the max on the row
- For saddle point we also have X[v]≥A[u,v]≥Y[u]
  i.e., equality X[v]=A[u,v]=Y[u]
- If this is true for A[u,v] it follows that

```
⟨ max v :: X[v] ⟩ ≥ X[v]≥A[u,v]≥Y[u] ≥ ⟨ min u :: Y[u] ⟩

Program  Saddle-Point

   declare   X,Y array[0..N-1] of integer

   always
         ⟨ [] v :: X[v] = ⟨ min i :: A[i,v] ⟩
      [] ⟨ [] u :: Y[u] = ⟨ max j :: A[u,j] ⟩
      [] sp = ( ⟨ max v :: X[v] ⟩ ≥ ⟨ min u :: Y[u] ⟩ )
end
```