Theory of concurrency

Lecture 11

Communication

Communication: Pipes

- *Pipes* are the processes with only two channels in their alphabet
  - an input channel *left* and an output channel *right*.

- The connection of pipes P and Q is
  - $P \gg Q$
  - The right channel of P is connected to the left channel of Q,
  - sequence of messages output by P and input by Q on this internal channel is
    - *concealed* from their common environment.
  - The connecting channel does not have a name due to concealment.
  - All messages input on the left channel of $(P \gg Q)$ are input by P
  - All messages output on the right channel of $(P \gg Q)$ are output by Q.

Communication: **Pipes**

- $(P \gg Q)$ is itself a pipe, and may again be placed in series with other pipes:
  - $(P \gg Q) \gg R$, $(P \gg Q) \gg (R \gg S)$, etc.
- $\gg$ is associative, we shall omit brackets in such a series.
- $\gg$ is possible if
  - $a(P \gg Q) = aleft(P) \cup aright(Q)$
  - $aright(P) = aleft(Q)$
    - and a further constraint states that the connected channels are capable of transmitting the same kind of message

Communication: **Pipes**

$$DOUBLE = \mu\, X \bullet (left\ ?\ x \rightarrow right\ !\ (x + x) \rightarrow X)$$

X1. A pipe outputs each input value multiplied by four

$$QUADRUPLE = DOUBLE \gg DOUBLE$$

X2. A process inputs cards of eighty characters and outputs their text, tightly packed into lines of 125 characters each:

$$UNPACK \gg PACK$$

$$PACK = P_{<>}$$
$$P_s = right\ !\ s \rightarrow P_{<>} \quad \text{if } \#s = 125$$
$$P_s = left\ ?\ x \rightarrow P_{s^\wedge<x>} \quad \text{if } \#s < 125$$

$$UNPACK = P_{<>}$$
$$P_{<>} = left\ ?\ s \rightarrow P_s$$
$$P_{<x>} = right\ !\ x \rightarrow P_{<>}$$
$$P_{<x>^\wedge s} = right\ !\ x \rightarrow P_s$$

- The problem of *structure clash* (Michael Jackson).
  - it is not clear whether the major loop should iterate
    - once per input card, or once per output line.
  - Quite difficult to write using conventional programming techniques.
- This solution contains a *separate loop* in each of the two processes
  - matches the structure of the original problem.

Communication: **Pipes**

$$SQUASH = \mu X \bullet left\ ?\ x \rightarrow \textbf{if}\ x \neq \text{“*”}\ \textbf{then}\ (right\ !\ x \rightarrow X)\ \textbf{else}\ left\ ?\ y \rightarrow \textbf{if}\ y = \text{“*”}\ \textbf{then}\ (right\ !\ \text{“↑”} \rightarrow X)\ \textbf{else}\ (right\ !\ \text{“*”} \rightarrow right\ !\ y \rightarrow X))$$

X3. Same as X2, except that each pair of consecutive asterisks is replaced by “↑”:

$$UNPACK \gg SQUASH \gg PACK$$

Modularity

- In a conventional sequential program,
  - this minor change in specification could cause some problems.
- In CSP, such problem is avoided by simple inserting an additional process.
- This kind of modularity has been introduced and exploited by
  - the designers of operating systems.

Communication: **Pipes**

$$PACK = P_{<>}$$
$$P_s = right\,!\,s \rightarrow P_{<>} \quad \text{if } \#s = 125$$
$$P_s = left\,?\,x \rightarrow P_{s\hat{}<x>} \quad \text{if } \#s < 125$$

$$BUFFER = P_{<>}$$
$$P_{<>} = left\,?\,x \rightarrow P_{<x>}$$
$$P_{<x>\hat{}s} = (left\,?\,y \rightarrow P_{<x>\hat{}s\hat{}<y>} \mid right\,!\,x \rightarrow P_s)$$

X4 Same as X2, except that the reading of cards may continue when the printer is held up, and the printing can continue when the card reader is held up:

$$UNPACK = P_{<>}$$
$$P_{<>} = left\,?\,s \rightarrow P_s$$
$$P_{<x>} = right\,!\,x \rightarrow P_{<>}$$
$$P_{<x>\hat{}s} = right\,!\,x \rightarrow P_s$$

$$UNPACK \gg BUFFER \gg PACK$$

- The buffer
  - holds characters produced by the *UNPACK*, but not yet consumed by the *PACK*.
  - smoothes out *temporary* variations in the rate of production and consumption.
  - does not solve the problem of global mismatch between the rates of production and consumption.
    - If the card reader is on average slower than the printer,
      - the buffer is nearly always empty, and no smoothing effect is achieved.
    - If the reader is faster,
      - the buffer expands indefinitely, until it consumes all available storage space.

$COPY = \mu\, X \bullet (left\ ?\ x \rightarrow right\ !\ x \rightarrow X)$

X4: $UNPACK \gg BUFFER \gg PACK$

Communication: Pipes

X5 In order to avoid undesirable expansion of buffers,
- it is usual to limit the number of messages buffered.

- The *COPY* process is the single buffer.
- A version of X4 reads one card ahead on input and buffers one line on output:

$$COPY \gg UNPACK \gg PACK \gg COPY$$

- The alphabets of the two instances of *COPY* are different due to
  - the context in which they are placed.

X6. A double buffer accepts up to two messages before requiring output of the first:

$$COPY \gg COPY$$

- Its behaviour is similar to that of *VMS2*.

$VMS2 = (coin \rightarrow \mu X \bullet (coin \rightarrow choc \rightarrow X \mid choc \rightarrow coin \rightarrow X))$

Communication: Pipes: **Laws**

- Associativity:

L1. $P \gg (Q \gg R) = (P \gg Q) \gg R$

- The laws for connecting input and output in a pipe
  - Simplification of process descriptions by a form of symbolic execution.
    - The process on the left of $\gg$ starts with output of a message v to the right,
    - the process on the right of $\gg$ starts with input from the left,
    - the message v is transmitted from the former process to the latter;
    - the actual communication is concealed:

L2. $(right\,!\,v \rightarrow P) \gg (left\,?\,y \rightarrow Q(y)) = P \gg Q(v)$

Communication: Pipes: **Laws**

- One of the processes is determined to communicate with the other,
- the other is prepared to communicate externally,
  - the external communication takes place first, and
  - the internal communication is saved up for a subsequent occasion

L3. $(right\ !\ v \rightarrow P) \gg (right\ !\ w \rightarrow Q) = right\ !\ w \rightarrow ((right\ !\ v \rightarrow P) \gg Q)$

L4. $(left\ ?\ x \rightarrow P(x)) \gg (left\ ?\ y \rightarrow Q(y)) = left\ ?\ x \rightarrow (P(x) \gg (left\ ?\ y \rightarrow Q(y)))$

- If both processes are prepared for external communication, then
  - either may happen first

L5. $(left\ ?\ x \rightarrow P(x)) \gg (right\ !\ w \rightarrow Q) = (left\ ?\ x \rightarrow (P(x) \gg (right\ !\ w \rightarrow Q))$

$$\mid right\ !\ w \rightarrow ((left\ ?\ x \rightarrow P(x)) \gg Q))$$

Communication: Pipes: **Laws**

- The modification of L5 with replacing $\gg$ by $\gg R \gg$
  - pipes in the middle of a chain cannot communicate directly with the environment

L6. $(left\ ?\ x \to P(x)) \gg R \gg (right\ !\ w \to Q) = (left\ ?\ x \to (P(x) \gg R \gg (right\ !\ w \to Q))$

$$\mid right\ !\ w \to ((left\ ?\ x \to P(x)) \gg R \gg Q))$$

- Similar generalisations:

L7. If R is a chain of processes all starting with output to the right,

$$R \gg (right\ !\ w \to Q) = right\ !\ w \to (R \gg Q)$$

L8. If R is a chain of processes all starting with input from the left,

$$(left\ ?\ x \to P(x)) \gg R = left\ ?\ x \to (P(x) \gg R)$$

Communication: Pipes: **Laws**

X1. Let us define $R(y) = (right\,!\,y \rightarrow COPY) \gg COPY$

- So

$$\begin{aligned} R(y) &= (right\,!\,y \rightarrow COPY) \gg (left\,?\,x \rightarrow right\,!\,x \rightarrow COPY) && [\text{def } COPY] \\ &= COPY \gg (right\,!\,y \rightarrow COPY) && [\text{L2}] \end{aligned}$$

X2.

$$\begin{aligned} & COPY \gg COPY \\ &= (left\,?\,x \rightarrow right\,!\,x \rightarrow COPY) \gg COPY && [\text{def } COPY] \\ &= left\,?\,x \rightarrow ((right\,!\,x \rightarrow COPY) \gg COPY) && [\text{L4}] \\ &= left\,?\,x \rightarrow R(x) && [\text{def } R(x)] \end{aligned}$$

L2. $(right\,!\,v \rightarrow P) \gg (left\,?\,y \rightarrow Q(y)) = P \gg Q(v)$

L4. $(left\,?\,x \rightarrow P(x)) \gg (left\,?\,y \rightarrow Q(y)) = left\,?\,x \rightarrow (P(x) \gg (left\,?\,y \rightarrow Q(y)))$

$$R(y) = (right\,!\,y \to COPY) \gg (left\,?\,x \to right\,!\,x \to COPY)$$
$$= COPY \gg (right\,!\,y \to COPY)$$

Communication: Pipes: **Laws**

X3. From the last line of X1 we deduce

$$R(y) = (left\,?\,x \to right\,!\,x \to COPY) \gg (right\,!\,y \to COPY)$$
$$= (left\,?\,x \to ((right\,!\,x \to COPY) \gg (right\,!\,y \to COPY)) \quad \text{[L5]}$$
$$\mid right\,!\,y \to (COPY \gg COPY))$$
$$= (left\,?\,x \to right\,!\,y \to R(x) \mid right\,!\,y \to left\,?\,x \to R(x)) \quad \text{[L3, X2]}$$

X1-X3.

- A double buffer, after input of its first message, is prepared either
  - to output that message or
  - to input a second message before doing so.

$$R(y) = (right\,!\,y \to COPY) \gg COPY$$

L5. $(left\,?\,x \to P(x)) \gg (right\,!\,w \to Q) = (left\,?\,x \to (P(x) \gg (right\,!\,w \to Q)) \mid right\,!\,w \to ((left\,?\,x \to P(x)) \gg Q))$

L3. $(right\,!\,v \to P) \gg (right\,!\,w \to Q) = right\,!\,w \to ((right\,!\,v \to P) \gg Q)$

Communication: Pipes: Implementation Promela

- An event

```
chan eP = [0] of type
chan eQ = [0] of type
chan PQ = [0] of type

proctype P {
 type x;
  ...
  if
     :: eP ? x -> P(x); PQ ! v; P'
     :: else -> PQ ! v; P'
  fi
  ...
}
```

L2. $(right\ !\ v \rightarrow P) \gg (left\ ?\ y \rightarrow Q(y)) = P \gg Q(v)$
L3. $(right\ !\ v \rightarrow P) \gg (right\ !\ w \rightarrow Q) = right\ !\ w \rightarrow ((right\ !\ v \rightarrow P) \gg Q)$
L4. $(left\ ?\ x \rightarrow P(x)) \gg (left\ ?\ y \rightarrow Q(y)) = left\ ?\ x \rightarrow (P(x) \gg (left\ ?\ y \rightarrow Q(y)))$

Communication: Pipes: **Livelock**

- No deadlock
  - If both P and Q are nonstopping, then $(P \gg Q)$ will not stop either.
    - The chaining operator connects two processes by just one channel.
- Divergence
  - $P = (right\ !\ 1 \rightarrow P)$ and $Q = (left\ ?\ x \rightarrow Q)$
  - the processes P and Q spend the whole time communicating with each other
  - $(P \gg Q)$
    - never communicates with the external world,
    - a useless process,
    - may consume unbounded computing resources without achieving anything.

Communication: Pipes: **Livelock**

- $P = (right\,!\,1 \rightarrow P \mid left\,?\,x \rightarrow P1(x))$ and $Q = (left\,?\,x \rightarrow Q \mid right\,!\,1 \rightarrow Q1)$
- In $(P \gg Q)$, divergence derives from the possibility of infinite internal communication
  - it exists even though
    - the choice of external communication is offered on every possible occasion,
    - after such an external communication the subsequent behaviour of $(P \gg Q)$ does not diverge.
- $(P \gg Q)$ is *free of livelock* if
  - P is *left-guarded*
    - it can never output an infinite series of messages to the right without
      - interspersing inputs from the left.
    - the length of the sequence output to the right is always bounded above by some well-defined function f of the sequence of values input from the left:
      - P is left-guarded $\equiv \exists\, f \bullet P$ sat $(\#right \leq f(left))$
  - Left-guardedness is often obvious from the text of P.

Communication: Pipes: **Livelock**

$PACK = P_{<>}$
$P_s = right\ !\ s \rightarrow P_{<>}$ if $\#s = 125$
$P_s = left\ ?\ x \rightarrow P_{s^\wedge<x>}$ if $\#s < 125$

L1. If every recursion used in the definition of P is guarded by an input from the left, **then** P is left-guarded.

L2. If P is left-guarded **then** $(P \gg Q)$ is free of livelock.

- Right-guardedness of the second operand of $\gg$

L3. If Q is right-guarded **then** $(P \gg Q)$ is free of livelock.

$UNPACK = P_{<>}$
$P_{<>} = left\ ?\ s \rightarrow P_s$
$P_{<x>} = right\ !\ x \rightarrow P_{<>}$
$P_{<x>^\wedge s} = right\ !\ x \rightarrow P_s$

X1. The following are left-guarded by L1

- *COPY, DOUBLE, SQUASH, BUFFER*

$COPY = \mu\ X \bullet (left\ ?\ x \rightarrow right\ !\ x \rightarrow X)$

$DOUBLE = \mu\ X \bullet (left\ ?\ x \rightarrow right\ !\ (x + x) \rightarrow X)$

X2. These processes are left-guarded in accordance with the original definition, because

- *UNPACK* sat $\#right \leq \#(/\ left)$
- *PACK* sat $\#right \leq \#left$

$BUFFER = P_{<>}$
$P_{<>} = left\ ?\ x \rightarrow P_{<x>}$
$P_{<x>^\wedge s} = (left\ ?\ y \rightarrow P_{<x>^\wedge s^\wedge <y>}\ |\ right\ !\ x \rightarrow P_s)$

X3. *BUFFER* is *not* right-guarded, since

- it can input arbitrarily many messages from the left without
  - ever inputting from the right.

Communication: Pipes: **Specifications**

- A specification of a pipe is a relation *S(left, right)* between
  - the sequences of messages input on the left channel and
  - the sequence of messages output on the right.
- In two serial pipes,
  - the sequence *right* produced by the left operand is equal to
    - the sequence *left* consumed by the right operand;
      - this common sequence is concealed.
- The law to avoid the risk of livelock:

L1. If P sat $S(left, right)$ **and** Q sat $T(left, right)$ **and** P is left-guarded **or** Q is right-guarded

then $P \gg Q$ sat $\exists\, s \bullet S(left, s) \wedge T(s, right)$

- We do not reason about refusals
  - since the $\gg$ operator cannot introduce deadlock in pipes.

Communication: Pipes: **Specifications**

X1. *DOUBLE* sat $right \leq^1 double^*(left)$

- *DOUBLE* is left-guarded and right-guarded, so

$$(DOUBLE \gg DOUBLE)$$
$$\text{sat } \exists\, s \bullet (s \leq^1 double^*(left) \wedge right \leq^1 double^*(s))$$
$$\equiv right \leq^2 double^*(double^*(left))$$
$$\equiv right \leq^2 quadruple^*(left)$$

$DOUBLE = \mu\, X \bullet (left\ ?\ x \rightarrow right\ !\ (x + x) \rightarrow X)$

Communication: Pipes: **Specifications**

X2. An alternative definition of a buffer with recursion and with $\gg$:

$$BUFF = \mu\, X \bullet (left\ ?\ x \rightarrow (X \gg (right\ !\ x \rightarrow COPY)))$$

- Prove that
  - $BUFF$ sat $(right \leq left)$
- Assume that
  - X sat $\#left \geq n \vee right \leq left$
- We know that
  - $COPY$ sat $right \leq left$
- Therefore
  - $(right\ !\ x \rightarrow COPY)$ sat $(right = left = \langle\rangle \vee (right \geq \langle x \rangle \wedge right' \leq left))$

$$\Rightarrow right \leq \langle x \rangle \hat{}\, left$$

$$BUFFER = P_{\langle\rangle}$$
$$P_{\langle\rangle} = left\ ?\ x \rightarrow P_{\langle x \rangle}$$
$$P_{\langle x \rangle \hat{}\, s} = (left\ ?\ y \rightarrow P_{\langle x \rangle \hat{}\, s \hat{}\, \langle y \rangle} \mid right\ !\ x \rightarrow P_s)$$

$$COPY = \mu\, X \bullet (left\ ?\ x \rightarrow right\ !\ x \rightarrow X)$$

Communication: Pipes: Specifications

- Since the right operand is right-guarded, by L1 and the assumption

$$(X \gg (right\ !\ x \rightarrow COPY)) \text{ sat } (\exists s \bullet (\#left \geq n \vee s \leq left) \wedge right \leq \langle x \rangle^\frown s) \Rightarrow (\#left \geq n \vee right \leq \langle x \rangle^\frown left)$$

- Therefore

$$left\ ?\ x \rightarrow (\ldots) \text{ sat } right = left = \langle\rangle \vee (left > \langle\rangle \wedge (\#left' \geq n \vee right \leq \langle left_0 \rangle^\frown left')) \Rightarrow \#left \geq n + 1 \vee right \leq left$$

- The desired conclusion follows by the proof rule L8 for recursive processes.
  - The simpler law (L6) cannot be used, because
    - the recursion is not obviously guarded.

L1. If P sat $S(left, right)$ **and** Q sat $T(left, right)$ **and** P is left-guarded **or** Q is right-guarded **then** $P \gg Q$ sat $\exists s \bullet S(left, s) \wedge T(s, right)$

L8. If $S(0)$ **and** $(X \text{ sat } S(n)) \Rightarrow f(X)$ sat $S(n + 1))$ **then** $(\mu X \bullet f(X))$ sat $(\forall n \bullet S(n))$

L6. If $F(x)$ is guarded **and** $STOP$ sat S **and** $((X \text{ sat } S) \Rightarrow (F(X) \text{ sat } S))$ **then** $(\mu X \bullet F(X))$ sat S

Communication: Pipes: **Buffers and protocols**

- A buffer:
  - its outputs on the right is the same sequence of messages as its input from the left,
    - possibly after some delay;
  - when non-empty, it is always ready to output on the right.
  - a process P which never stops, is free of livelock, and meets the specification:

$$P \text{ sat } (right \leq left) \wedge (\text{if } right = left \text{ then } left \notin ref \text{ else } right \notin ref)$$

    - $c \notin ref$ iff the process cannot refuse to communicate on channel c.
  - All buffers are left-guarded.

X1. The following processes are buffers

$$COPY,\ (COPY \gg COPY),\ BUFF,\ BUFFER$$

$$BUFFER = P_{<>}$$
$$P_{<>} = left\ ?\ x \rightarrow P_{<x>}$$
$$P_{<x>^s} = (left\ ?\ y \rightarrow P_{<x>^s^<y>} \mid right\ !\ x \rightarrow P_s)$$

$$BUFF = \mu\ X \bullet (left\ ?\ x \rightarrow (X \gg (right\ !\ x \rightarrow COPY)))$$

$$COPY = \mu\ X \bullet (left\ ?\ x \rightarrow right\ !\ x \rightarrow X)$$

Communication: Pipes: **Buffers and protocols**

- Buffers as specifications for
  - the communications protocol
    - intended to deliver messages in the same order they have been submitted.
    - consists of two processes
      - a transmitter T and a receiver R
        - connected in series $(T \gg R)$.
- If the protocol is correct, $(T \gg R)$ is a buffer.
- The wire (channels, paths) connecting the transmitter to the receiver is quite long,
  - the messages sent are subject to corruption or loss.
  - The behaviour of the wire is modelled by a process *WIRE*, which may not be a buffer.
- It is the task of the protocol designer to ensure that
  - in spite of the bad behaviour of the wire, the system as a whole acts as a buffer:
    - $(T \gg WIRE \gg R)$ is a buffer

Communication: Pipes: **Buffers and protocols**

- A protocol is usually built in a number of layers:
  - (T_1,R_1), (T_2,R_2), … (T_n,R_n)
    - each one uses the previous layer as its communication medium

$$T_n \gg \ldots \gg (T_2 \gg (T_1 \gg WIRE \gg R_1) \gg R_2) \gg \ldots \gg R_n$$

- In practice, all the transmitters are collected into a single transmitter at one end and all the receivers at the other:

$$(T_n \gg \ldots \gg T_2 \gg T_1) \gg WIRE \gg (R_1 \gg R_2 \gg \ldots \gg R_n)$$

  - Due to the law of associativity of $\gg$, the regrouping does not change the behaviour.

- In practice, protocols must be more complicated:
  - it is necessary to add channels in the reverse direction:
    - the receiver could send back acknowledgement signals for successfully transmitted messages,
      - so that unacknowledged messages can be retransmitted.

Communication: Pipes: **Buffers and protocols**

- The laws for proving the correctness of protocols:

L1. If P and Q are buffers, **so are** $(P \gg Q)$ and $(left\ ?\ x \rightarrow (P \gg (right\ !\ x \rightarrow Q)))$

L2. If $T \gg R = (left\ ?\ x \rightarrow (T \gg (right\ !\ x \rightarrow R)))$ **then** $(T \gg R)$ is a buffer.

- The generalisation of L2:

L3. If for some function f and for all z

$$(T(z) \gg R(z)) = (left\ ?\ x \rightarrow (T(f(x, z)) \gg (right\ !\ x \rightarrow R(f(x, z)))))$$

then $T(z) \gg R(z)$ is a buffer for all z.

L1. If P and Q are buffers, **so are** $(P \gg Q)$ and $(left\ ?\ x \rightarrow (P \gg (right\ !\ x \rightarrow Q)))$

L2. If $T \gg R = (left\ ?\ x \rightarrow (T \gg (right\ !\ x \rightarrow R)))$ **then** $(T \gg R)$ is a buffer.

Communication: Pipes: **Buffers and protocols**

$BUFFER = P_{<>}$

$P_{<>} = left\ ?\ x \rightarrow P_{<x>}$

$P_{<x>^s} = (left\ ?\ y \rightarrow P_{<x>^s^<y>}\ |\ right\ !\ x \rightarrow P_s)$

X2. The following are buffers by L1

- $COPY \gg COPY$, $BUFFER \gg COPY$, $COPY \gg BUFFER$, $BUFFER \gg BUFFER$

$COPY = \mu\ X \bullet (left\ ?\ x \rightarrow right\ !\ x \rightarrow X)$

X3. It has been shown that

- $(COPY \gg COPY) = (left\ ?\ x \rightarrow (COPY \gg (right\ !\ y \rightarrow COPY)))$

- By L2 it is a buffer.

$BUFF = \mu\ X \bullet (left\ ?\ x \rightarrow (X \gg (right\ !\ x \rightarrow COPY)))$

X5. (Bit stuffing) The transmitter T reproduces the input bits from left to right, except that after three consecutive outputting *1*–bits, it inserts a single extra *0*.

- The input *01011110* is output as *010111010*.

- The receiver R removes these extra zeroes.

- $(T \gg R)$ must be proved to be a buffer.

- The construction of T and R, and the proof of their correctness, are left as an exercise.

Communication: Pipes: Specifications

X4. (Phase encoding) A phase encoder is a process T which inputs a stream of bits, and outputs *0, 1* for each *0* input and *1, 0* for each *1* input.

- A decoder R reverses this translation
  - $T = left\ ?\ x \rightarrow right\ !\ x \rightarrow right\ !\ (1 - x) \rightarrow T$
  - $R = left\ ?\ x \rightarrow left\ ?\ y \rightarrow$ if $y = x$ then $FAIL$ else $(right\ !\ x \rightarrow R)$
    - the process $FAIL$ is left undefined.
- Prove by L2 that $(T \gg R)$ is a buffer:

$$
\begin{aligned}
(T \gg R) &= left\ ?\ x \rightarrow ((right\ !\ x \rightarrow right\ !\ (1 - x) \rightarrow T) \gg \\
&\qquad (left\ ?\ x \rightarrow left\ ?\ y \rightarrow \textbf{if}\ y = x\ \textbf{then}\ FAIL\ \textbf{else}\ (right\ !\ x \rightarrow R))) \\
&= left\ ?\ x \rightarrow (T \gg\ \textbf{if}(1 - x) = x\ \textbf{then}\ FAIL\ \textbf{else}\ (right\ !\ x \rightarrow R)) \\
&= left\ ?\ x \rightarrow (T \gg (right\ !\ x \rightarrow R))
\end{aligned}
$$

- Therefore $(T \gg R)$ is a buffer, by L2.

Communication: Pipes: Specifications

X6. (Line sharing) Copy data from a channel *left1* to *right1* and from *left2* to *right2*.

- Only a single wire *mid* is available.
- Input messages are tagged by *T* before transmission along *mid*.
- Output messages are untagged by *R* on before sending to the right channel.

  $T = (left1\ ?\ x \rightarrow mid\ !\ tag1.x \rightarrow T \mid left2\ ?\ y \rightarrow mid\ !\ tag2.y \rightarrow T)$
  $R = mid\ ?\ t.z \rightarrow \textbf{if}\ t = tag1\ \textbf{then}\ (right1\ !\ z \rightarrow R)\ \textbf{else}\ (right2\ !\ z \rightarrow R)$
- This solution is quite unsatisfactory.
  - If two messages are input on *left1*, but the recipient is not yet ready for them,
    - the whole system will have to wait, and
      - transmission between *left2* and *right2* may be delayed.
- To insert buffers on the channels will only postpone the problem for a short while.
- The correct solution is to introduce another channel in the reverse direction:
  - *R* sends signals back to *T* to stop sending messages on the stream without necessity.
- This is known as *flow control*.

Communication: **Subordination**

- P and Q are processes with
  - $\alpha P \subseteq \alpha Q$
- In the combination $(P \parallel Q)$,
  - each action of P can occur only when Q permits it to occur;
    - P serves Q as a slave or subordinate process.
  - Q can engage independently in the actions of $(\alpha Q - \alpha P)$.
    - Q acts as a master of main process.
- The process $P /\!\!/ Q$
  - $P /\!\!/ Q = (P \parallel Q) \setminus \alpha P$
  - communications between a subordinate process and a main process are concealed from their common environment.
  - $\alpha(P /\!\!/ Q) = (\alpha Q - \alpha P)$
    - $\alpha P \subseteq \alpha Q$.

Communication: **Subordination**

- A name of the subordinate process is m
  - used in the main process for all interactions with its subordinate.
- A compound channel name is $m.c$
  - m is a process name and c is the name of its channel.
- Communication on this channel is $m.c.v$
  - $am.c(m : P) = ac(P)$ and $v \in ac(P)$.
- In $(m : P /\!/ Q)$:
  - Q communicates with P along channels with compound names $m.c$ and $m.d$;
  - P uses the corresponding simple channels c and d for the same communications.
  - $(m : (c\,!\,v \rightarrow P) /\!/ (m.c\,?\,x \rightarrow Q(x))) = (m : P /\!/ Q(v))$

Communication: **Subordination**

- The name m is a *local* name for the subordinate process
  - All these communications are concealed from the environment;
  - The name can never be detected from the outside.
- Subordination can be nested:
  - $(n : (m : P \mathbin{/\!\!/} Q) \mathbin{/\!\!/} R)$
  - All occurrences of events involving the name m are concealed before
    - the name n is attached to the remaining events.
      - These events are in the alphabet of Q, and not of P.
  - There is no way that R can communicate directly with P, or
    - even know of the existence of P or its name m.

$DOUBLE = \mu\, X \bullet (left\,?\,x \rightarrow right\,!\,(x + x) \rightarrow X)$

Communication: **Subordination**

X1. $doub : DOUBLE /\!/ Q$ $\qquad\qquad$ $doub.left\,!\,e \rightarrow (doub.right\,?\,x \rightarrow \ldots)$

- The subordinate process acts as a simple subroutine called from the main process Q.
- Inside Q, the value of $2 \times e$ may be obtained by a successive output of the argument e on the left channel of *doub*, and input of the result on the right channel

X2.

One subroutine may use another as a subroutine, and do so several times

$$QUADRUPLE = (doub : DOUBLE /\!/ (\mu\, X \bullet left\,?\,x \rightarrow doub.left\,!\,x \rightarrow doub.right\,?\,y \rightarrow doub.left\,!\,y \rightarrow doub.right\,?\,z \rightarrow right\,!\,z \rightarrow X))$$

- This is designed itself to be used as a subroutine:
  - $quad : QUADRUPLE /\!/ Q$
    - This version of $QUADRUPLE$ is similar to $QUADRUPLE_0$, but
      - does not have the same double-buffering effect.

$QUADRUPLE_0 = DOUBLE \gg DOUBLE$

Communication: **Subordination**

$BUFFER = P_{<>}$
$P_{<>} = left\ ?\ x \rightarrow P_{<x>}$
$P_{<x>\hat{}s} = (left\ ?\ y \rightarrow P_{<x>\hat{}s\hat{}<y>} \mid right\ !\ x \rightarrow P_s)$

X3. A conventional program variable named m is modelled as a subordinate process

- $m : VAR \mathbin{/\!/} Q$

$VAR = left\ ?\ x \rightarrow VAR_x$
$VAR_x = (left\ ?\ y \rightarrow VAR_y \mid right\ !\ x \rightarrow VAR_x)$

- Inside the main process Q, the value of m can be
  - assigned, read, and updated by input and output:
- $m := 3; P$ is implemented by $(m.left\ !\ 3 \rightarrow P)$
- $x := m; P$ is implemented by $(m.right\ ?\ x \rightarrow P)$
- $m := m + 3; P$ is implemented by $(m.right\ ?\ y \rightarrow m.left\ !\ (y + 3) \rightarrow P)$

X4. $(q : BUFFER \mathbin{/\!/} Q)$

- The subordinate process serves as an *unbounded queue* named q. Within Q:
  - the output $q.left\ !\ v$ adds v to one end of the queue,
  - $q.right\ ?\ y$ removes an element from the other end, and gives its value to y.
- If the queue is empty, the queue will not respond, and the system may deadlock.

Communication: **Subordination**

X5. A stack with name *st* is declared

- *st* : *STACK* // *Q*
- Inside the main process *Q*:
  - *st.left ! v* can be used to push the value *v* onto the stack,
  - *st.right ? x* will pop the top value.
  - Specify the possibility that the stack is empty:

$$(st.right\ ?\ x \rightarrow Q1(x)\ |\ st.empty \rightarrow Q2)$$

    - If the stack is non-empty, the first alternative is selected;
    - if empty, deadlock is avoided and the second alternative is selected.

$STACK = P_{<>}$
$P_{<>} = (empty \rightarrow P_{<>}\ |\ left\ ?\ x \rightarrow P_{<x>})$
$P_{<x>^s} = (right\ !\ x \rightarrow P_s\ |\ left\ ?\ y \rightarrow P_{<y>^<x>^s})$

Communication: **Subordination**

$$BUFFER = P_{<>}$$
$$P_{<>} = left\ ?\ x \rightarrow P_{<x>}$$
$$P_{<x>\hat{}s} = (left\ ?\ y \rightarrow P_{<x>\hat{}s\hat{}<y>} \mid right\ !\ x \rightarrow P_s)$$

- A subordinate process with several channels may be used by several concurrent processes, provided that they do not use the same channel.

X6. A process Q is intended to communicate a stream of values to R;

- these values are to be buffered by a subordinate buffer process named b,
  - so that output from Q will not be delayed when R is not ready for input.
- Q uses channel $b.left$ for its output and R uses $b.right$ for its input:
  - $(b : BUFFER /\!/ (Q \parallel R))$
- If R attempts to input from an empty buffer,
  - the system will not necessarily deadlock;
  - R will simply be delayed until Q next outputs a value to the buffer.
- If Q and R communicate with the buffer on the *same* channel, then
  - that channel must be in the alphabet of both of them.
  - By the def. of $\parallel$, they should communicate simultaneously the same value — wrong.

Communication: **Subordination**

- The subordination operator may be used to define subroutines by recursion.
- Each level of recursion (except the last) declares a *new* local subroutine to deal with the recursive call(s).

X7 (Factorial)

$$FAC = \mu\, X \bullet left\ ?\ n \rightarrow (\textbf{if}\ n = 0\ \textbf{then}\ (right\ !\ 1 \rightarrow X)\ \textbf{else}\ (f : X \,/\!/\, (f.left\ !\ (n - 1) \rightarrow f.right\ ?\ y \rightarrow right\ !\ (n \times y) \rightarrow X)))$$

- The subroutine *FAC* uses channels
  - *left* and *right* to communicate parameters and results to its calling process;
  - *f.left* and *f.right* to communicate with its subordinate process named *f*.
- In these respects it is similar to the *QUADRUPLE* subroutine.
  - The difference is that the subordinate process is isomorphic to *FAC* itself.

$$QUADRUPLE = (doub : DOUBLE \,/\!/\, (\mu\, X \bullet left\ ?\ x \rightarrow doub.left\ !\ x \rightarrow doub.right\ ?\ y \rightarrow doub.left\ !\ y \rightarrow doub.right\ ?\ z \rightarrow right\ !\ z \rightarrow X))$$

Communication: **Subordination**

$$FAC = \mu\, X \bullet left\,?\, n \rightarrow$$
$$\quad (\textbf{if } n = 0 \textbf{ then } (right\,!\, 1 \rightarrow X)$$
$$\quad\ \textbf{else } (f : X \,/\!/\, (f.left\,!\, (n-1) \rightarrow f.right\,?\, y \rightarrow right\,!\, (n \times y) \rightarrow X)))$$

```
proctype fact( int n; chan p) {
    chan child = [1] of { int };
    int result;
    if
      :: (n <= 1) -> p ! 1
      :: (n >= 2) -> run fact(n-1, child);
                     child ? result;
                     p ! n*result
    fi }
init{
    chan child = [1] of { int };
    int result;
    run fact(7, child);
    child ? result;
    printf("MSC: result: %d\n", result) }
```

Communication: **Subordination**

- Using recursion with subordination to implement an *unbounded data structure*.
  - Each level of the recursion stores a single component of the structure, and
    - declares a *new* local subordinate data structure to deal with the rest.

X8. (Unbounded finite set) A process which implements a set
  - inputs its members on its left channel:

- After each input, it outputs
  - *YES* if it has already input the same value, and
  - *NO* otherwise.

$$SET = left\ ?\ x \rightarrow right\ !\ NO \rightarrow (rest : SET \,/\!/\, LOOP(x))$$
$$LOOP(x) = \mu\, X \bullet left\ ?\ y \rightarrow (\textbf{if}\ y = x\ \textbf{then}\ right\ !\ YES \rightarrow X$$
$$\textbf{else}\ (rest.left\ !\ y \rightarrow rest.right\ ?\ z \rightarrow right\ !\ z \rightarrow X))$$

Communication: **Subordination**

X8. (Unbounded finite set) A process implements a set inputs its members on its left channel:

$$SET = left\ ?\ x \rightarrow right\ !\ NO \rightarrow (rest : SET \ /\!/\ LOOP(x))$$
$$LOOP(x) = \mu\ X \bullet left\ ?\ y \rightarrow (\textbf{if}\ y = x\ \textbf{then}\ right\ !\ YES \rightarrow X$$
$$\textbf{else}\ (rest.left\ !\ y \rightarrow rest.right\ ?\ z \rightarrow right\ !\ z \rightarrow X))$$

- The set starts empty:
  - on input of its first member x is immediately outputs *NO*.
    - It then declares a subordinate process called *rest*,
      - it is going to store all members of the set except x.
- The *LOOP* is designed to input subsequent members of the set.
- If the newly input member is equal to x,
  - the answer *YES* is sent back immediately on the right channel.
- Otherwise, the new member is passed on for storage by *rest*.
- Then the answer (*YES* or *NO*) sent by *rest* is passed on again, and
  - the *LOOP* repeats.

Communication: **Subordination**

X9. (Binary tree) A representation of a set as a binary tree, which
- relies on some given total ordering $\leq$ over its elements.

- Each node
  - stores its earliest inserted element, and
  - declares *two* subordinate trees,
    - one to store elements smaller than the earliest, and
    - one to store the bigger elements.

- The external specification of the tree is the same as X8

$$TREE = left\ ?\ x \rightarrow right\ !\ NO \rightarrow (smaller : TREE \ /\!/\ (bigger : TREE \ /\!/\ LOOP))$$

- The design of the *LOOP* is left as an exercise.

Communication: Subordination: Laws

- The laws restrict communications between a process and its subordinates.
- The first law describes concealed communication in each direction:

L1A. $(m : (c\ ?\ x \rightarrow P(x))) /\!/ (m.c\ !\ v \rightarrow Q) = (m : P(v)) /\!/ Q$

L1B. $(m : (d\ !\ v \rightarrow P)) /\!/ (m.d\ ?\ x \rightarrow Q(x)) = (m : P) /\!/ Q(v)$

- If b is a channel not named by m,
  - the main process can communicate on b without affecting the subordinate:

L2. $(m : P /\!/ (b\ !\ e \rightarrow Q)) = (b\ !\ e \rightarrow (m : P /\!/ Q))$

- The main process only is capable to make a choice for a subordinate process:

L3. $(m : (c\ ?\ x \rightarrow P1(x)\ |\ d\ ?\ y \rightarrow P2(y))) /\!/ (m.c\ !\ v \rightarrow Q) = (m : P1(v) /\!/ Q)$

Communication: Subordination: **Laws**

- If two subordinate processes have the same name, one of them is inaccessible:

L4. $m : P \mathbin{/\!\!/} (m : Q \mathbin{/\!\!/} R) = (m : Q \mathbin{/\!\!/} R)$

- The order in which subordinate processes are written does not matter:

L5. $m : P \mathbin{/\!\!/} (n : Q \mathbin{/\!\!/} R) = n : Q \mathbin{/\!\!/} (m : P \mathbin{/\!\!/} R)$

  - if m and n are distinct names

$SET = left\ ?\ x \rightarrow right\ !\ NO \rightarrow (rest : SET \,/\!/\, LOOP(x))$
$LOOP(x) = \mu\, X \bullet left\ ?\ y \rightarrow (\textbf{if}\ y = x\ \textbf{then}\ right\ !\ YES \rightarrow X\ \textbf{else}\ (rest.left\ !\ y \rightarrow rest.right\ ?\ z \rightarrow right\ !\ z \rightarrow X))$

Communication: Subordination: Laws

- Traces of recursive process with subordination

X1. A typcal trace of *SET* is

- $s = \langle left.1, right.NO, left.2, right.NO \rangle$

L1A. $(m : (c\ ?\ x \rightarrow P(x))) \,/\!/\, (m.c\ !\ v \rightarrow Q) = (m : P(v)) \,/\!/\, Q$
L1B. $(m : (d\ !\ v \rightarrow P)) \,/\!/\, (m.d\ ?\ x \rightarrow Q(x)) = (m : P) \,/\!/\, Q(v)$
L2. $(m : P \,/\!/\, (b\ !\ e \rightarrow Q)) = (b\ !\ e \rightarrow (m : P \,/\!/\, Q))$

- The value of $SET\ /\ s$ can be calculated using laws L1A, L1B, and L2:

$SET\ /\ \langle left.1 \rangle = right\ !\ NO \rightarrow (rest : SET \,/\!/\, LOOP(1))$
$SET\ /\ \langle left.1, right.NO \rangle = (rest : SET \,/\!/\, LOOP(1))$
$SET\ /\ \langle left.1, right.NO, left.2 \rangle = (rest : SET /\!/ (rest.left\ !\ 2 \rightarrow rest.right\ ?\ z \rightarrow right\ !\ z \rightarrow LOOP(1)))$
$= (rest : (right\ !\ NO \rightarrow (rest : SET \,/\!/\, LOOP(2)))) \,/\!/\, (rest.right\ ?\ z \rightarrow right\ !\ z \rightarrow LOOP(1)))$
$= rest : (rest : SET \,/\!/\, LOOP(2)) \,/\!/\, (right\ !\ NO \rightarrow LOOP(1))$
$SET\ /\ s = rest : (rest : SET \,/\!/\, LOOP(2)) \,/\!/\, LOOP(1)$

- It is obvious from this that $\langle left.1, right.NO, left.2, right.YES \rangle$ is *not* a trace of *SET*.
- The reader may check that

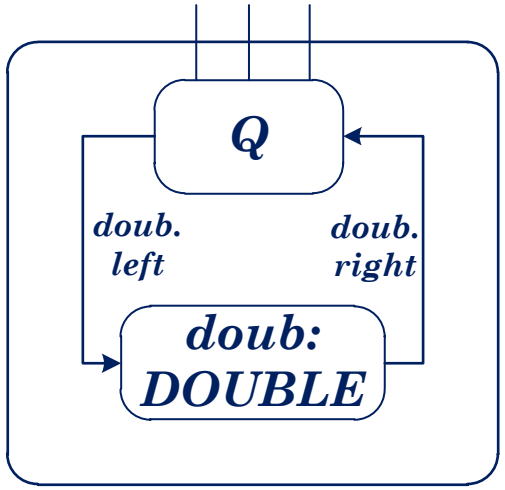
$SET\ /\ s\ \hat{}\ \langle left.2, right.YES \rangle = SET\ /\ s$ and
$SET\ /\ s\ \hat{}\ \langle left.5, right.NO \rangle = rest : (rest : (rest : SET \,/\!/\, LOOP(5)) \,/\!/\, LOOP(2)) \,/\!/\, LOOP(1)$

Communication: Subordination: **Connection diagrams**

- A subordinate process is drawn *inside* the box of the process that uses it, as for **X1**:

X1. *doub* : *DOUBLE* // *Q*

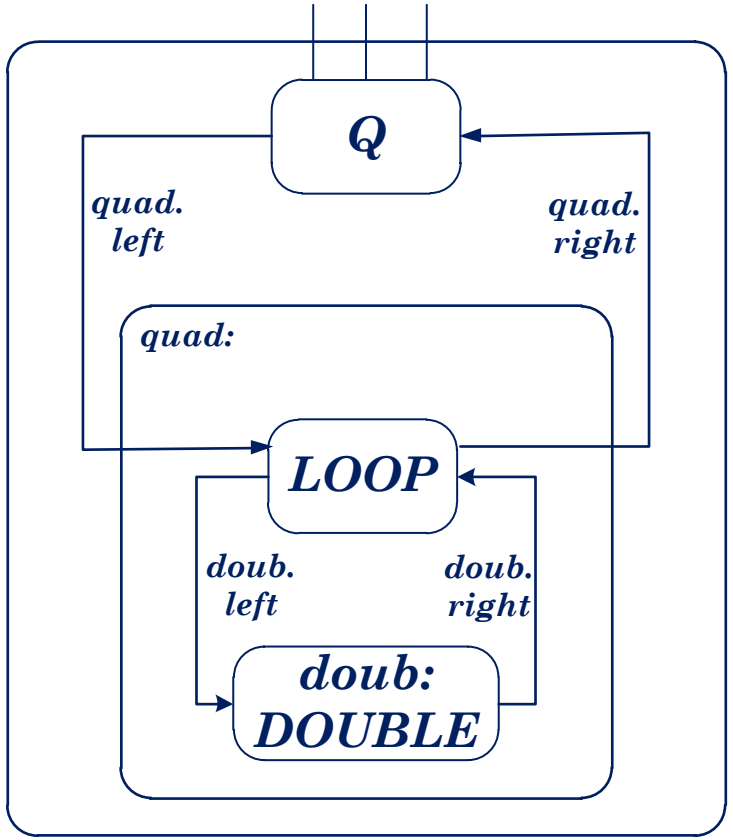
DOUBLE = μ *X* • (*left* ? *x* → *right* ! (*x* + *x*) → *X*)



Communication: Subordination: **Connection diagrams**

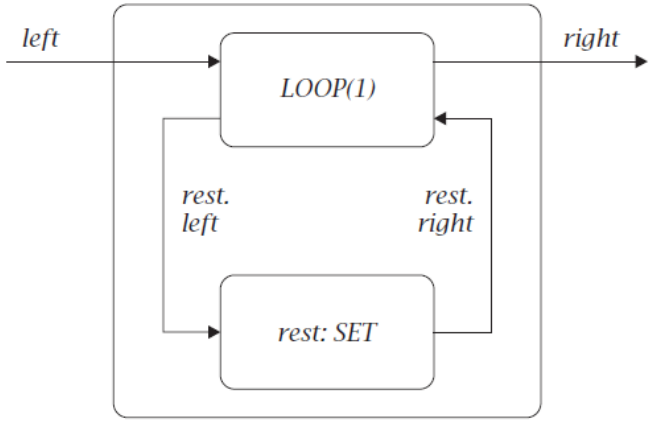
- For nested subordinate processes, the boxes nest more deeply, as for **X2**:

$$QUADRUPLE = (doub : DOUBLE \;/\!/\; (\mu\, X \bullet left\,?\,x \rightarrow doub.left\,!\,x \rightarrow doub.right\,?\,y \rightarrow doub.left\,!\,y \rightarrow doub.right\,?\,z \rightarrow right\,!\,z \rightarrow X))$$

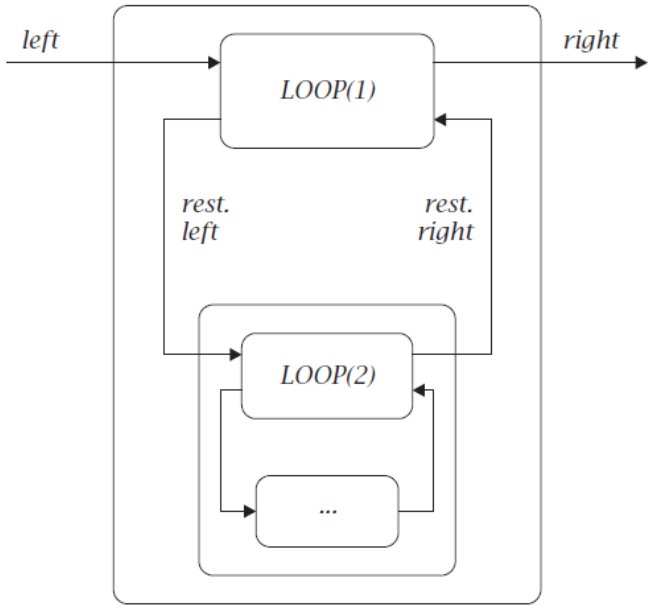


Communication: Subordination: **Connection diagrams**

- A recursive process is one that is nested inside itself.
- We picture successive stages in the early history of a set.



$SET = left\ ?\ x \rightarrow right\ !\ NO \rightarrow (rest : SET \ /\!/\ LOOP(x))$
$LOOP(x) = \mu\ X \bullet left\ ?\ y \rightarrow$
if $y = x$ **then** $right\ !\ YES \rightarrow X$
else $(rest.left\ !\ y \rightarrow rest.right\ ?\ z \rightarrow right\ !\ z \rightarrow X))$



- *SET / <left.1, right.NO>*:

- *SET / <left.1, right.NO, left.2, right.NO>*:

Communication: Subordination: **Connection diagrams**

- If we ignore the nesting of the boxes, this can be drawn as a linear structure:

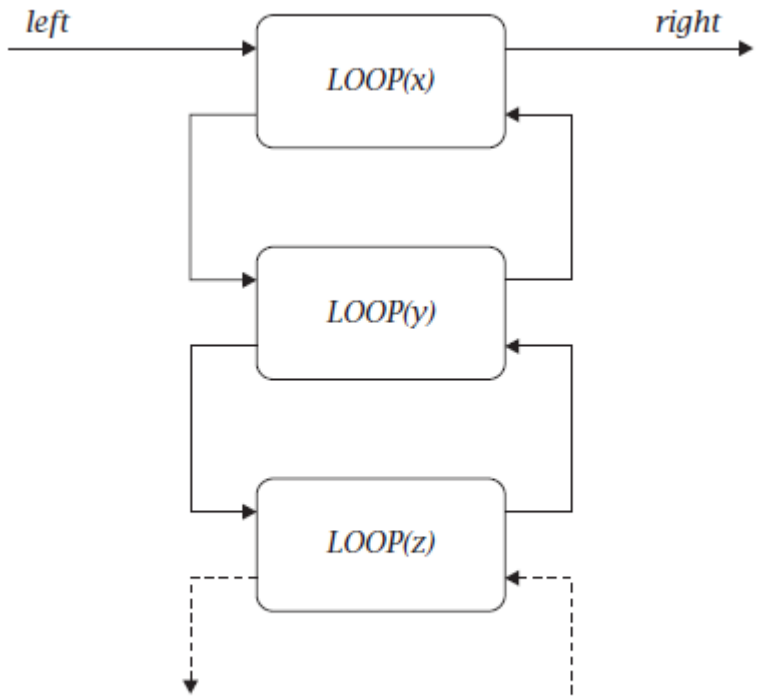


Figure 4.13

Communication: Subordination: **Connection diagrams**

- The example *TREE:*

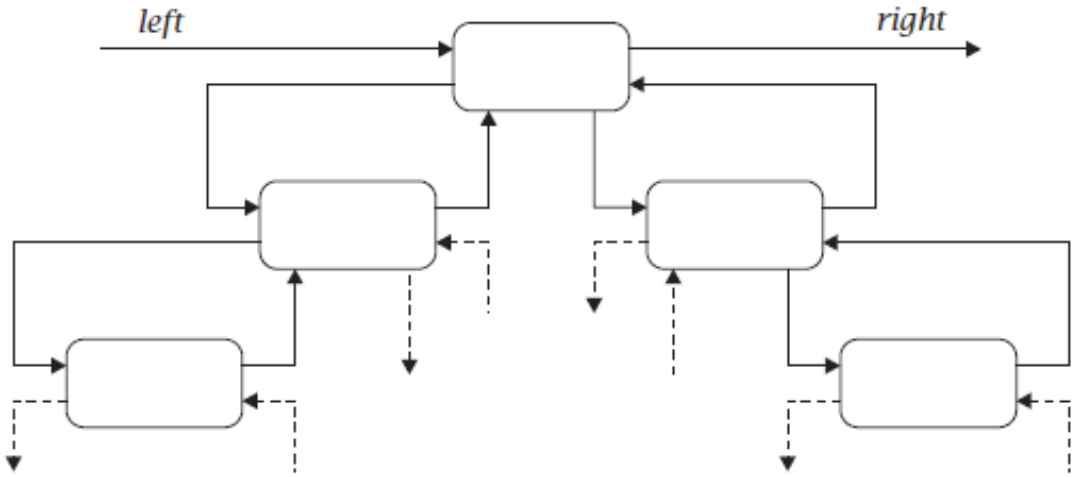


Figure 4.14

$$TREE = left\ ?\ x \rightarrow right\ !\ NO \rightarrow (smaller : TREE \ /\!/\ (bigger : TREE \ /\!/\ LOOP))$$