The LNT Language

Introduction

- LNT mixes concepts from the process algebra and programming "worlds"
  - Control part: Processes, gates, synchronization
  - Data part: Variables, functions, expressions…
- Advantages:
  - Describing complex concurrent systems is easier
  - Lots of similarities with programming languages

Tools for LNT programs

Given a `program.lnt` file, with CADP you can…

- Generate its LTS
  - `lnt.open program.lnt generator program.bcg`
- Perform equivalence checking
  - `lnt.open program.lnt bisimulator spec.bcg`
- Do a simulation (execute e.g., 100 steps randomly)
  - `lnt.open program.lnt executor 100 2`
- Perform model checking
  - `lnt.open program.lnt evaluator property.mcl`

"Coffee or tea" machine in LNT

```
module CoffeeOrTeaMachine is
  process AcceptCoin [coin1, coin2: none] is
    select coin1 [] coin2 end select --choice
  end process
  process Main [nickel, dime, makeC,
    makeT, giveC, giveT: none] is
      AcceptCoin [nickel, dime] ; --invocation
      select makeC; giveC [] makeT; giveT
      end select -- ";" denotes sequencing
  end process
end module
```

Another LNT example (1/2)

```
module University is
    process CS [pub, coin, coffee: none] is
        -- infinite loop
        loop pub ; coin ; coffee end loop
    end process

    process CM [coin, coffee: none] is
        loop coin ; coffee end loop
    end process
    (* another syntax for (multi-line) comments *)
    -- continues on next slide
```

Another LNT example (2/2)

```
  process Main [pub: none] is
    -- rename coin, coffee to i (internal action)
    hide coin, coffee: none in
      -- parallel composition
      -- (forced rendezvous on coin, coffee)
      par coin, coffee in
        CS || CM
      end par
    end hide
  end process
end module
```

Files and modules

- 1 file = 1 module
  - Module must have the same name as the file
  - Names are case-insensitive (as most of LNT)
  - Names can only contain letters, numbers, underscores
  - You can import other modules in the same directory
- Example
  - File `mymodule.lnt`, imports `a.lnt` and `b.lnt` :

```
module MyModule(A, B) is
        …
end module
```

Contents of a module

- Definitions related to the control part
  - Processes, Channels
- Definitions related to the data part
  - Functions, Custom data types
- If you call `lnt.open` on a file, that file *must* contain a `Main` process
  - "Entry point", describes the whole system
  - Similar to main() function in C

LNT

CONTROL PART

Processes

- Definition

```
process MyProcess [gates] (parameters) is
   …
end process
```

- Composition operators
  - Sequential P_1 ; P_2 ; … ; P_n
  - Choice select P_1 [] P_2 [] … [] P_n end select
  - Parallel par P_1 || P_2 || … || P_n end par
  - …

Process parameters

```
process OddOrEven [odd, even: none] (x : int) is
    if (x mod 2) == 0 then even else odd end if
end process
process Main [odd, even : none] is
    OddOrEven [odd, even] (4)  -- invocation
end process
```

- Similar to function parameters
- The behaviour of `OddOrEven` changes according to the actual parameter (in this case, 4).
- `Main` cannot have parameters!

Variables and assignments

- `var` is used to declare one or more variables.
- Variables are never shared, always local
- Within processes, assignments (`:=`) may be deterministic or not (`any`)
- Nondet. assignments may be constrained (`where`)

```
var x : nat in
    x := 3 * 4 + 1 ;
    x := any nat ;
    x := any nat where x < 4
end var -- x cannot be accessed after this
```

Semantics of any … where

Nondeterministic assignment is equivalent to a `select` of deterministic assignments for every possible value (possibly constrained by `where`)

```
x := any nat where x < 4
```

is equivalent to

```
select
    x := 0 [] x := 1 [] x := 2 [] x := 3
end select
```

Exercise

Write an LNT process that

- Performs do *a*, *b*, *c* in any order
- After performing all of the above, if performs either *d* or *e*

- Hints
  - You will need all the basic composition operators (`;` , `select`, `par`)
  - Use a gate for each action

Solution

```
process Exercise1 [a, b, c, d, e : none] is
   par a || b || c end par ;
   select d [] e end select
end process
```

Notice that ; is an operator, not a terminator

- unlike C or Java

So you must not put ; after `end select`

Gates and channels (1/2)

- A gate is a communication endpoint for a process
- Until now, we have only seen none gates
  - Pure synchronization, without exchange of data
  - Like CCS, but symmetrical (no complementary actions)
- In general, LNT allows to describe gates where data can be sent and received
- We can constrain the type of data allowed on a gate, by means of channels
  - none: no data is allowed
  - any: everything is allowed

Gates and channels (2/2)

- Example

```
channel natChannel is nat end channel
process P [g1: none, g2 : natChannel] is
  g1 ;       -- Synchronise over gate g1
  g2 (10)    -- Offer “10” over gate g2
end process
```

- More complex channel definitions:

```
-- either one nat or a pair of ints
channel chan is nat, (int, int) end channel
```

- Predefined types: **bool**, **char**, **nat**, **int**, **real**, **string**

Data reception (1/2)

```
process P1 [g : any] is
    var n : nat in
        g (?n)
    end var
end process
```

`var` is used to declare a variable

`g(?n)` = if someone else sends a `nat` over gate g, P1 will receive it and store it in variable `n`

Data reception (2/2)

We can add constraints on the data we want to receive with where

```
process P1 [g : any] is
    var n : nat in
        g (?n) where n > 10
    end var
end process
```

P1 will only accept values > 10

Semantics of reception

- These 3 fragments are equivalent (n is a variable of type `nat`):
  - `g(?n)`
  - `n := any nat ; g(n)`
  - `select n := 0 [] … end select ; g(n)`
- Reception *looks like* an asymmetrical rendezvous, but is actually symmetrical
- When, say, `g(10)` synchronises with `g(?n)`, it is actually synchronising with the branch where `n` has been set to `10` (thus, both processes are performing an action `g(10)`)

User-defined data types

LNT allows user-defined types, for instance:

- Enums

```
type Answer is Yes, No, Maybe end type
```

- Records

```
type Point2D is point (x: Int, y: Int) end type
```

- Arrays (static size)

```
type Triangle is array [0..2] of Point2D end type
```

After being defined, they can be used just like predefined types (e.g., in channels)

`null` and `stop`

`null` is a "null operation", while `stop` is the deadlocked process

- `null ; P` is equivalent to `P`
  - `null` simply terminates without visible actions
- `stop ; P` is equivalent to `stop`
  - `stop` does not terminate, thus `P` can never be executed

Parallel composition (1/2)

- No synchronization

```
par P1[g1, g2…] || P2[…] || … || Pn[…] end par
```

- Global synchronization

```
par g1, g2, … in … end par
```

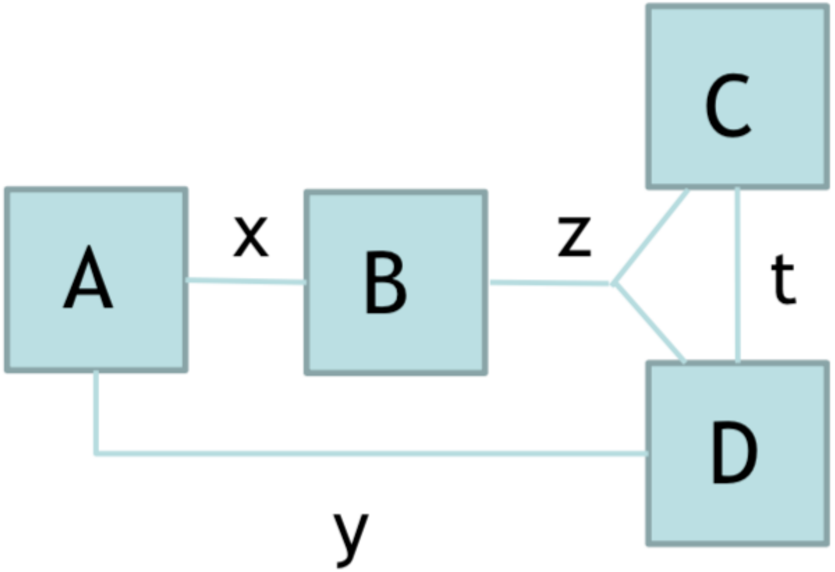
- Partial synchronization

```
par
      g1 -> P1 [… , g1, …]
   || g1 -> P2 [… , g1, …]
   || g2 -> P3 [g1 , g2, …]
end par -- P3 won't sync with P1, P2
```

Parallel composition (2/2)

- Partial synchronization: process `g1, … -> P` must synchronize with all other processes having g1 in their synchronization list (`…, g1, … -> Q`)
- Think graphically:

```
par
   x, y -> A
|| x, z -> B
|| z, t -> C
|| y, z, t -> D
end par
```

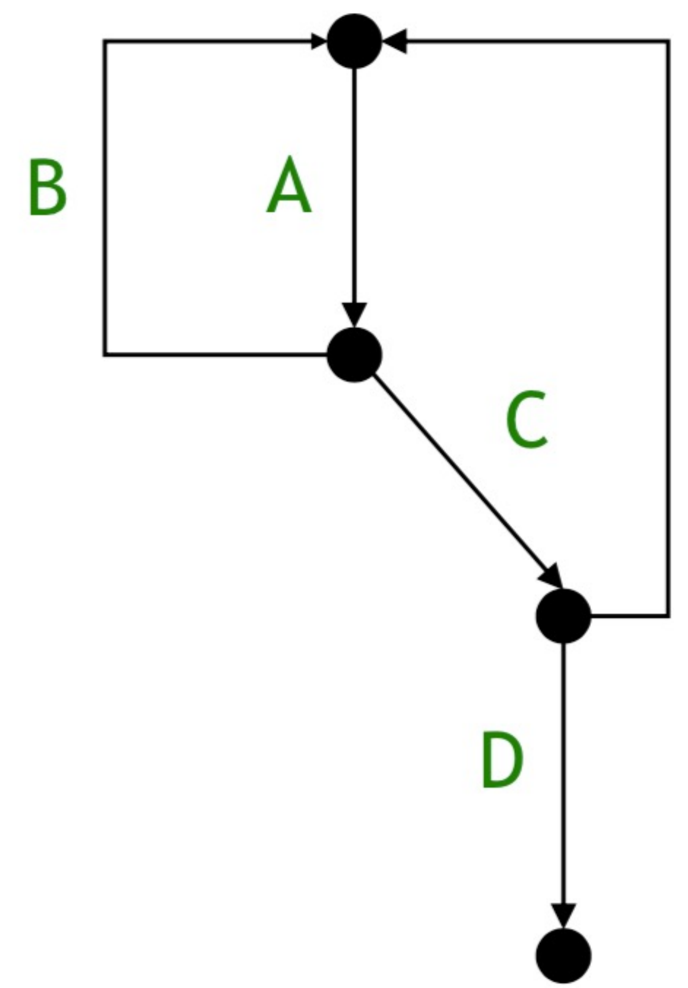


LNT control part: other constructs

- Conditionals
  - `if c1 then P elsif c2 then P2 else P3 end if`
  - `only if c1 then P1 elsif c2 then P2 end if`
    - Same as `if … else stop end if`
- Loops
  - `loop P end loop` (infinite)
  - `loop L in … break L … end loop` (breakable)
  - `while c loop … end loop`
  - `for x:=0 while x<10 by x:=x+1 loop … end loop`
- Pattern matching (similar to C's `switch`)
  - `case x in case1 -> P1 | … | any -> P2 end case`

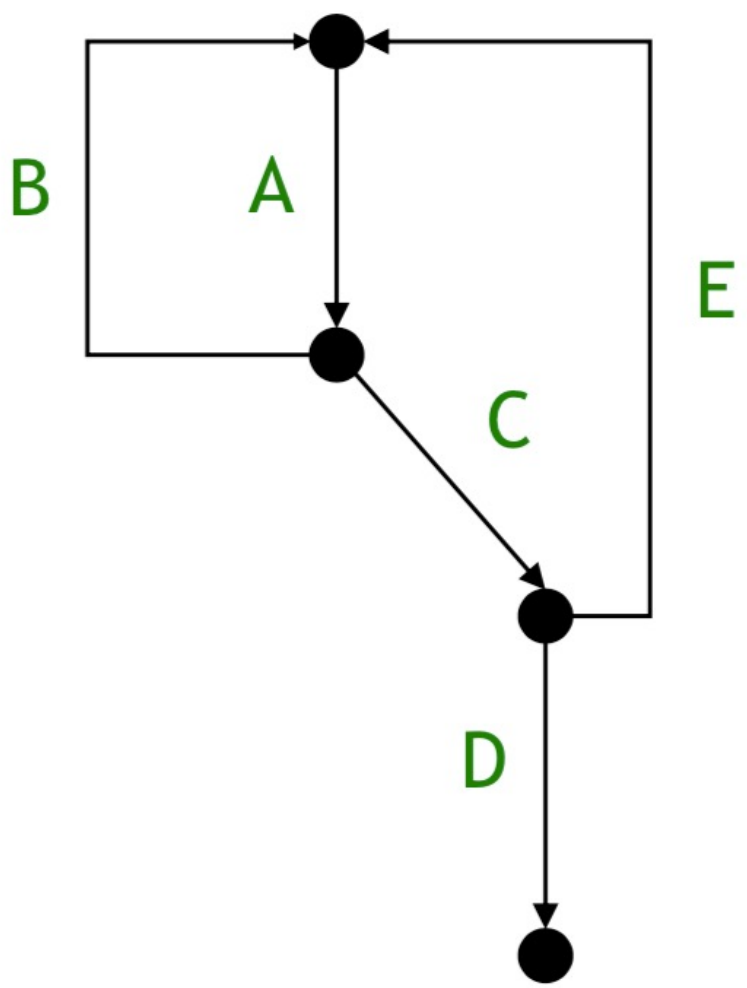
Exercise

- Encode this LTS as an LNT process P
- Hints:
  - You only need `;` and `select` (no `par`)
  - For cyclic behaviour, You can either use loops or recursion (up to you)



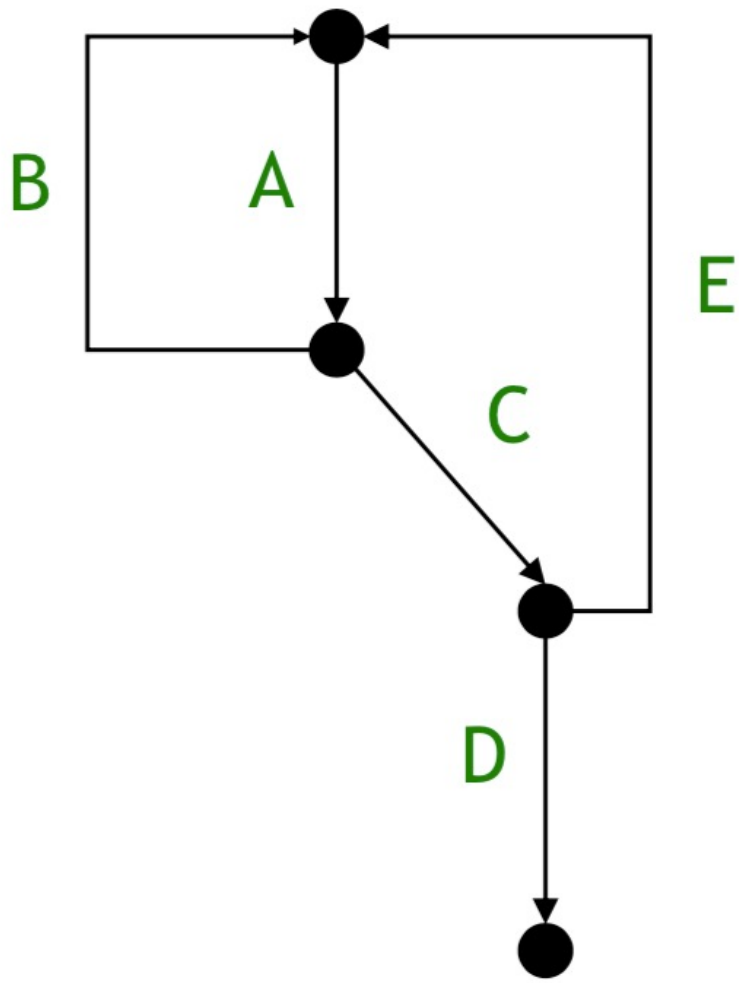
Solution (with loops)

```
process P [A,B,C,D,E:none] is
  loop
    A ;
    select
       B
    [] C ;
       select
         D ; stop [] E
       end select
    end select
  end loop
end process
```



Solution (with recursion)

```
process P [A,B,C,D,E:none] is
  A ;
  select
     B; P[A,B,C,D,E]
  [] C ;
     select
       D ; stop
     []
       E ; P [A,B,C,D,E]
     end select
  end select
end process
```



LNT

DATA PART

Functions

- Definition

```
function myFunction (parameters): returnType is
        …
end function
```

- Similar to processes, but:
  - Cannot have gates
  - May have a return type: for instance:

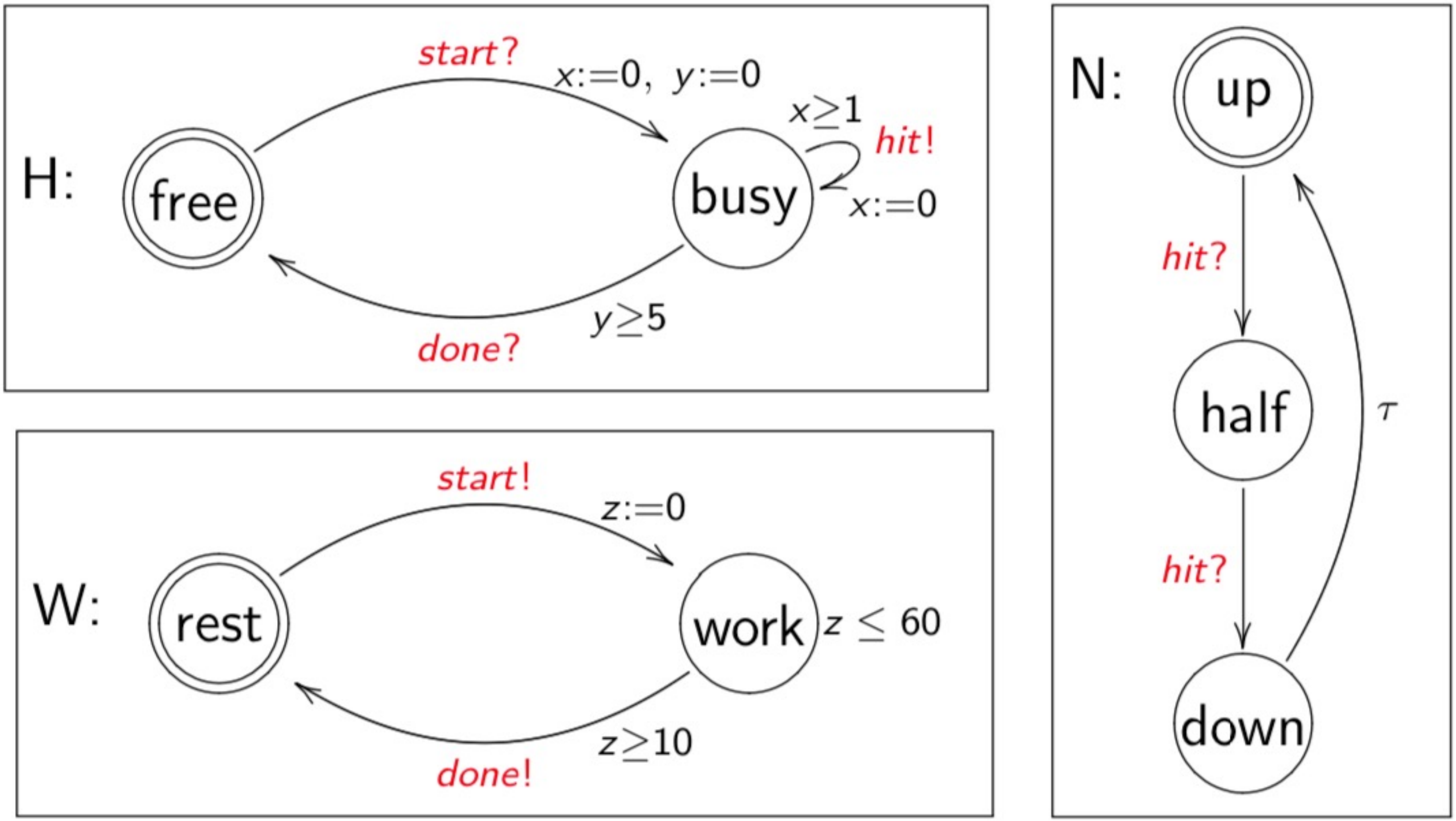
```
function sum (x, y: nat): nat is
      return x + y
end function
```

Differences between control/data parts

- LNT functions are deterministic and sequential
- Within a function, you cannot use:
  - `stop`, `only if`
  - gate actions
  - `any`
  - `select` , `par`, `hide`

- You cannot call processes from functions
- (You can call functions from processes)
- You can use `return` only in functions

Exercise (1/2)



H:
free
busy
start?
x:=0, y:=0
x≥1
hit!
x:=0
y≥5
done?
W:
rest
work
start!
z:=0
z ≤ 60
z≥10
done!
N:
up
half
down
hit?
hit?
τ

Exercise (2/2)

- Describe H, N, W as LNT processes
  - Disregard `?`/`!` and temporal constraints
  - The invisible action τ is written `i` in LNT

- Write a `Main` process such that:
  - H and W synchronise on *start*, *done*
  - H and N synchronise on *hit*

Solution (1/3)

- Processes `N` and `W`: use unbreakable loops
- Of course, recursive processes can also be used

```
process N [hit: none] is
  loop hit; hit; i end loop
end process

process W [start, done: none] is
  loop start; done end loop
end process
```

Solution (2/3)

- Process H: use a breakable loop to describe the *hit* self-transition in the *busy* state

```
process H [start, done, hit: none] is
  loop
    start;
    loop L in select
         hit [] break L
     end select end loop;
    done
  end loop
end process
```

Solution (3/3)

- Process `Main`: use partial synchronization

```
process Main [start, done, hit: none] is
  par
    start, done, hit -> H[start, done, hit]
  ||
    start, done -> W[start, done]
  ||
    hit -> N[hit]
  end par
end process
```

LNT reference manual

- Champelovier et al., “Reference Manual of the LNT to LOTOS Translator”
  - Technical report, available on the CADP website
  - Complete description of LNT
  - Despite the title, no knowledge of LOTOS is required

- Relevant sections:
  - Ch. 5, 6, 7, 8: types, channels, functions, processes
  - Appendix B: Built-in functions and operators