

**LINFO1104 – LSINC1104**  
**Concepts, paradigms, and semantics**  
**of programming languages**

**Lecture 4**  
**Higher-order programming**

Peter Van Roy

ICTEAM Institute  
Université catholique de Louvain

[peter.vanroy@uclouvain.be](mailto:peter.vanroy@uclouvain.be)



1

## Overview of lecture 4

- Refresher of procedure semantics
- Higher-order programming
  - Order of a function
  - Genericity
  - Instantiation
  - Function composition
  - Abstracting an accumulator
  - Encapsulation
  - Delayed execution



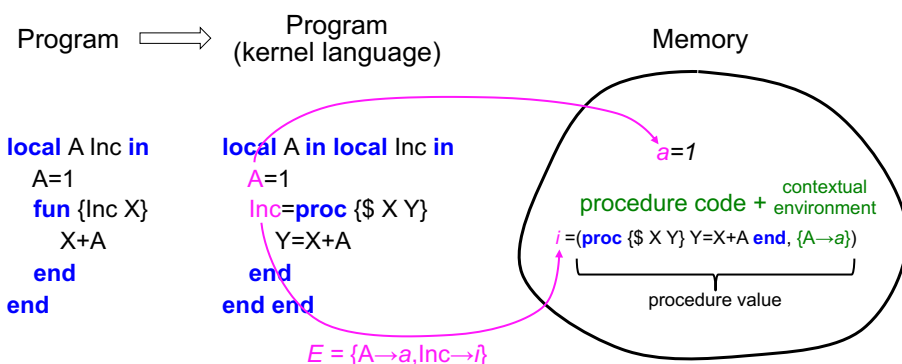
2

# Procedure semantics refresher



3

## Procedure value in memory



4

## Procedure call

- Practical language:

{Browse {Inc 10}}

- Kernel language:

```
local M in
  local N in
    M=10
    {Inc M N}
    {Browse N}
  end
end
```

5

5

## Execution of {Inc M N}

Important slide

- $\sigma \left[ \left( \{ \text{Inc } M \ N \}, \{ M \rightarrow m, N \rightarrow n, \text{Inc} \rightarrow i, \text{Browse} \rightarrow b \} \right), \right.$   
 $\left. \left( \{ \text{Browse } N \}, \{ M \rightarrow m, N \rightarrow n, \text{Inc} \rightarrow i, \text{Browse} \rightarrow b \} \right) \right],$   
 $\left\{ \begin{array}{l} m=10, n, i = (\text{proc } \{ \$ X Y \} Y=X+A \text{ end}, \{ A \rightarrow a \}), \\ a=1, b=(\dots \text{browser code} \dots) \end{array} \right\}$

One execution step

Procedure body  $Y=X+A$   
 New environment  $\{ A \rightarrow a, X \rightarrow m, Y \rightarrow n \} =$   
 contextual environment  $\{ A \rightarrow a \}$   
 + formal arguments  $\{ X \rightarrow m, Y \rightarrow n \}$

- $\sigma \left[ \left( Y=X+A, \{ A \rightarrow a, X \rightarrow m, Y \rightarrow n \} \right), \right.$   
 $\left. \left( \{ \text{Browse } N \}, \{ M \rightarrow m, N \rightarrow n, \text{Inc} \rightarrow i, \text{Browse} \rightarrow b \} \right) \right],$

6

6

# Higher-order programming



7

## Higher-order programming



- Defining a procedure as **a procedure value with a contextual environment** is enormously expressive
  - It is the **most important invention** in programming languages
  - It makes possible building large systems based on data abstraction, which underlies the success of information technology in society
- Since procedures (and functions) are values, we can pass them as inputs to other functions and return them as outputs
  - Remember that in our kernel language, we consider functions and procedures to be the same concept: a function is a procedure with an extra output argument

8



## Order of a function

- We define the **order** of a function (or procedure)
  - A function whose inputs and output are not functions is **first order**
  - A function is **order N+1** if its inputs and output contain a function of maximum order N
- Let's give some examples to show what we can do with higher-order functions (where the order is greater than 1)
  - We will give more examples later in the course

9



## Genericity

- Genericity is when a function is passed as an input

```
declare
fun {Map F L}
  case L of nil then nil
  [] H|T then {F H}{Map F T}
  end
end

{Browse {Map fun {$ X} X*X end [7 8 9]}}
```

What is the order of Map in this call?

10



## Instantiation

- Instantiation is when a function is returned as an output

```
declare  
fun {MakeAdd A}  
  fun {$ X} X+A end  
end  
Add5={MakeAdd 5}  
  
{Browse {Add5 100}}
```

What is the order of MakeAdd?

What is the contextual environment of the function returned by MakeAdd?

11



## Function composition

- We take two functions as input and return their composition

```
declare  
fun {Compose F G}  
  fun {$ X} {F {G X}} end  
end  
Fnew={Compose fun {$ X} X*X end  
             fun {$ X} X+1 end}
```

What is the contextual environment of the function returned by Compose?

- What does {Fnew 2} return?
- What does {{Compose Fnew Fnew} 2} return?

12

## Abstracting an accumulator



- We can use higher-order programming to do a computation that **hides an accumulator**
- Let's say we want to sum the elements of a list  $L=[a_0 \ a_1 \ a_2 \ \dots \ a_{n-1}]$ :
  - $S = a_0 + a_1 + a_2 + \dots + a_{n-1}$
  - $S = (\dots(((0 + a_0) + a_1) + a_2) + \dots + a_{n-1})$
- We can write this **generically** with a function  $F$ :
  - $S = \{F \ \dots \ \{F \ \{F \ \{F \ 0 \ a_0\} \ a_1\} \ a_2\} \ \dots \ a_{n-1}\}$
- Now we can define the **higher-order function FoldL**:
  - $S = \{\text{FoldL} \ [a_0 \ a_1 \ a_2 \ \dots \ a_{n-1}] \ F \ 0\}$
  - The accumulator is hidden inside FoldL!

13

## Definition of FoldL



- Here is the definition of FoldL:

```
declare
fun {FoldL L F U}
  case L
  of nil then U
  [] H|T then {FoldL T F {F U H}}
  end
end
S={FoldL [5 6 7] fun {$ X Y} X+Y end 0}
```

The argument U is an accumulator

What is the order of FoldL?

14

## Encapsulation



- We can hide a value inside a function:

```
declare
fun {Zero} 0 end
fun {Inc H}
  N={H}+1 in
    fun {$} N end
  end
  Three={Inc {Inc {Inc Zero}}}
  {Browse {Three}}
```

- This is the foundation of encapsulation as used in data abstraction
- What is the difference if we write Inc as follows:  
`fun {Inc H} fun {$} {H}+1 end end`

15

## Delayed execution



- We can define an statement and pass it to a function which decides whether or not to execute it

```
proc {IfTrue Cond Stmt}
  if {Cond} then {Stmt} end
end
Stmt = proc {$} {Browse 111*111} end
{IfTrue fun {$} 1<2 end Stmt}
```

- This can be used to build control structures from scratch (if statement, while loop, for loop, etc.)

16



## Building a while loop (1)



- We build a generic while loop of this form:  
`s=init; while (cond(s)) s=transform(s);`  
All while loops can be written in this form

```
fun {While S Cond Transform}  
  if {Cond S} then  
    {While {Transform S} Cond Transform}  
  else S end  
end  
{While Init Cond Transform}
```

17

## Building a while loop (2)



- Here is a while loop that sums integers from 1 to  $n$ 
  - State is a pair  $s(I\ A)$  where  $I$  is the index and  $A$  is an accumulator
- {Browse {While  $s(10\ 0)$   
 **fun** { $\$ S$ }  $S.1 > 0$  **end**  
 **fun** { $\$ S$ }  $s(S.1-1\ S.1+S.2)$  **end}}**
- Practical languages will define syntactic sugar for this:

```
i=10; a=0; while (i>0) { a=a+i; i--; }
```

18

## Summary of higher-order



- We have given **six examples** to illustrate the expressiveness of higher-order programming:
  - Genericity
  - Instantiation
  - Function composition
  - Abstracting an accumulator
  - Encapsulation
  - Delayed execution
- We will use these techniques and others when we introduce the concepts of data abstraction
  - Data abstraction is built on top of higher-order programming!