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

Lecture 3
Formal semantics

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview of lecture 3

- Refresher of lecture 1
  - Environment and scope
  - How procedures are stored in memory
- Why do we need semantics?
- Operational semantics in five parts
  - Complete kernel language
  - Executing the abstract machine
  - Defining the abstract machine
  - Proving correctness of programs
  - Procedures
- Semantics summary

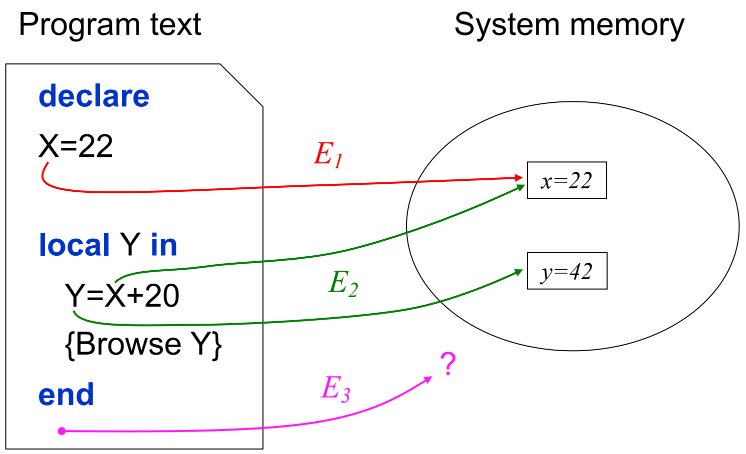
2

2

Refresher of lecture 1

3

Program and memory…



4

4

Environment

- Environments E_1, E_2, E_3
  - Function from identifiers to memory variables
  - A set of pairs $X \to x$
    - Identifier X, memory variable x
- Example environment E_2
  - $E_2=\{X \to x, Y \to y\}$
  - $E_2(X)=x$
  - $E_2(Y)=y$

5

5

An exercise on static scope

What does this program display?

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

6

6

What is the scope of P?

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

7

7

What is the scope of P?

Scope of **P**

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

The P definition *inside* the scope

8

8

Contextual environment of Q

Scope of P

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

All the violet Ps refer to the same variable

Procedure Q must know the definition of P ⇒ it stores this in its *contextual environment*

9

9

The contextual environment

- The contextual environment of a function (or procedure) contains all the identifiers that are used *inside* the function but declared *outside* of the function

  ```
  declare
  A=1
  proc {Inc X Y} Y=X+A end
  ```

- The contextual environment of Inc is $E_C = \{A \rightarrow a\}$
  - Where *a* is a variable in memory: *a=1*

10

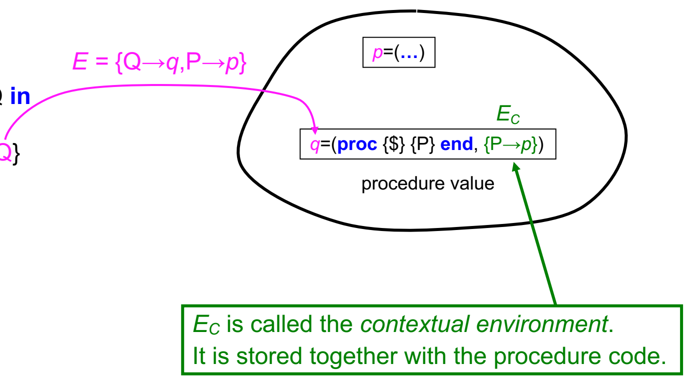
10

How procedure Q is stored in memory

Program text

System memory

```
local P Q in
  (...)
  proc {Q}
    {P}
  end
  (...)
end
```



11

11

Procedure values

- A procedure value is stored in memory as a pair:

$$inc = (\underbrace{\textbf{proc}\ \{\$\ X\ Y\}\ Y{=}X{+}A\ \textbf{end}}_{\substack{\text{Procedure code} \\ \text{(no name: '\$' replaces the identifier)}}},\ \underbrace{\{A \rightarrow a\}}_{\text{Contextual environment}})$$

- The variable *inc* is bound to the procedure value
  - Terminology: a procedure value is also called a *closure* or a *lexically scoped closure*, because it "closes" over the free identifiers when it is defined

12

12

How Q is defined and called

- Recall the definition of Q:
  proc {Q} {P} **end**

- When Q is defined, an environment E_c is created that contains P and E_c is stored together with Q's code
  - E_c={P→*p*} is called the contextual environment of Q

- When Q is called, E_c is used to get the right variable for P
  - This is guaranteed to always get the right variable, even if there is another definition of P right next to the call of Q

- The identifiers in E_c are used inside Q and defined outside of Q
  - They are called the free identifiers of Q

13

13

Free identifiers

- A free identifier of an instruction is an identifier used inside the instruction that is declared outside the instruction

- Procedure arguments are not free because the argument defines the identifier

- The instruction:
  local Z **in** Z=X+Y **end**
  has two free identifiers:
  {X,Y}

- The instruction:
  local Q **in**
  proc {Q A} {P A+1} **end**
  end
  has one free identifier:
  {P}

- A is not free!

14

14

Why do we need semantics?

15

Why do we need semantics?

- If you do not understand something, then you do not master it – it masters you!
  - If you know nothing about how a car works, then a car mechanic can charge you whatever he wants
  - If you do not understand how government works, then you cannot vote wisely and the government becomes a tyranny
- The same holds true for programming
  - To write correct programs and to understand other people's programs, you have to understand the language deeply
  - All software developers should have this level of understanding
  - This understanding comes with the formal semantics

16

16

What is the semantics of a language?

- The semantics of a programming language is a fully precise explanation of how programs execute
  - With it we can reason about program design and correctness
- We give the semantics for all paradigms of this course
  - We start by giving the semantics of functional programming
- Before taking the plunge, let's take a step back and talk about semantics in general

17

17

Four ways to define language semantics

- Four ways have been invented to define semantics:
  - Operational semantics: Explains a program in terms of its execution on a simplified computer, called the abstract machine
    - This works for all paradigms!
  - Axiomatic semantics: Explains a program as an implication: if certain properties hold before the execution, then some other properties will hold after the execution
    - « If the precondition holds before, then the postcondition will hold after » as shown in LEPL1402
    - This works well for imperative paradigms (like object-oriented programming as in Java)
  - Denotational semantics: Explains a program as a function over an abstract domain, which simplifies certain kinds of mathematical analysis of the program
    - This works well for functional paradigms such as implemented by Haskell and Scheme
  - Logical semantics: Explains a program as a logical model of a set of logical axioms, so program execution is deduction: the result of a program is a true property derived from the axioms
    - This works well for logic paradigms such as implemented by Prolog and constraint programming
- We will focus on operational semantics

This course

LEPL1402

18

18

Operational semantics

- The operational semantics has two parts
  - Kernel language: first translate the program into the kernel language
  - Abstract machine: then execute the program on the abstract machine

- We will introduce the operational semantics in five parts
  1. The complete kernel language for functional programming
  2. Executing a program on the abstract machine
  3. Defining the abstract machine and its semantic rules
  4. Proving the correctness of a program using the semantics
  5. Procedure definition and call are special because they are the foundation of data abstraction. We define the semantic rules of procedure definition and call.

Semantics 1: Complete kernel language

Kernel language of functional programming

- We have seen all concepts of functional programming
  - Now we can define its complete kernel language
- We will use this kernel language to understand exactly what a functional program does
  - We have used it to see why list functions are tail-recursive
  - We will use it to prove correctness of programs
- Each time we introduce a new paradigm in the course we will define its kernel language
  - Each extends the functional kernel language with a new concept

21

The functional kernel language (what we saw before)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <list> | …
- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

This is the kernel language with lists and procedure statements

Still incomplete

22

The functional kernel language

- <s> ::= **skip**
  - | <s>$_1$ <s>$_2$
  - | **local** <x> **in** <s> **end**
  - | <x>$_1$=<x>$_2$
  - | <x>=<v>
  - | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  - | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  - | {<x> <y>$_1$ … <y>$_n$}
  - | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <list> | ...
- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

This is what we have seen so far; it needs *two changes* to become the full kernel language of the functional paradigm

1. Procedure declarations (should be values)

2. Records instead of lists (records subsume lists)

23

23

The functional kernel language (procedure values)

- <s> ::= **skip**
  - | <s>$_1$ <s>$_2$
  - | **local** <x> **in** <s> **end**
  - | <x>$_1$=<x>$_2$
  - | <x>=<v>
  - | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  - ~~| **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**~~
  - | {<x> <y>$_1$ … <y>$_n$}
  - | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> | <list> | ...
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <list>, <p> ::= nil | <x> | <x> '|' <list>

1. Procedures are values in memory (like numbers and lists)

This is called an "anonymous procedure". The procedure name is replaced by a placeholder "$".

24

24

The functional kernel language (records)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> ~~| <list>~~ | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

2. Records subsume lists

25

25

The functional kernel language (complete)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

Procedure values and records are important basic types. They allow to define data abstractions including all of object-oriented programming.

26

26

Semantics 2: Executing the abstract machine

27

Executing a program with the abstract machine

- We execute the program using the semantics by following two steps:
- First, we translate the program into kernel language
  - We use the kernel language of functional programming
  - All programs can be translated into kernel language
- Second, we execute the translated program on the abstract machine
  - The abstract machine is a simplified computer with a precise mathematical definition

→ Let's see an example execution

28

28

The example program in kernel language

```
local X in
  local B in
    B=true
    if B then X=1 else skip end
  end
end
```

29

29

Start of the execution: the initial execution state

```
([(local X in
     local B in
       B=true
       if B then X=1 else skip end
     end
   end, {})],
 {})
```

instruction stack [...]

empty environment

empty memory

memory

([(<s>,*E*)],σ)

instruction environment

Execution state

- The initial execution state has an empty memory {} and an empty environment {}
- We start execution with ***local*** *X* ***in*** <*s*> ***end***

30

30

The *local X in ... end* instruction

```
([(local B in
      B=true
      if B then X=1 else skip end
    end,
    {X → x})],
 {x})
```

- We create a new variable x so the memory becomes $\{x\}$
- We create a new environment $\{X \rightarrow x\}$ so that X can refer to the new variable x

31

31

The *local B in ... end* instruction

```
([((B=true
       if B then X=1 else skip end),
     {B → b, X → x})],
 {b,x})
```

- We create a new variable b in memory
- We put the inner instruction on the stack and add $B \rightarrow b$ to its environment, giving $\{B \rightarrow b, X \rightarrow x\}$

32

32

The sequential composition instruction

([(B=**true**,{B $\rightarrow$ *b*, X $\rightarrow$ *x*}),
(**if** B **then** X=1
else skip end,{B $\rightarrow$ *b*, X $\rightarrow$ *x*})],
{*b*,*x*})

- We split the sequential composition into its two parts
  - B=**true** *and* **if** B **then** X=1 **else skip end**
- We put the two instructions on the stack
- Each instruction gets the same environment

33

33

The *B=true* instruction

([(**if** B **then** X=1
else skip end,{B $\rightarrow$ *b*, X $\rightarrow$ *x*})],
{*b*=**true**, *x*})

- We bind variable *b* to **true** in memory
- We remove the (B=**true**,{B $\rightarrow$ *b*, X $\rightarrow$ *x*}) instruction from the stack
- Now only the **if** instruction remains

34

34

The conditional (if) instruction

([(X=1,{B $\rightarrow$ *b*, X $\rightarrow$ *x*})],
{*b*=**true**, *x*})

- We read the value of B
- Since B is **true**, it puts the instruction after **then** on the stack
- (If B is **false**, it will put the instruction after **else** on the stack)
- If B has any other value, then the conditional raises an error

- (If B is unbound then the execution of the semantic stack stops until B becomes bound – this can only happen in another semantic stack, i.e., with concurrency, as we will see) 35

35

The *X=1* instruction

([],
{*b*=**true**, *x*=1})

- We bind *x* to 1 in memory
- Execution stops because the stack is empty

36

36

Semantic rules we have seen

- This example has shown us the execution of four instructions:
  - **local** <x> **in** <s> **end** (variable creation)
  - <s>$_1$ <s>$_2$ (sequential composition)
  - **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end** (conditional)
  - <x>=<v> (assignment)
- We will define the semantic rules corresponding to these instructions

37

37

Semantics 3: Defining the abstract machine

38

Abstract machine execution algorithm

For a microprocessor, this pseudocode definition is implemented by digital circuits!

```
procedure execute(<s>)
var ST, σ, SI;
begin
     ST:=[(<s>,{})];      /* Initial semantic stack: one instruction, empty env. */
     σ:={};               /* Initial memory: empty (no variables) */
     while (ST≠{}) do
          SI:=top(ST);    /* Get topmost element of semantic stack */
          (ST,σ) :=rule(SI, (ST,σ)); /* Execute SI according to its rule */
     end
end
```

each kernel instruction has a rule

- While the semantic stack is nonempty, get the instruction at the top of the semantic stack, and execute it according to its semantic rule
- Each instruction of the kernel language has a rule that defines its execution
- (Note: When we introduce concurrency, we will extend this algorithm to run with more than one semantic stack)

39

39

Semantic rules for kernel language instructions

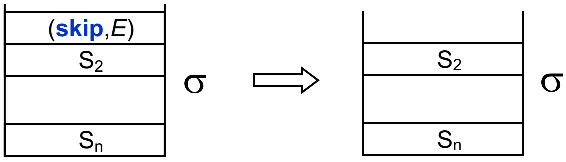
- For each instruction in the kernel language, we will define its rule in the abstract machine
- Each instruction takes one execution state as input and returns one execution state
  - Execution state (ST,σ) = semantic stack ST + memory σ
- Let's look at three instructions in detail:
  - **skip**
  - $<s>_1$ $<s>_2$ (sequential composition)
  - **local** <x> **in** <s> **end**
- We will see the others in less detail. You can learn about them in the exercises and in the book.

40

40

skip

- The simplest instruction
- It does nothing at all!
- Input state: ([(**skip**,*E*), S_2, …, S_n], σ)
- Output state: ([S_2, …, S_n], σ)
- That's all!

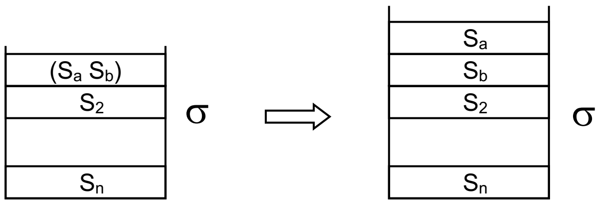


41

41

$(<s>_1\ <s>_2)$ (sequential composition)

- Almost as simple as **skip**
- The instruction removes the top of the stack and adds two new elements
- Input state: ([(S_a S_b), S_2, …, S_n], σ)
- Output state: ([S_a, S_b, S_2, …, S_n], σ)

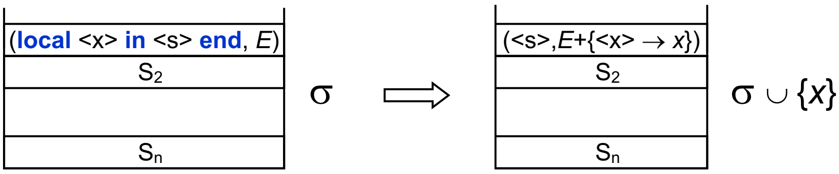


42

42

local <x> in <s> end

- Create a fresh new variable x in memory σ
- Add the pair {X $\rightarrow$ *x*} to the environment *E* (using adjunction operation)



Some other instructions

- <x>=<v> (value creation + assignment)
  - Note: when <v> is a procedure, you have to create the contextual environment
- **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end** (conditional)
  - Note: if <x> is unbound, the instruction will wait ("block") until <x> is bound to a value
  - The activation condition: "<x> is bound to a value"
- **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
  - Note: **case** statements with more patterns are built by combining several kernel instructions
- {<x> <y>$_1$ ... <y>$_n$}
  - Note: since procedure definition and procedure call are the foundation of data abstraction, we define them later!

Abstract machine concepts

- Single-assignment memory $\sigma = \{x_1=10, x_2, x_3=20\}$
  - Variables and the values they are bound to
- Environment $E = \{X \rightarrow x,\ Y \rightarrow y\}$
  - Link between identifiers and variables in memory
- Semantic instruction $(\langle s \rangle, E)$
  - An instruction with its environment
- Semantic stack $ST = [(\langle s \rangle_1, E_1), \ldots, (\langle s \rangle_n, E_n)]$
  - A stack of semantic instructions
- Execution state (ST, σ)
  - A pair of a semantic stack and a memory
- Execution $(ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow (ST_3, \sigma_3) \rightarrow \ldots$
  - A sequence of execution states

45

45

Semantics 4: Proving correctness with the semantics

46

When is a program correct?

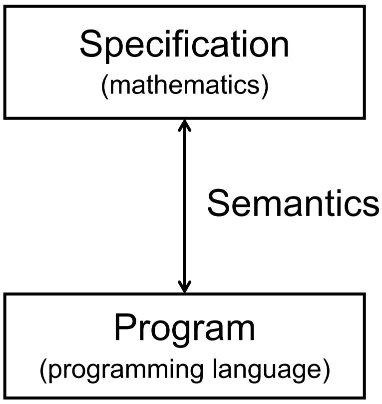
- "A program is correct when its execution gives the result we want"
  - How can we be sure?
- We need to make precise what is the result
  - We introduce the concept of specification
- We need to make precise what is an execution
  - We introduce the concept of semantics
- We need to prove that the program satisfies the specification, when it executes following the semantics

47

47

The three pillars

- The specification: the result we want
- The semantics connects these two: it lets us prove that executing the program gives the desired result
- The program: what will execute



48

48

Example: correctness of factorial

- The specification defining *n!* (mathematics)

  $0! = 1$

  $n! = n \times (n-1)!$ when $n>0$

- The semantics connects the two
  - Executing R={Fact N} following the semantics gives *r=n!*

- The program defining {Fact N} (programming language)

```
fun {Fact N}
    if N==0 then 1 else N*{Fact N-1} end
end
```


49

Mathematical induction

- To do this proof for a recursive function we need to use mathematical induction
  - A recursive function calculates on a recursive data structure, which has a base case and a general case
  - We first show the correctness for the base case
  - We then show that if the program is correct for a general case, it is correct for the next case

- For integers, the base case is usually 0 or 1, and the general case *n-1* leads to the next case *n*

- For lists, the base case is usually nil or a small list [X], and the general case T leads to the next case H|T

50

The inductive proof

- We must show that {Fact N} calculates *n!* for all *n*≥0
- Base case: *n=0*
  - The specification says: *0!=1*
  - The execution of {Fact 0}, *using the semantics*, gives {Fact 0}=1
    - *It's correct!*
- General case: *(n-1)* → *n*
  - The specification says: *n!* = *n* × *(n-1)!*
  - The execution of {Fact N}, *using the semantics*, gives {Fact N} = *n* × {Fact N-1}
    - We assume that {Fact N-1}=*(n-1)!* (induction hypothesis)
    - We assume that the language correctly implements multiplication "×"
    - Therefore: {Fact N} = *n*×{Fact N-1} = *n*×*(n-1)!* = *n!*
    - *It's correct!*
- Now we just need to understand the magic words "*using the semantics*"!

51

51

How to execute a program *using the semantics*

- We execute the program using the semantics in two steps
- First step: translate the program into kernel language
  - The kernel language is a simple language that has all essential concepts
  - All programs can be translated into kernel language
  - → We translate the definition of Fact into kernel language
- Second step: execute the translated program on the abstract machine
  - The abstract machine is a simplified computer with a precise definition
  - → We execute {Fact 0 R} and {Fact N R} on the abstract machine

52

52

Executing Fact using the semantics

- We need to execute both {Fact 0} and {Fact N} using the semantics
- First we translate the definition of Fact into kernel language:

```
proc {Fact N R}
    local B in
        B=(N==0)
        if B then R=1
        else local N1 R1 in
            N1=N-1
            {Fact N1 R1}
            R=N*R1
          end
        end
    end
end
```

There are a few mistakes in this translation! Can you find them?

53

53

Executing Fact using the semantics

- Here is the correct translation:

```
proc {Fact N R}
    local B in
        local Z in Z=0 B=(N==Z) end
        if B then R=1
        else local N1 in
            local R1 in
              local U in U=1 N1=N-U end
              {Fact N1 R1}
              R=N*R1
            end
          end
        end
    end
end
```

54

54

Execution of {Fact 0} (1)

- Let's first look at the function call {Fact 0}
- We execute the procedure call {Fact Z F} where Z=0
- We need a memory σ and an environment *E*:

  σ = {*fact*=(**proc** {$ N R} … **end**,{Fact→*fact* }), *n*=0, *r* }
  E = {Fact→*fact*, Z→*n*, F→*r* }

- Here is what we will execute: (initial execution state)

  ([({Fact Z F}, *E*)] , σ)

55

55

Execution of {Fact 0} (2)

- To execute {Fact Z F} we replace it by the procedure body and we replace the calling environment by a new environment
- The call:

  ([({Fact Z F}, {Fact→*fact*, Z→*n*, F→*r* })], σ)

  Calling environment {Fact, Z, F}

  is replaced by the body of {Fact N R}:

  Calling environment {Fact, Z, F} is replaced by definition environment {Fact, N, R}. Later we will see the general rule how to do this for any procedure call.

```
( [(local B in
      B=(N==0)
      if B then R=1 else … end
   end, {Fact→fact, N→n, R→r })], σ )
```

  Definition environment {Fact, N, R}

Definition environment is different from the calling environment!

56

56

Execution of {Fact 0} (3)

- To execute the **local** instruction:

  ([(**local** B **in**
  B=(N==0)
  if B **then** R=1 **else** … **end**
  end, {Fact→*fact*, N→*n*, R→*r* })], σ)

  we do two operations:
  - We extend the memory with a new variable *b*
  - We extend the environment with {B → *b*}

- We then replace the instruction by its body:

  ([(B=(N==0)
  if B **then** R=1 **else** … **end**,
  {Fact→*fact*, N→*n*, R→*r*, B→*b*})], σ∪{*b*})

57

Execution of {Fact 0} (4)

- We now do the same for:
  B=(N==0)
  and:
  if B **then** R=1 **else** … **end end**
- This will first bind *b*=**true** and then bind *r=1*
- This completes the execution of {Fact 0}
- We have executed {Fact 0} with the semantics and shown that the result is 1
- To complete the proof, we still have to show that the result of {Fact N} is the same as N*{Fact N-1}

58

We have proved the correctness of Fact

- Let's recapitulate the approach
- Start with the specification and program of Fact
  - We want to prove that the program satisfies the specification
  - Since the function is recursive, our proof uses mathematical induction
- We need to prove the base case and the general case:
  - Prove that {Fact 0} execution gives 1
  - Prove that {Fact N} execution gives n × *(result of* {Fact N-1} *execution)*
- We prove both cases using the semantics
  - To use the semantics, we first translate Fact into kernel language, and then we execute on the abstract machine
- This completes the proof

59

59

Semantics 5: Procedures

60

Procedures

- Procedures are very important, since they are the foundation of all data abstraction
  - Higher-order programming
  - Layered program organization (libraries calling libraries)
  - Encapsulation (hiding the implementation)
  - Object-oriented programming (objects and classes)
  - Abstract data types
  - Component-oriented programming (packages, modules)
  - Multi-agent programming (agents sending messages)
- This is why we show them separately

61

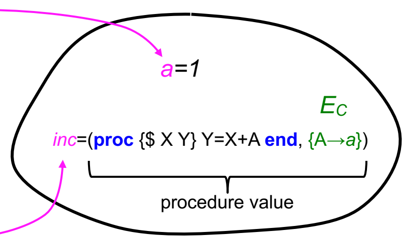
61

We recall how procedures are stored in memory

Program ⟹ Program (kernel language) — Memory

```
local A Inc in
   A=1
   fun {Inc X}
      X+A
   end
end
```

```
local A in local Inc in
   A=1
   Inc=proc {$ X Y}
      Y=X+A
   end
end end
```



E = {A→a,Inc→inc}

62

62

Defining and calling procedures

- Defining a procedure
  - Create the contextual environment
  - Store the procedure value, which contains both procedure code and contextual environment
- Calling a procedure
  - Create a new environment by combining two parts:
    - The procedure's contextual environment
    - The formal arguments (identifiers in the procedure definition), which are made to reference the actual argument values (at the call)
  - Execute the procedure body with this new environment
- We first give an example execution to show what the semantic rules have to do

63

63

Procedure call example (1)

```
local Z in
  Z=1
  proc {P X Y} Y=X+Z end
end
```

- The free identifiers of the procedure (here, just Z) are the identifiers declared outside the procedure
- When executing P, the identifier Z must be known
- Z is part of the procedure's contextual environment, which must be part of the procedure value

64

64

Important slide

Procedure call example (2)

```
local P in
   local Z in
      Z=1
      proc {P X Y} Y=X+Z end  % EC = {Z→z}
   end
   local B A in
      A=10
      {P A B}        % P's body Y=X+Z must do b=a+z
      {Browse B}  % Therefore: EP = {Y→b, X→a, Z→z}
   end
end
```

65

65

Semantic rule for procedure definition

- Semantic instruction:
  $(<x>=\mathbf{proc}\ \{\$\ <x>_1 \ldots <x>_n\}\ <s>\ \mathbf{end},\ E)$
  - Formal arguments:
    $<x>_1, \ldots, <x>_n$
  - Free identifiers in <s>:
    $<z>_1, \ldots, <z>_k$
  - Contextual environment:
    $E_C=E|_{<z>1,\ \ldots,\ <z>k}$ (restriction of *E* to free identifiers)
- Create the following binding in memory:
  $x=(\mathbf{proc}\ \{\$\ <x>_1 \ldots <x>_n\}\ <s>\ \mathbf{end},\ E_C)$

66

66

Semantic rule for procedure call (1)

- Semantic instruction:

  ({⟨x⟩ ⟨y⟩$_1$ … ⟨y⟩$_n$}, *E*)

- If the activation condition is false (*E*(⟨x⟩) unbound)
  - Suspension (do not execute, wait until *E*(⟨x⟩) is bound)
- If *E*(⟨x⟩) is not a procedure
  - Raise an error condition (an exception, see later)
- If *E*(⟨x⟩) is a procedure with the wrong number of arguments ($\neq n$)
  - Raise an error condition (an exception, see later)

67

67

Semantic rule for procedure call (2)

Most important slide of the course

- Semantic instruction on stack:

  ({⟨x⟩ ⟨y⟩$_1$ … ⟨y⟩$_n$}, *E*)

  with procedure definition in memory:

  E(⟨x⟩) = (**proc** {$ ⟨z⟩$_1$…⟨z⟩$_n$} ⟨s⟩ **end**, E_C)

- Put the following instruction on the stack:

  (⟨s⟩, E_C + {⟨z⟩$_1 \rightarrow$ *E*(⟨y⟩$_1$), …, ⟨z⟩$_n \rightarrow$ *E*(⟨y⟩$_n$)})

68

68

Computing with environments

- The abstract machine does two kinds of computations with environments
- Adjunction: E_2=E_1+{X → y}
  - Add a pair (identifier→variable) to an environment
  - Overrides the same identifier in E_1 (if it exists)
  - Needed for **local** <x> **in** <s> **end** (and others)
- Restriction: E_C=$E_{|\{X,Y,Z\}}$
  - Limit identifiers in an environment to a given set
  - Needed to calculate the contextual environment

Adjunction

- For a **local** instruction

```
local X in (E1)
    X=1
    local X in (E2)
        X=2
        {Browse X}
    end
end
```

- E_1 = {Browse → b, X → x}
- E_2 = E_1 + {X → y} = {Browse → b, X → y}

Restriction

- For a procedure declaration

```
local A B C AddB in
    A=1 B=2 C=3 (E)
    fun {AddB X} (E_C: contextual environment)
        X+B
    end
end
```

- $E = \{A \to a, B \to b, C \to c, AddB \to a'\}$
- $E_C = E_{|\{B\}} = \{B \to b\}$

71

71

Semantics summary

72

Bringing it all together

- Defining the semantics brings many concepts together
  - Concepts we have seen before: identifier, variable, environment, memory, instruction, kernel language
  - New concepts: procedure value, semantic instruction, semantic stack, semantic rule, execution state, execution, abstract machine
- We give semantic rules for kernel language instructions, to show how they execute in the abstract machine
- We use the semantics to prove program correctness, by using it as bridge between specification and program

Specification ←Semantics→ Program

Mathematical formula — *Programming language text*

73

73

Why semantics is important

- Semantics is at the heart of programming
  - Software development means to extend the system's semantics: a new library is like extending the language with new instructions
- When you write a piece of software, you should design its semantics
  - The semantics should be simple and complete
  - Users don't need to understand the semantics: its existence is enough
    - Existence of a simple semantics means no unpleasant surprises!
- « Semantics is the ultimate programming language »
  - Invariants are the ultimate loop (invariant programming)
  - Data abstractions are like new kernel language instructions

74

74

Using the semantics

- Semantics has many uses:
  - For design (ensuring the design is simple and predictable)
  - For understanding (the nooks and crannies of programs)
  - For verification (correctness and termination)
  - For debugging (a bug is only a bug with respect to a correct execution)
  - For visualization (a visual representation must be correct)
  - For education (pedagogical uses of semantics)
  - For program analysis and compiler design
- We don't need to bring in details of the processor architecture or compiler in order to understand many things about programs
  - For example, our semantics can be used to understand garbage collection
  - We will use the semantics when needed in the rest of the course

Discrete mathematics

- The abstract machine is built with discrete mathematics
- It is probably the most complex construction that you have seen built with discrete mathematics!
  - Engineering students are quite used to integrals, differential equations, and complex analysis, which are all continuous mathematics, and the abstract machine is not like this because it is discrete!
- Discrete mathematics is important because that's how computing systems work (both software and hardware)
  - Surprising behavior and bugs become less surprising if you understand the discrete mathematics of computing systems
  - Too often, continuous models are used for computing systems
  - All this applies to the real world as well (beyond computing systems)

“Semantics is the ultimate programming language”

77

77