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

Lecture 2 & 3

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview of lecture 2 & 3

- Refresher of lecture 1
- Symbolic programming
  - Lists
  - Pattern matching
  - Trees
  - Tuples and records
- Formal semantics
  - Kernel language
  - Abstract machine
  - Proving correctness of programs
  - Semantic rules for kernel instructions
  - Semantics of procedures

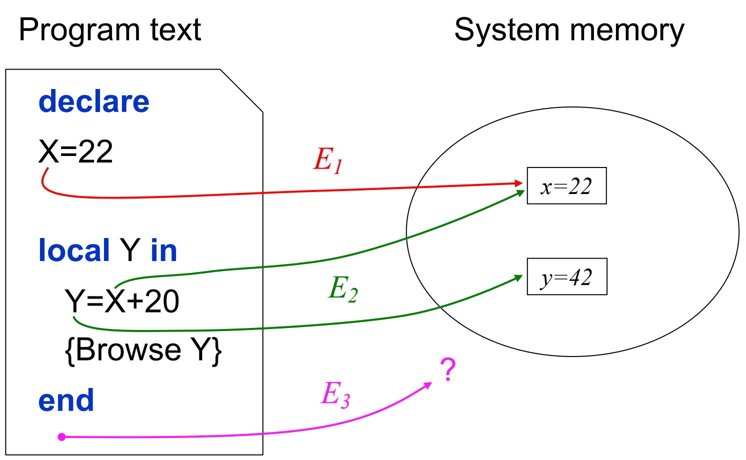
2

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 $X \rightarrow x$
    - Identifier X, memory variable x
- Example environment E_2
  - $E_2=\{X \rightarrow x, Y \rightarrow y\}$
  - $E_2(X)=x$
  - $E_2(Y)=y$

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→*a*}
  - Where *a* is a variable in memory: *a=1*

10

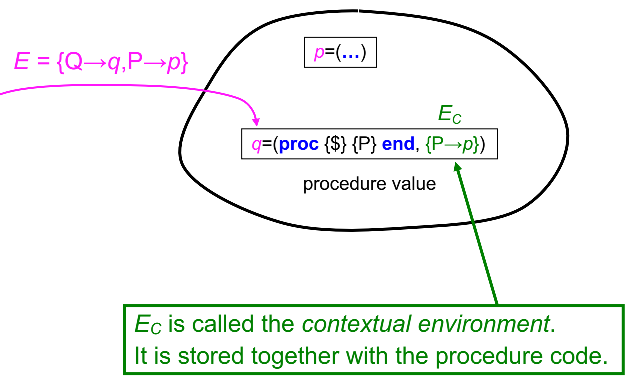
10

How procedure Q is stored in memory

Program text

System memory

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



Procedure values

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

$$inc = (\underbrace{\textbf{proc}\ \{\$\ X\ Y\}\ Y{=}X{+}A\ \textbf{end}}_{\text{Procedure code}},\ \underbrace{\{A \rightarrow a\}}_{\text{Contextual environment}})$$

(without a name: '$' replaces the identifier)

- 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

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 value p
  - This is guaranteed to always get the right value, even if there is another definition of P right next to the call of Q
- The identifiers in E_c are the identifiers inside Q that are defined outside of Q
  - They are called the free identifiers of Q

Free identifiers

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

- The instruction:
  local Q **in**
  proc {Q A} {P A+1} **end**
  end
  has one free identifier:
  {P}
- The instruction:
  local Z **in** Z=X+Y **end**
  has two free identifiers:
  {X,Y}

Lists

15

Definition of a list

- A list is a recursive type: defined in terms of itself
  - Recursion is used both for computations and data!
  - We also use recursion for functions on lists
- A list is either an empty list or a pair of an element followed by another list
  - This definition is recursive because it defines lists in terms of lists. There is no infinite regress because the definition is used constructively to build larger lists from smaller lists.
- Let's introduce a formal notation

16

16

Syntax definition of a list

- Using an EBNF grammar rule we write:

  <List T> ::= nil | T ‘|’ <List T>

- This defines the textual representation of a list
- EBNF = Extended Backus-Naur Form
  - Invented by John Backus and Peter Naur
  - <List T> represents a list of elements of type T
  - T represents one element of type T
- Be careful to distinguish between | and ‘|’ : the first is part of the grammar notation (it means “or”), and the second is part of the syntax being defined

17

Some examples of lists

- According to the definition (if T is integer type):

  nil
  10 | nil
  10 | 11 | nil
  10 | 11 | 12 | nil
  10 | 11 | 12 | 13 | nil

18

Type notation

- <Int> represents an integer; more precisely, it is the set of all syntactic representations of integers
- <List <Int>> represents the set of all syntactic representations of lists of integers
- T represents the set of all syntactic representations of values of type T; we say that T is a type variable
  - Do not confuse a type variable with an identifier or a variable in memory! Type variables exist only in grammar rules.

19

19

Don’t confuse a thing and its representation

René Magritte, *La trahison des images*, 1928-29, oil, Los Angeles County Museum of Art, Los Angeles.

- This is not a pipe.
  It is a digital display of a photograph of a painting of a pipe (thanks to Belgian surrealist René Magritte for pointing this out!).

1234

- This is not an integer.
  It is a digital display of a visual representation of an integer using numeric symbols in base 10.

20

20

Representations for lists

- The EBNF rule gives one textual representation
  - <List <Int>> ⇒
    10 | <List <Int>> ⇒
    10 | 11 | <List <Int>> ⇒
    10 | 11 | 12 | <List <Int>> ⇒
    10 | 11 | 12 | nil

We repeatedly replace the left-hand side of the rule by a possible value, until no more can be replaced

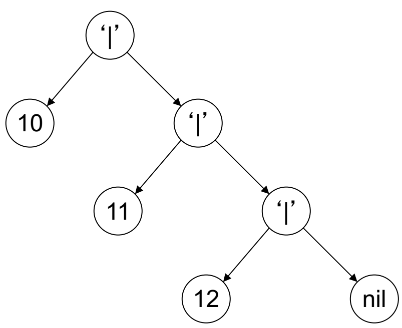
- Oz allows another textual representation
  - Bracket notation: [10 11 12]
  - In memory, [10 11 12] is identical to 10 | 11 | 12 | nil
  - Different textual representations of the same thing are called syntactic sugar

21

21

Graphical representation of a list

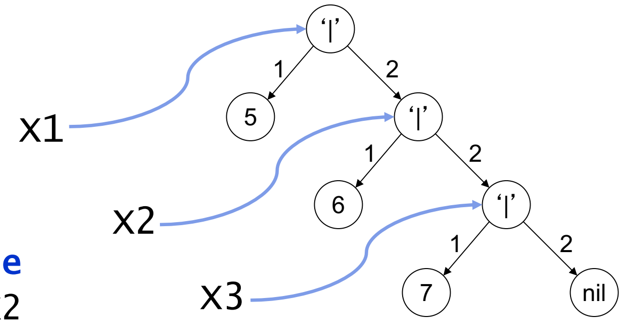
- Graphical representations are very useful for reasoning
  - Humans have very powerful visual reasoning abilities
- We start from the leftmost pair, namely 10 | <List <Int>>
  - We draw three nodes with arrows between them
  - We then replace the node <List <Int>> as before
- This is an example of a more general structure called a tree



22

22

Building a list incrementally




```
declare
X1=5|X2
X2=6|X3
X3=7|nil
```


Computing with lists

- A non-empty list is a pair of head and tail
- Accessing the head:

  X.1
- Accessing the tail:

  X.2
- Comparing the list with nil:

  if X==nil **then** … **else** … **end**

Head and tail functions

- We can define functions

```
fun {Head Xs}
  Xs.1
end

fun {Tail Xs}
  Xs.2
end
```

25

25

Example with Head and Tail

- {Head [a b c]}
  returns a
- {Tail [a b c]}
  returns [b c]
- {Head {Tail {Tail [a b c]}}}
  returns c

- Draw the graphical picture of [a b c]!

26

26

Functions on lists

27

Functions that create lists

- Let us now define a function that outputs a list
  - We will use both pattern matching and recursion, as before, but this time the output will also be a list
  - We will define the Sum function to compute the sum of elements of a list
  - We give first the naïve version and then the smart version (based on invariants)

28

28

Sum of list elements

- We are given a list of integers
- We would like to calculate their sum
  - We will define the function “Sum”
- Inductive definition following the list structure
  - Sum of an empty list: 0
  - Sum of a non-empty list L: {Head L} + {Sum {Tail L}}

29

29

Sum of list elements (naïve method)

```
fun {Sum L}
  if L==nil then
    0
  else
    {Head L} + {Sum {Tail L}}
  end
end
```

30

30

Sum of list elements (with accumulator)

```
fun {Sum2 L A}
   if L==nil then
      A
   else
      {Sum2 {Tail L} A+{Head L}}
   end
end
```

What is the invariant?

31

31

Another example: Nth function

- Define the function {Nth L N} which returns the nth element of L
- The type of Nth is:
  <fun {$ <List T> <Int>}:<T>>
- Reasoning:
  - If N==1 then the result is {Head L}
  - If N>1 then the result is {Nth {Tail L} N-1}

32

32

The Nth function

- The complete definition:

```
fun {Nth L N}
   if N==1 then {Head L}
   elseif N>1 then
      {Nth {Tail L} N-1}
   end
end
```

- What happens if the nth element does not exist?

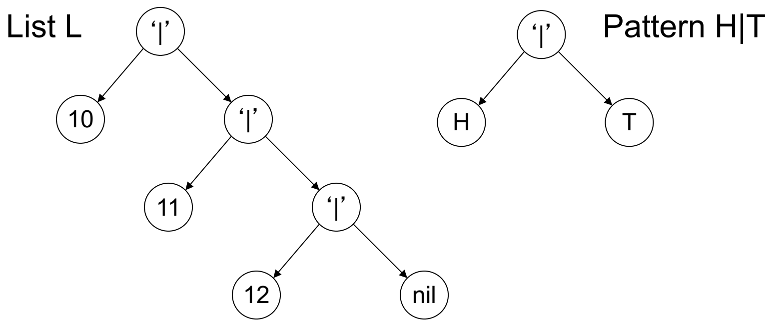
33

33

Pattern matching

34

Pattern matching

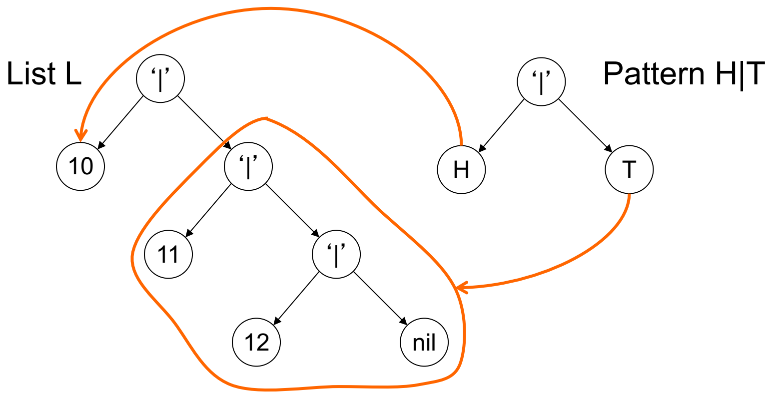


- **case** L **of** H|T **then** ... **else** ... **end**

35

35

Pattern matching



- **case** L **of** H|T **then** ... **else** ... **end**
- H=10, T=11|12|nil

36

36

Sum with pattern matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```

37

37

Sum with pattern matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```

A clause

- "nil" is the *pattern* of the clause

38

38

Sum with pattern matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```

A clause

- "H|T" is the *pattern* of the clause

39

39

Pattern matching

- The first clause uses **of**, the others use **[]**
- Clauses are tried in their textual order
- A clause matches if its pattern matches
- A pattern matches if its label and its arguments match
  - The identifiers in the pattern are assigned to their corresponding values in the input
- The first matching clause is executed, following clauses are ignored

40

40

Kernel language introduction

41

The kernel language

- The kernel language is the first part of the formal semantics of a programming language
  - The second part is the *abstract machine* which we will see later on
- Remember in lecture 1, we explained that each programming paradigm has a simple core language called its kernel language
  - We now introduce the kernel language of functional programming
- All programs in functional programming can be translated into the kernel language
  - All intermediate results of calculations are visible — Kernel principle
  - All functions become procedures with one extra argument
  - Nested function calls are unnested by introducing new identifiers

42

42

Length of a list

```
fun {Length Xs N}
   case Xs
   of nil then N
   [] X|Xr then {Length Xr N+1}
   end
end
```

43

43

Length of a list translated into kernel language

- The instruction **case** (with one pattern) is part of the kernel language:

```
proc {Length Xs N R}
  case Xs
  of nil then R=N
  else
     case Xs
     of X|Xr then
        local N1 in
           N1=N+1
           {Length Xr N1 R}
        end
     else
        raise typeError end /* type error: see later in the course! */
     end
  end
end
```

44

44

A function is a procedure with one extra argument

- The kernel language does not need functions
  - It's enough to have procedures
  - Factored design: each concept occurs only once
- A function is translated as a procedure with one extra argument, which gives the function's result
- N={Length L Z}
  is equivalent to:
  {Length L Z N}

45

45

Translating to kernel language

- All practical programs can be translated into kernel language
- How to translate:
  - Only kernel language instructions can be used
  - The consequence is that all « hidden » variables become visible
    - Functions become procedures with one extra argument
    - Nested expressions become sequences, with extra local identifiers
    - Each pattern has its own case statement
  - The kernel language is a subset of Oz!
    - It can be executed in Mozart
- Consequences:
  - Kernel programs are longer
  - It is easy to see when programs are tail-recursive
  - It is easy to see exactly how programs execute

46

46

Kernel language of the functional paradigm (so far)

- <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**

This is almost complete!

- <v> ::= <number> | <list> | **...**

still something missing

- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

47

47

Trees

48

Trees

- Trees are the second most important data structure in computing, next to lists
  - Trees are extremely useful for efficiently organizing information and performing many kinds of calculations
- Trees illustrate well goal-oriented programming
  - Many tree data structures are based on a global property, that must be maintained during the calculation
- In this lesson we will define trees and use them to store and look up information
  - We will define ordered binary trees and algorithms to add information, look up information, and remove information

49

49

Trees

- A tree is a recursive structure: it is either an empty tree (called a leaf) or an element and a set of trees

root node

<tree T> ::= leaf | t(T <tree T> ... <tree T>)

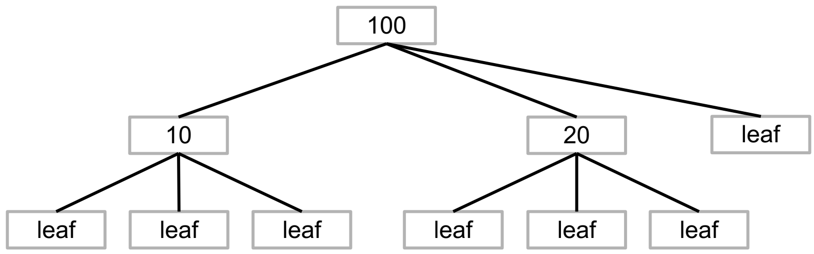
subtrees

50

50

Example tree

- **declare**
  T=t(100 t(10 leaf leaf leaf) t(20 leaf leaf leaf) leaf)



51

51

Trees compared to lists

- A tree is a recursive structure: it is either an empty tree (called a leaf) or an element and a set of trees

  <tree T> ::= leaf | t(T <tree T> ... <tree T>)

  <list T> ::= nil | '|'(T <list T>)

Notice the similarity with lists!

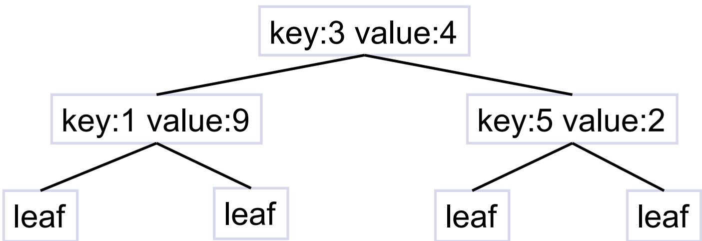
52

52

Ordered binary tree (1)

- <obtree T> ::= leaf
  | tree(key:T value:T left:<obtree T> right:<obtree T>)

- Binary: each non-leaf tree has two subtrees (named left and right)
- Ordered: for each tree (including all subtrees):
  all keys in the left subtree < key of the root
  key of the root < all keys in the right subtree



53

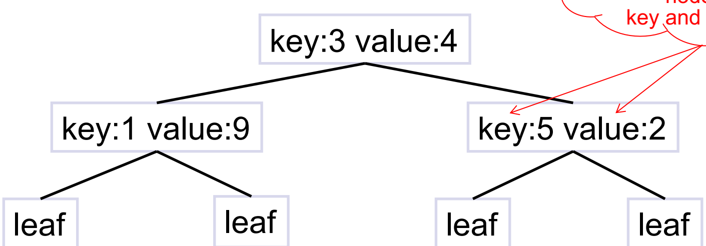
53

Ordered binary tree (2)

- <obtree T> ::= leaf
  | tree(key:T value:T left:<obtree T> right:<obtree T>)

- Binary: each non-leaf tree has two subtrees (named left and right)
- Ordered: for each tree (including all subtrees)
  all keys in the left subtree < key of the root
  key of the root < all keys in the right subtree

This tree has two information fields at each node: key and value

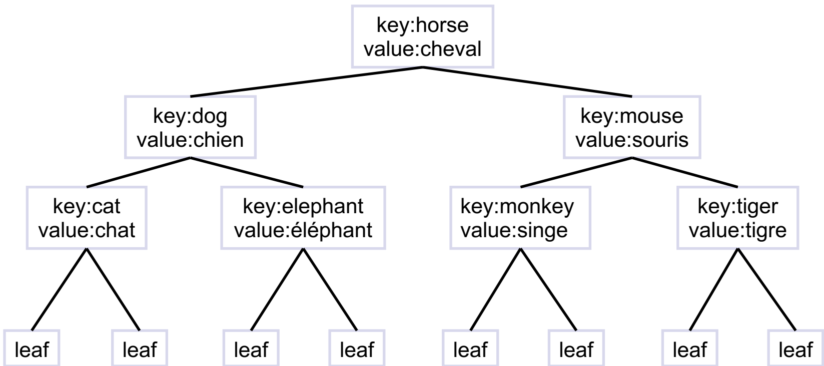


54

54

Ordered binary tree (3)

- This ordered binary tree is a translation dictionary from English to French

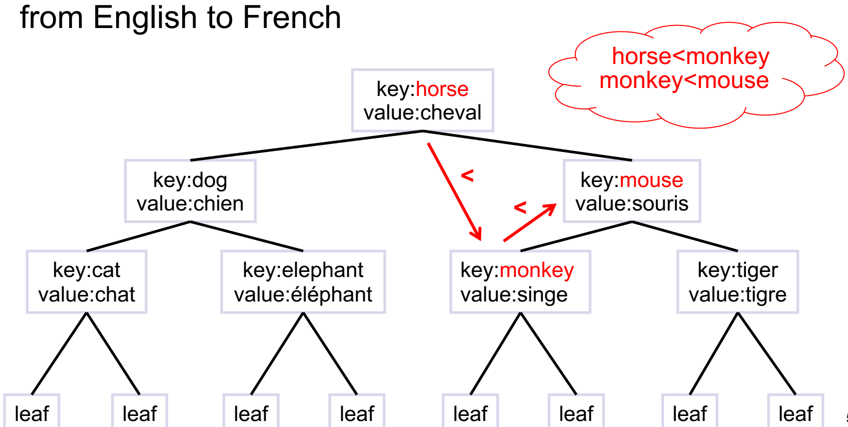


55

55

Ordered binary tree (4)

- This ordered binary tree is a translation dictionary from English to French



56

56

Search tree

- Search tree: A tree that is used to organize information, and with which we can perform various operations such as looking up, inserting, and deleting information

- Let's define these three operations:
  - {Lookup K T}: returns the value V corresponding to key K
  - {Insert K W T}: returns a new tree with added (K,W)
  - {Delete K T}: returns a new tree that does not contain K

57

57

Looking up information

- There are four possibilities:
- K is not found
- K is found
- K might be in the left subtree
- K might be in the right subtree

```
fun {Lookup K T}
   case T
   of leaf then notfound
   [] tree(key:Y value:V T1 T2) andthen K==Y then
         found(V)
   [] tree(key:Y value:V T1 T2) andthen K<Y then
         {Lookup K T1}
   [] tree(key:Y value:V T1 T2) andthen K>Y then
         {Lookup K T2}
   end
end
```

58

58

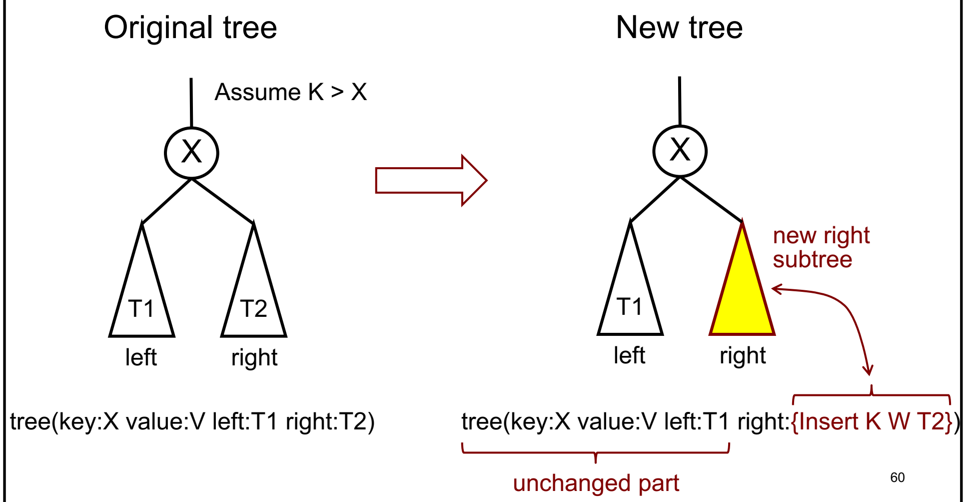
Efficiency of Lookup

- How efficient is the Lookup function?
  - If there are n words in the tree, and each node's subtrees are approximately equal in size (we say the tree is balanced), then the average lookup time is proportional to $\log_2 n$
  - Tree lookup is much more efficient than list lookup: if for 1000 words the average time is 10, then for 1000000 words this will increase to 20 (instead of being multiplied by 1000)
- If the tree is not balanced, say all the right subtrees are very small, then the time will be much larger
  - In the worst case, the tree will look like a list
- How can we arrange for the tree to be balanced?
  - There exist algorithms for balancing an unbalanced tree, but if we insert words randomly, then we can show that the tree will be approximately balanced, good enough to achieve logarithmic time

59

59

Inserting a new key/value pair



60

60

Inserting information

- There are four possibilities:
- (K,W) replaces a leaf node
- (K,W) replaces an existing node
- (K,W) is inserted in the left subtree
- (K,W) is inserted in the right subtree

```
fun {Insert K W T}
   case T
   of leaf then tree(key:K value:W leaf leaf)
   [] tree(key:Y value:V T1 T2) andthen K==Y then
         tree(key:K value:W T1 T2)
   [] tree(key:Y value:V T1 T2) andthen K<Y then
         tree(key:Y value:V {Insert K W T1} T2)
   [] tree(key:Y value:V T1 T2) andthen K>Y then
         tree(key:Y value:V T1 {Insert K W T2})
   end
end
```

61

61

Deleting information

- There are four possibilities:
- (K,_) is not in the tree
- (K,_) is removed immediately
- (K,_) is removed from the left subtree
- (K,_) is removed from the right subtree
- Right?

```
fun {Delete K T}
   case T
   of leaf then leaf
   [] tree(key:Y value:W T1 T2) andthen K==Y then
         leaf
   [] tree(key:Y value:W T1 T2) andthen K<Y then
         tree(key:Y value:W {Delete K T1} T2)
   [] tree(key:Y value:W T1 T2) andthen K>Y then
         tree(key:Y value:W T1 {Delete K T2})
   end
end
```

62

62

Deleting information

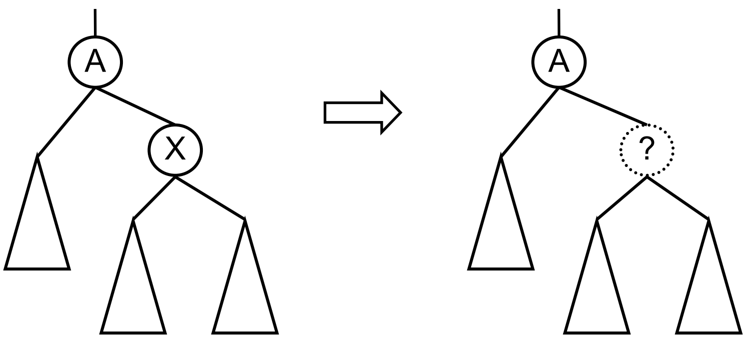
- There are four possibilities:
- (K,_) is not in the tree
- (K,_) is removed immediately
- (K,_) is removed from the left subtree
- (K,_) is removed from the right subtree
- Right? WRONG!

```
fun {Delete K T}
   case T
   of leaf then leaf
   [] tree(key:Y value:W T1 T2) andthen K==Y then
         leaf
   [] tree(key:Y value:W T1 T2) andthen K<Y then
         tree(key:Y value:W {Delete K T1} T2)
   [] tree(key:Y value:W T1 T2) andthen K>Y then
         tree(key:Y value:W T1 {Delete K T2})
   end
end
```

63

63

Deleting an element from an ordered binary tree

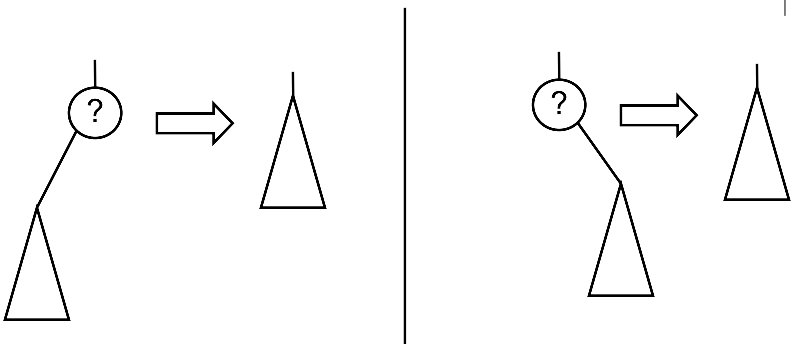


The problem is to repair the tree after X disappears

64

64

Deleting the root when one subtree is empty

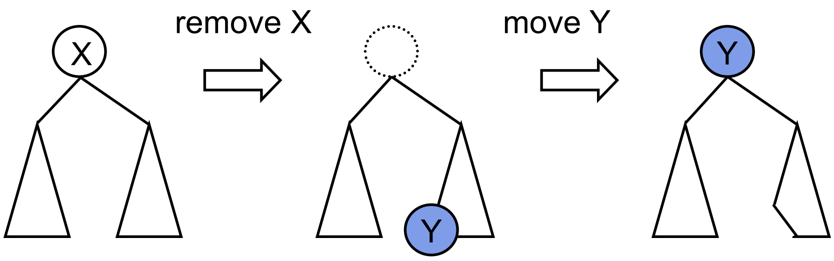


It's easy when one of the subtrees is empty:
just replace the tree by the other subtree

65

65

Deleting the root when both subtrees are not empty



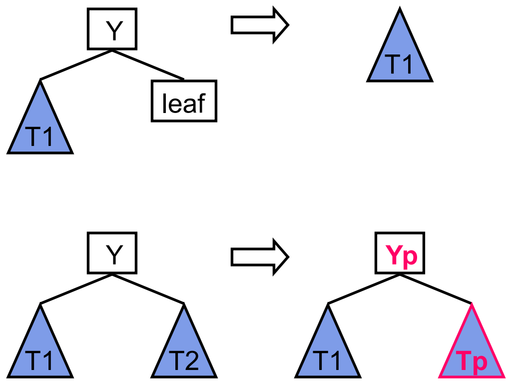
The idea is to fill the "hole" that appears after X is removed. We can put there the smallest element in the right subtree, namely Y.

66

66

Deleting the root

- To remove the root Y, there are two possibilities:
- One subtree is a leaf. Just return the other.
- Neither subtree is a leaf. Remove an element from one of its subtrees.

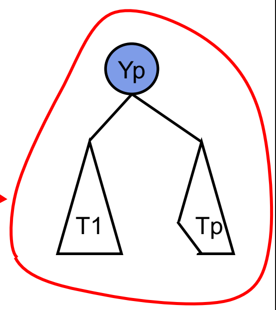


67

67

We need a new function: RemoveSmallest

```
fun {Delete K T}
   case T
   of leaf then leaf
   [] tree(key:X value:V left:T1 right:T2) andthen K==X then
         case {RemoveSmallest T2}
         of none then T1
         [] triple(Tp Yp Vp) then
             tree(key:Yp value:Vp left:T1 right:Tp)
         end
   [] ... end
end
```



- RemoveSmallest takes a tree and returns three values:
  - The new subtree Tp without the smallest element
  - The smallest element's key Yp
  - The smallest element's value Vp
- With these three values we can build the new tree where Yp is the root and Tp is the new right subtree

68

68

Recursive definition of RemoveSmallest

```
fun {RemoveSmallest T}
   case T
   of leaf then none
   [] tree(key:X value:V left:T1 right:T2) then
      case {RemoveSmallest T1}
      of none then triple(T2 X V)
      [] triple(Tp Xp Vp) then
        triple(tree(key:X value:V left:Tp right:T2) Xp Vp)
      end
   end
end
```

To understand this definition, draw diagrams with trees!

- RemoveSmallest takes a tree T and returns:
  - The atom none when T is empty
  - The record triple(Tp Xp Vp) when T is not empty

69

Delete operation is complex

- Why is the delete operation so complex?
- It is because the tree satisfies a global condition, namely it is ordered
- The delete operation has to work to keep this condition true
- Many tree algorithms depend on global conditions and must work to keep the conditions true
- The interesting thing about a global condition is that it gives the tree **a spark of life**: the tree behaves a bit like it is alive (« goal-oriented behavior»)
  - Living organisms have goal-oriented behavior

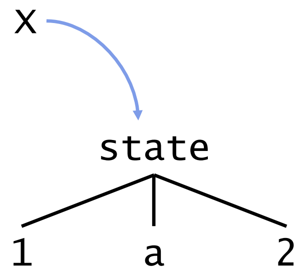
70

Goal-oriented programming

- Many tree algorithms depend on global properties and most of the work they do is in maintaining these properties
  - The ordered binary tree satisfies a global ordering condition. The insert and delete operations must maintain this condition. This is easy for insert, but harder for delete.
- Goal-oriented programming is widely used in artificial intelligence algorithms
  - It can give unexpected results as the algorithm does its thing to maintain the global property.
  - Goal-oriented behavior is characteristic of living organisms. So defining algorithms that are goal-oriented gives them a spark of life!

Tuples and records

Tuples



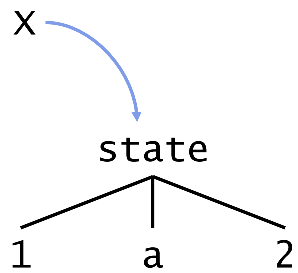
X=state(1 a 2)

- A tuple allows grouping several values together
  - For example: 1, a, 2
  - The position is meaningful: first, second, third!
- A tuple has a label
  - For example: `state`

73

73

Operations on tuples



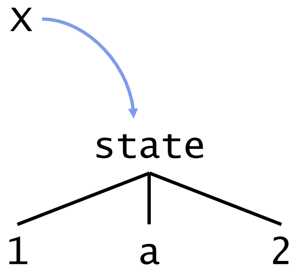
X=state(1 a 2)

- `{Label X}` returns *the label* of tuple `X`
  - For example: `state`
  - The label is a constant, called an atom
- `{Width X}` returns *the width* (number of fields)
  - For example: 3
  - Always a positive integer or zero

74

74

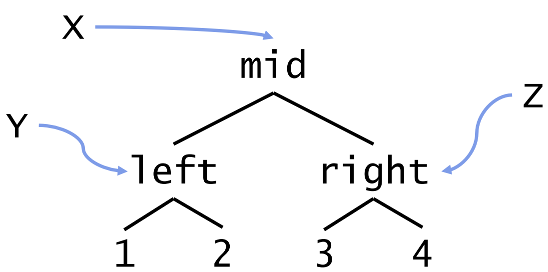
Accessing fields ("." operation)



X=state(1 a 2)

- Fields are numbered from `1` up to `{Width X}`
- `X.N` returns the nth field of tuple X:
  - `X.1` returns `1`
  - `X.3` returns `2`
- In the expression `X.N`, N is called the field name or "feature"

Building a tree



- A tree can be built with tuples:

```
declare
Y=left(1 2) Z=right(3 4)
X=mid(Y Z)
```

Testing equality (==)

- Equality testing with a number or atom
  - Easy: the number or atom must be the same
- Equality testing of trees
  - Also easy: the two trees must have the same root tuples and the same subtrees in corresponding fields
  - Careful when the tree has a cycle!
    - Comparison with == works, but naïve programs may loop
    - Advice: avoid this kind of tree

77

77

Tuples summary

- Tuple
  - Label
  - Width
  - Field
  - Field name, feature
- Accessing fields with "." operation
- Build trees with tuples
- Pattern matching with tuples
- Comparing tuples with "=="

78

78

A list is a tuple

- The list H|T is actually a tuple ‘|’ (H T)
- Principle of simplicity in the kernel language: instead of two concepts (tuples and lists), only one concept is needed (tuple)
- Because of their usefulness, lists have a syntactic sugar
  - It is purely for programmer comfort, it makes no difference in the kernel language

79

79

Syntax of lists as tuples

- A list is a special case of a tuple
- Prefix syntax (put the label ‘|’ in front)
  - nil
    ‘|’(5 nil)
    ‘|’(5 ‘|’(6 nil))
    ‘|’(5 ‘|’(6 ‘|’(7 nil)))
- Prefix syntax with field names
  - nil
    ‘|’(1:5 2:nil)
    ‘|’(1:5 2: ‘|’(1:6 2:nil))
    ‘|’(1:5 2: ‘|’(1:6 2: ‘|’(1:7 2:nil)))

80

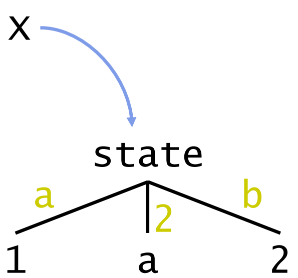
80

Records

- A record is a generalization of a tuple
  - Field names can be atoms (i.e., constants)
  - Field names can be any integer
    - Does not have to start with 1
    - Does not have to be consecutive
- A record also has a label and a width

Records

```
X=state(a:1 2:a b:2)
```



- The position of a field is no longer meaningful
  - Instead, it is the field name that is meaningful
- Accessing fields is done the same as for tuples
  - `X.a=1`

Record operations

- Label and width operations:
  - {Label X}=state
  - {Width X}=3
- Equality test:
  - X==state(a:1 b:2 2:a)
- New operation: arity
  - Returns a list of field names
  - {Arity X}=[2 a b] (in lexicographic order)
  - Arity also works for tuples and lists!

83

83

A tuple is a record

- The record:

  ```
  X = state(1:a 2:b 3:c)
  ```

  is the same as the tuple:

  ```
  X = state(a b c)
  ```
- In a tuple, all fields are numbered consecutively from 1
- What happens if we write:

  ```
  X = state(a 2:b 3:c)
  or
  X = state(2:b 3:c a)
  ```
- In a record, all unnamed fields are numbered consecutively starting with 1

84

84

A list is a tuple and a tuple is a record ⇒ many list syntaxes

- The list syntax
  ```
  X1=5|6|7|nil
  ```
  is a short-cut for
  ```
  X1=5|(6|(7|nil))
  ```
  which is a short-cut for
  ```
  X1='|'(5 '|'(6 '|'(7 nil)))
  ```
  which is a short-cut for
  ```
  X1='|'(1:5 2:'|'(1:6 2:'|'(1:7 2:nil)))
  ```
- The shortest syntax (the 'nil' is implied!)
  ```
  X1=[5 6 7]
  ```

85

85

The kernel language has only records

- In the kernel language there are only records
  - An atom is a record whose width is 0
  - A tuple is a record whose field names are numbered consecutively starting from 1
    - If this condition is not satisfied, the data structure is still a record but it is no longer a tuple
  - A list is built with tuples nil and '|'(X Y)
- This keeps the kernel language simple
  - It has just one data structure

86

86

Kernel language with 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**
  | **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> | <record> | …
- <number> ::= <int> | <float>
- <record>, <p> ::= <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

Records replace lists

87

87

Exercises

- Which of these records are tuples?
  - A=a(1:a 2:b 3:c)
  - B=a(1:a 2:b 4:c)
  - C=a(0:a 1:b 2:c)
  - D=a(1:a 2:b 3:c d)
  - E=a(a 2:b 3:c 4:d)
  - F=a(2:b 3:c 4:d a)
  - G=a(1:a 2:b 3:c foo:d)
  - H= '|' (1:a 2:' |' (1:b 2:nil))
  - I= '|' (1:a 2:' |' (1:b 3:nil))

88

88

Introduction to formal semantics

89

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

90

90

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

Different approaches to define language semantics

- Four general approaches have been invented:
  - This course → Operational semantics: Explains a program in terms of its execution on a rigorously defined abstract machine
    - This works for all paradigms!
  - LEPL1402 → 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 programming languages
  - 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 programming languages such as Prolog and constraint programming
- We will focus on operational semantics

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 full kernel language for functional programming
  2. Executing an example program on the abstract machine
  3. Defining the abstract machine and its semantic rules
  4. Proving the correctness of an example program
  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.

93

93

Semantics 1: Full kernel language

94

Kernel language of functional programming

- We have seen all concepts of functional programming
  - Now we can define its full 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

95

95

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

96

96

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)

97

97

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 "$".

98

98

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

99

99

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.

100

100

Semantics 2: Executing with the abstract machine

101

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

102

102

The example program in kernel language

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

103

103

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

$([(\langle s \rangle, E)], \sigma)$

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***

104

104

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 → *x*} so that X can refer to the new variable *x*

105

105

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→*b* to its environment, giving {B → *b*, X → *x*}

106

106

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

107

107

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

108

108

The conditional 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

- (Note: 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)

109

109

The *X=1* instruction

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

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

110

110

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

111

111

Semantics 3: The abstract machine

112

All 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 (<s>,E)
  - An instruction with its environment
- Semantic stack ST = [(<s>$_1$,E_1), …, (<s>$_n$,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

113

113

Abstract machine execution algorithm

-
```
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)

114

114

Semantic rules for kernel language instructions

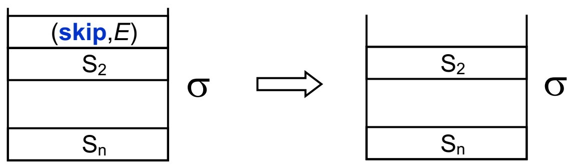
- For each instruction in the kernel language, we will define its rule in the abstract machine
- Each instruction takes one execution state as input and returns one execution state
  - Execution state = semantic stack ST + memory σ
- Let's look at three instructions in detail:
  - **skip**
  - <s>$_1$ <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.

115

115

skip

- The simplest instruction
- It does nothing at all!
- Input state: $([(\textbf{skip},E), S_2, \ldots, S_n], \sigma)$
- Output state: $([S_2, \ldots, S_n], \sigma)$
- That's all

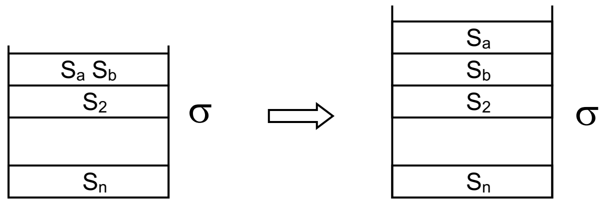


116

116

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

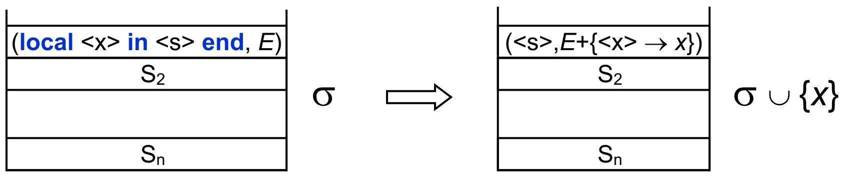
- Almost as simple as **skip**
- The instruction removes the top of the stack and adds two new elements
- Input state: $([(S_a\ S_b), S_2, \ldots, S_n], \sigma)$
- Output state: $([S_a, S_b, S_2, \ldots, S_n], \sigma)$



117

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)



118

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 will take a special look!

119

119

Semantics 4: Proving correctness with the semantics

120

When is a program correct?

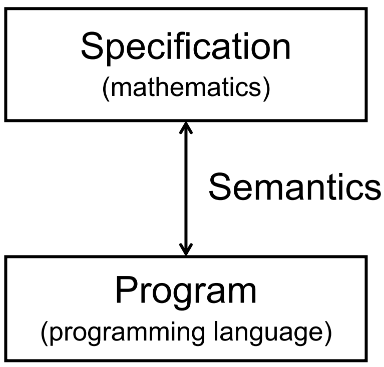
- "A program is correct when it does what we want"
  - How can we be sure?
- We need to make precise what we want it to do:
  - We introduce the concept of specification
- We need to prove that the program satisfies the specification, when it executes according to the semantics

121

121

The three pillars

- The specification: what we want
- The program: what we have
- The semantics connects these two: proving that what we have executes according to what we want



122

122

Example: correctness of factorial

- The specification of {Fact N} (mathematics)

  $0! = 1$

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

- The program (programming language)

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

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

123

Mathematical induction

- To make 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, and the general case T leads to the next case H|T

124

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*"!

125

125

How to execute a program *using the semantics*

- We execute the program using the semantics by following two steps
- First, we 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, we 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

126

126

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 mistakes in this translation! Can you find them?

127

127

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 local U in U=1 R=U end
         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
```

128

128

Execution of {Fact 0} (1)

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

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

- Here is what we will execute:

  {Fact N R}, E, σ

Execution of {Fact 0} (2)

- To execute {Fact N R} we replace it by the procedure body and we replace the calling environment by a new environment
- The instruction:

  {Fact N R}, {Fact→*fact*, N→*n*, R→*r* }, σ

  (N,R: arguments of Fact call)

  is replaced by the instruction:

  Later on we will see how to replace the calling environment by a new environment inside the procedure body.

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

(N,R: arguments of Fact definition)

⚠ **This environment can be different from the calling environment!**

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*}

131

131

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}

132

132

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 and the program
  - To use the semantics, we first translate Fact into kernel language, and then we execute on the abstract machine
- This completes the proof

133

133

Semantics 5: Semantic rules of procedures

134

Procedures are the building blocks of abstraction

- Procedure definition and call are very important, since they are the foundation of all data abstraction
  - Higher-order programming
  - Layered program organization
  - Encapsulation
  - Object-oriented programming (objects and classes)
  - Abstract data types
  - Component-oriented programming (packages, modules)
  - Multi-agent programming (agents sending messages)

- This is why we study them separately

135

135

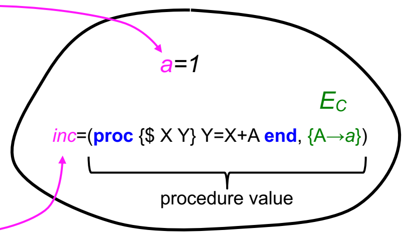
We recall how procedures are stored in memory

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*}



136

136

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

137

137

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

138

138

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
```

139

139

Semantic rule for procedure definition

- Semantic instruction: (<x>=**proc** {$ <x>$_1$ … <x>$_n$} <s> **end**, *E*)
  - Formal arguments: <x>$_1$, …, <x>$_n$
  - Free identifiers in <s>: <z>$_1$, …, <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*=(**proc** {$ <x>$_1$ … <x>$_n$} <s> **end**, E_C)

140

140

Semantic rule for procedure call (1)

- Semantic instruction:

  $(\{\langle x\rangle\ \langle y\rangle_1 \ldots \langle y\rangle_n\}, E)$

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

141

141

Semantic rule for procedure call (2)

Most important slide of the course

- Semantic instruction on stack:

  $(\{\langle x\rangle\ \langle y\rangle_1 \ldots \langle y\rangle_n\}, E)$

  with procedure definition in memory:

  $E(\langle x\rangle) =$ (**proc** {\$ $\langle z\rangle_1 \ldots \langle z\rangle_n$} $\langle s\rangle$ **end**, E_C)

- Put the following instruction on the stack:

  $(\langle s\rangle, E_C + \{\langle z\rangle_1 \to E(\langle y\rangle_1), \ldots, \langle z\rangle_n \to E(\langle y\rangle_n)\})$

142

142

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

143

143

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}

144

144

Restriction

- For a procedure declaration

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

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

145

145

Semantics summary

146

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 gave semantic rules for the kernel language instructions, to show how they execute in the abstract machine
- We used the semantics to prove program correctness, by using it as bridge between specification and program

Specification $\xleftrightarrow{\text{Semantics}}$ Program

Mathematical formula | *Programming language text*

147

147

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 a new construction!
- 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)

148

148

Why semantics is important

- Semantics is an intrinsic part of programming
  - As a programmer, you are extending the system's semantics: you are writing specifications, designing and implementing abstractions (which we will see later on), and reasoning about your work
- The design of any complicated system with parts that interact in interesting ways (like programming languages and programs) should be done hand in hand with designing a semantics
  - Designing a *simple* semantics is the only way to avoid unpleasant surprises and to guarantee a simple mental model
  - Users don't need to understand the semantics to take advantage of it: its mere existence is enough
    - Only the system's designers need to understand the semantics
- « Semantics is the ultimate programming language »
  - Invariants are the ultimate loop construct (invariant programming)
  - Data abstractions as new kernel language instructions

149

149

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

150

150

“Semantics is the ultimate programming language”

151