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

Lecture 2
Symbolic programming

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview of lectures 2 & 3

- Symbolic programming (lecture 2)
  - Lists
  - Pattern matching
  - Trees
  - Tuples and records
- Formal semantics (lecture 3)
  - Kernel language
  - Abstract machine
  - Proving correctness of programs
  - Semantic rules for kernel instructions
  - Semantics of procedures

2

2

Lists

3

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

4

4

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

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

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.

7

7

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.

8

8

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

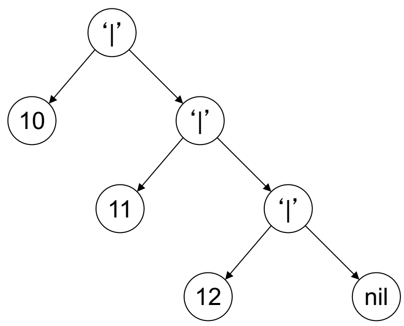
- Oz allows another textual representation
  - Bracket notation: [10 11 12]
  - In memory, [10 11 12] is identical to 10 | 11 | 12 | nil
  - Different textual representations of the same thing are called syntactic sugar

9

9

Graphical representation of a list

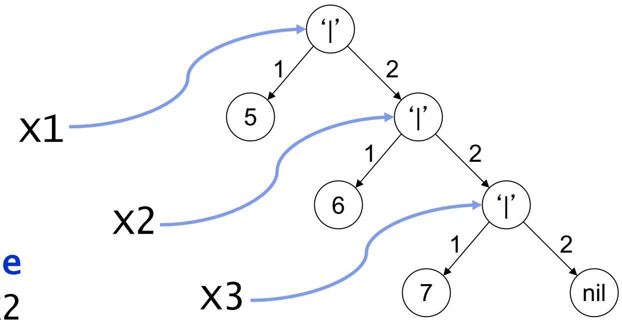
- Graphical representations are very useful for reasoning
  - Humans have very powerful visual reasoning abilities
- We start from the leftmost pair, namely 10 | <List <Int>>
  - We draw three nodes with arrows between them
  - We then replace the node <List <Int>> as before
- This is an example of a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



10

10

Building a list incrementally




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

11

11

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**

12

12

Head and tail functions

- We can define functions

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

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

13

13

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]!

14

14

Functions on lists

15

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)

16

16

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}}

17

17

Sum of list elements (naïve method)

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

18

18

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?

19

19

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}

20

20

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?

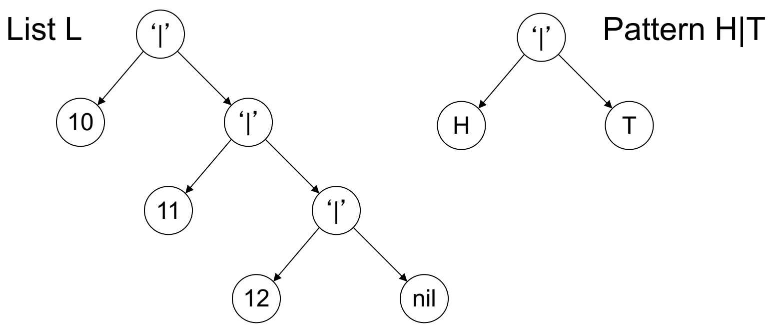
21

21

Pattern matching

22

Pattern matching

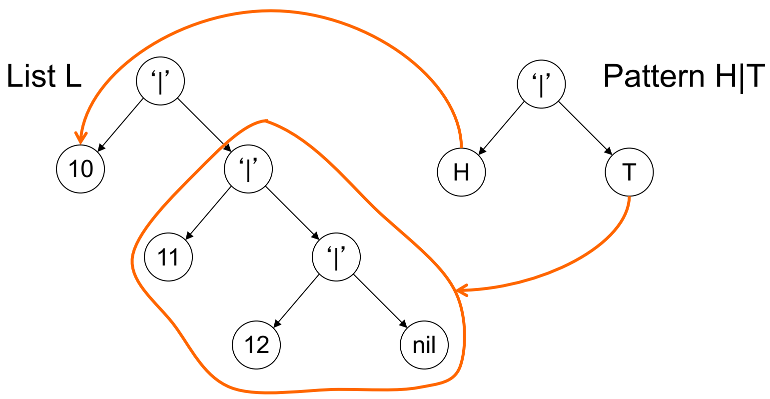


- **case** L **of** H|T **then** ... **else** ... **end**

23

23

Pattern matching



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

24

24

Sum with pattern matching

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

25

25

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

26

26

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

27

27

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

28

28

Kernel language introduction

29

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

30

30

Length of a list

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

31

31

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
```

32

32

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}

33

33

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

34

34

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>

35

35

Trees

36

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

37

37

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>)

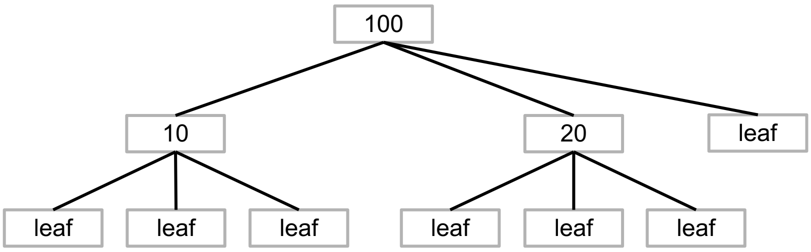
subtrees

38

38

Example tree

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



39

39

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>)

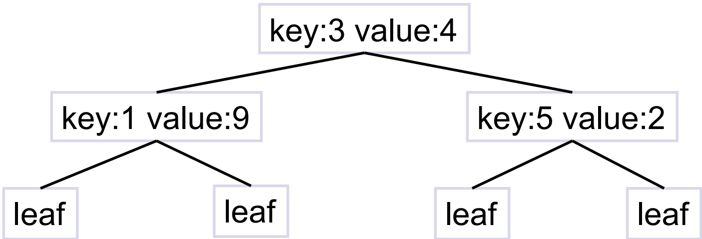
Notice the similarity with lists!

40

40

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

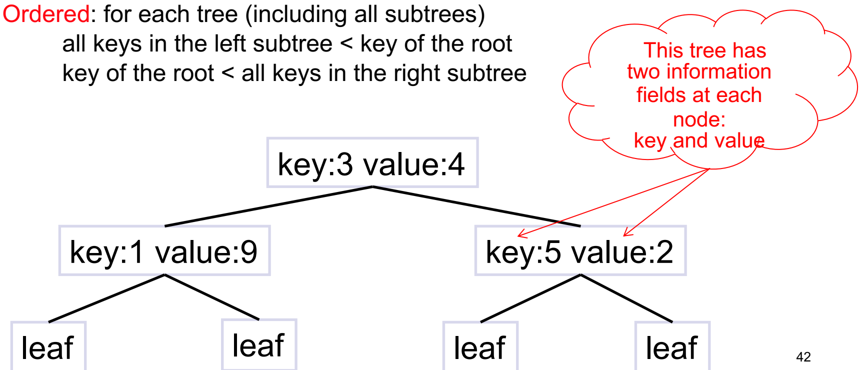


41

41

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

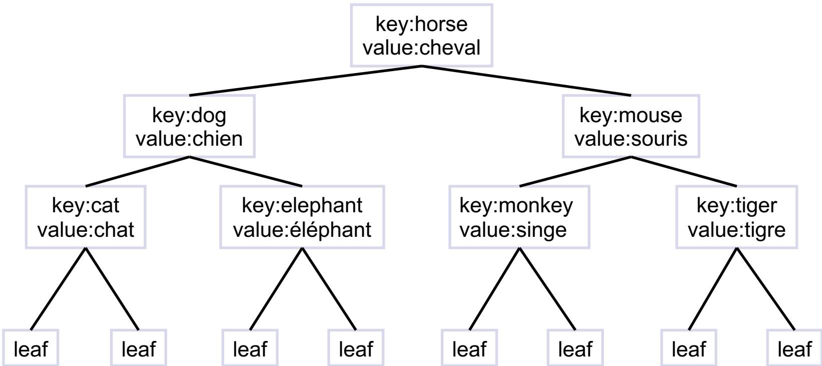


42

42

Ordered binary tree (3)

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

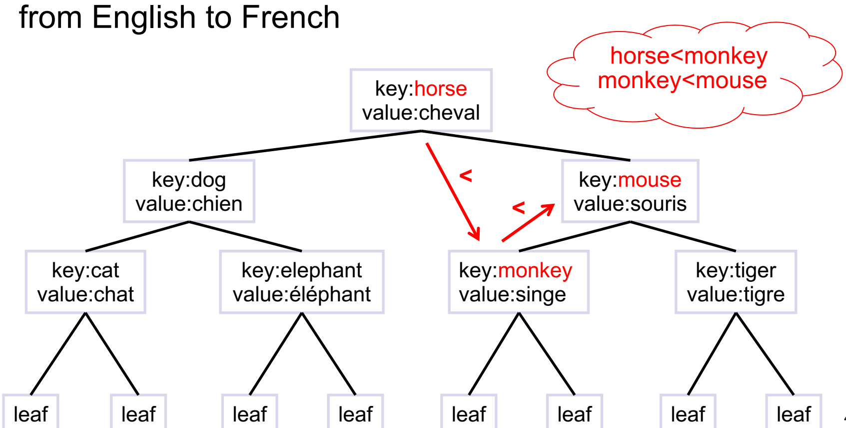


43

43

Ordered binary tree (4)

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



44

44

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

45

45

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
```

46

46

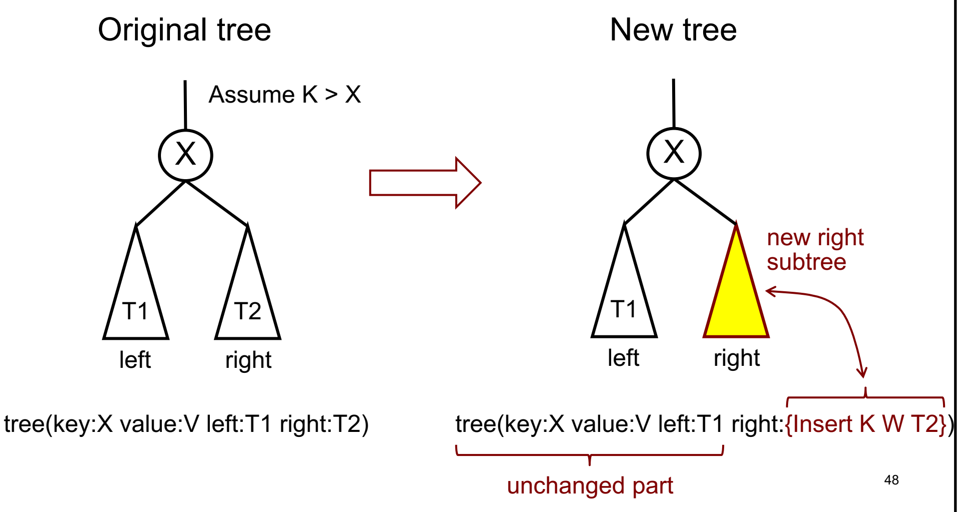
Efficiency of Lookup

- 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

47

47

Inserting a new key/value pair



48

48

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
```

49

49

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
```

50

50

Deleting information

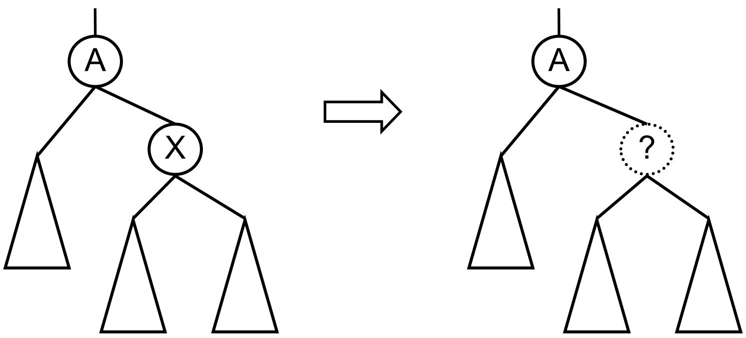
- There are four possibilities:
- (K,_) is not in the tree
- (K,_) is removed immediately
- (K,_) is removed from the left subtree
- (K,_) is removed from the right subtree
- Right? WRONG!

```
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
```

51

51

Deleting an element from an ordered binary tree

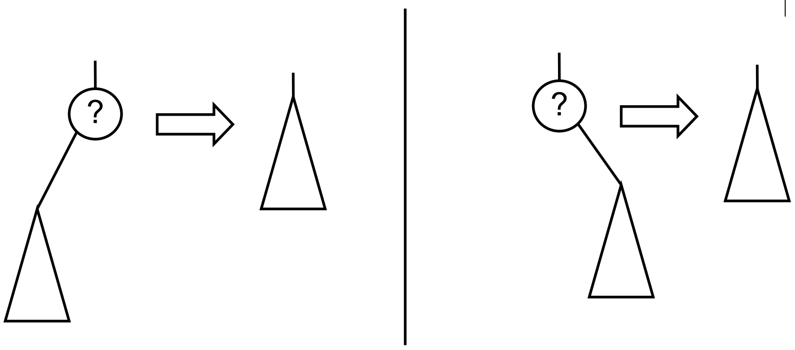


The problem is to repair the tree after X disappears

52

52

Deleting the root when one subtree is empty

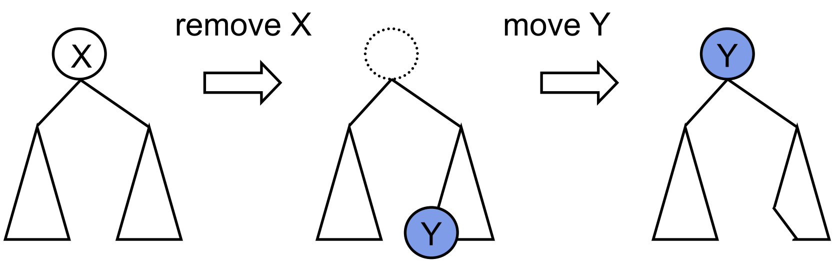


It's easy when one of the subtrees is empty:
just replace the tree by the other subtree

53

53

Deleting the root when both subtrees are not empty



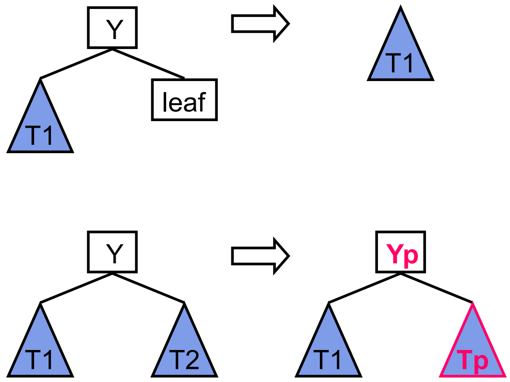
The idea is to fill the "hole" that appears after X is removed. We can put there the smallest element in the right subtree, namely Y.

54

54

Deleting the root

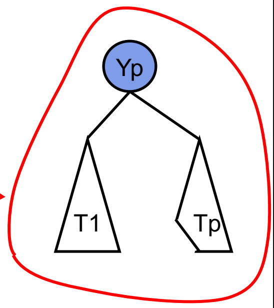
- To remove the root Y, there are two possibilities:
- One subtree is a leaf. Just return the other.
- Neither subtree is a leaf. Remove an element from one of its subtrees.



55

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

56

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

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

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!

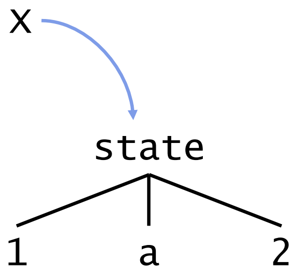
59

59

Tuples and records

60

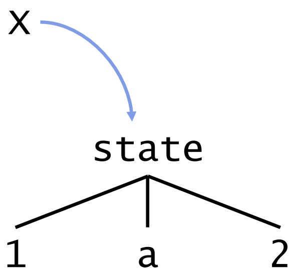
Tuples



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`

Operations on tuples

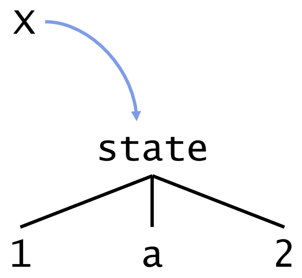


X=state(1 a 2)

- `{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

Accessing fields ("." operation)

X=state(1 a 2)

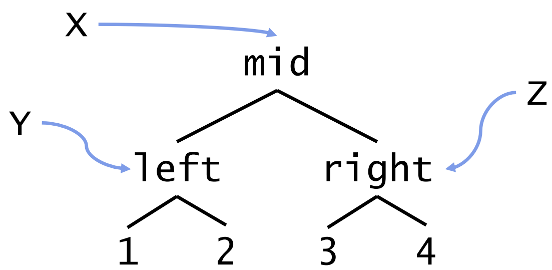


- Fields are numbered from `1` up to `{Width X}`
- `X.N` returns the nth field of tuple X:
  - `X.1` returns `1`
  - `X.3` returns `2`
- In the expression `X.N`, N is called the field name or "feature"

63

63

Building a tree



- A tree can be built with tuples:

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

64

64

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

65

65

Tuples summary

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

66

66

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

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)))

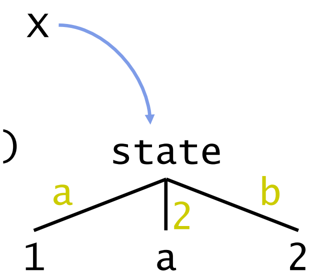
Records

- 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

69

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`

70

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!

71

71

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

72

72

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]
  ```


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

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

75

75

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))

76

76

Conclusions

77

Conclusions

- Lists
  - Pattern matching
  - List functions should be tail-recursive
  - Lists in the kernel language
- Trees
  - Ordered binary trees
  - Lookup and insert are easy; delete is harder
- Tuples and records
  - A list is a tuple and a tuple is a record
  - Records in the kernel language

78