Functions

prof.dr.ir. Wil van der Aalst
www.vdaalst.com
complex network and simple inscriptions/types/functions

simple network and complex inscriptions/types/functions
Folding and Unfolding Petri Nets
Partially folded/unfolded
5 philosophers
Increase number of philosophers: \( n = 20 \)
Unfolded
Background

• CPN language based on the functional programming language Standard ML
• Functional programming: computation proceeds by evaluating expressions
• Imperative programming: computation proceeds by sequential execution of statements and is based on the concept of a state

• Supports recursion and polymorphism
Outline

• Recursion
• List functions and higher order functions
• Pattern matching
• Dealing with complex functions
• Sudoku example
Imperative vs. functional programming

• Sum up a list [1,2,3] of integer values!

**Imperative (OO) programming (Java):**

```java
Iterator iterator = lList.iterator();
int sum = 0;
while (iterator.hasNext()) {
    sum = sum + iterator.next();
}
System.out.println(sum);
```
• Sum up a list [1,2,3] of integer values!

**Functional programming (Standard ML):**

```haskell
fun sum(l:INTlist) = if l = [] then 0
                      else hd(l) + sum(tl(l));
```

**Recursive function: induction base, induction step**
Function in detail

fun sum(l:INTlist) = if l = [] then 0 else hd(l) + sum(tl(l));
fun sum(l:INTlist) = if l = [] then 0 else hd(l) + sum(tl(l));

sum([1,2,3]) = 1 + sum([2,3])
sum([2,3]) = 2 + sum([3])
sum([3]) = 3 + sum([ ]) = 0

Results in 1 + 2 + 3 + 0 = 6
Recursion

• Is the method of defining functions in which a function calls itself
• That way, a kind of iteration is created without using iterative constructs such as loops
• Induction base: trivial value for which we know the answer
• Induction step: describes relation between results of the function of higher parameter values and results of the function of lower parameter values
fun sum(l:INTlist) = if l = [] then 0 
else hd(l) + sum(tl(l));

base; the trivial value 0 if the list is empty

step; sum up the first element and the sum of the remaining list
Example: Factorial

fun fac(x:INT) = if x > 1 then x * fac(x - 1) else 1

• Two cases:
  • fac(x) = x*fac(x-1)
  • fac(1) = 1

• fac(10)=10*fac(9)=10*9*fac(8)=10*9*8*fac(7)=
  ... = 10*9*8*7*6*5*4*3*2*1 = 3,628,800
Recursion (1)

color Product = string;
color Number = int;
color StockItem = record prod:Product * number:Number;
color Stock = list StockItem;
fun totalstock(s:Stock) =
  if s = [] then 0
  else (#number(hd(s))) + totalstock(tl(s));

Also see http://cpntools.org/documentation/concepts/colors/declarations/colorsets/record_colour_sets
Recursion (2)

fun maxstock(s:Stock) =
    if s = [] then 0
    else if (#number(hd(s))) >= maxstock(tl(s))
        then #number(hd(s))
    else maxstock(tl(s));
fun maxstockname(s:Stock) =
  if s = [] then "no product found"
  else if (#number(hd(s))) >= maxstock(tl(s))
    then #prod(hd(s))
  else maxstockname(tl(s));

<table>
<thead>
<tr>
<th>Prod:Product</th>
<th>Number:number</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;apple&quot;</td>
<td>301</td>
</tr>
<tr>
<td>&quot;orange&quot;</td>
<td>504</td>
</tr>
<tr>
<td>&quot;pear&quot;</td>
<td>423</td>
</tr>
<tr>
<td>&quot;banana&quot;</td>
<td>134</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
fun enoughstock(s:Stock, n:Number) =

if s = [] then []

else if (#number(hd(s))) >= n

then hd(s)::enoughstock(tl(s),n)

else enoughstock(tl(s),n);
fun enoughstockn(s:Stock, n:Number) =

if s = [] then 0
else if (#number(hd(s))) >= n
    then 1 + enoughstockn(tl(s), n)
else enoughstockn(tl(s), n);
color Product = string;
color Number = int;
color StockItem = product Product * Number;
color Stock = list StockItem;
var x:StockItem;
var s:Stock;
fun incrs(x:StockItem,s:Stock) = if s=[] then [x]
    else if #1(hd(s)) = #1(x) then (#1(hd(s)),#2(hd(s)) + #2(x))::tl(s)
    else hd(s)::incrs(x,tl(s));
fun decrs(x:StockItem,s:Stock) = incrs((#1(x),~(#2(x))),s);
fun check(s:Stock) = if s=[] then true
    else if #2(hd(s))<0 then false
    else check(tl(s));
val initstock = ["bike",1], ["wheel",2], ["bell",3], ["steering_wheel",3], ["frame",2]:Stock;

Example revisited: Stock keeping system
Store

- Place **stock** is a store; that is, it will always contain a single token
- Only the value of the token matters (not its presence)
- Stores that aggregate elements are always of type **list**
- **Drawback**: complex functions/inscriptions
- **Advantage**: easy to query the individual items as a whole—for example, taking the sum of things...

...
Function "totalstock"

fun totalstock(s:Stock) =

  if s = [] then 0

  else (#2(hd(s))) + totalstock(tl(s));

color Product = string;
color Number = int;
color StockItem = product Product * Number;
color Stock = list StockItem;
var x:StockItem;
var s:Stock;
fun incrs(x:StockItem,s:Stock) = if s=[] then [x]
  else if #1(hd(s)) = #1(x) then (#1(hd(s)),#2(hd(s)) + #2(x)):tl(s)
  else hd(s)::incrs(x,tl(s));
fun decrs(x:StockItem,s:Stock) = incrs((#1(x),~(#2(x))),s);
fun check(s:Stock) = if s=[] then true
  else if #2(hd(s))<0 then false
  else check(tl(s));
val initstock = [("bike",1), ("wheel",2), ("bell",3), ("steering_wheel",3), ("frame",2)]:Stock;
Alternative model

```
color Product = string;
color Number = int;
color StockItem = product Product * Number;
var p:Product;
var x,y:Number;
```

Note the simplicity/elegance of the arc inscriptions.
Outline

• Recursion
• List functions and higher order functions
• Pattern matching
• Dealing with complex functions
• Sudoku example
How to find basic functions?

- cpntools.org, see for example
  http://cpntools.org/documentation/concepts/colors/declarations/colorsets/list_colour_sets and
  http://cpntools.org/documentation/concepts/colors/declarations/colorsets/colour_set_functions

- www.standardml.org, see for example
  http://www.standardml.org/Basis/list.html#LIST:SIG:SPEC for list functions,
  http://www.standardml.org/Basis/integer.html#INTEGER:SIG:SPEC for integer functions,
  http://www.standardml.org/Basis/string.html#STRING:SIG:SPEC for string functions, etc.
**List functions from Standard ML**

- `nil`: empty list (same as `[]`)
- `::`: prepend element `e` as head of list `l`
- `hd l`: head, the first element of the list `l`
- `tl l`: tail, list with exception of first element
- `length l`: length of list `l`
- `rev l`: reverse list `l`
- `map f l`: use function `f` on each element in list `l` and returns a list with all the results
- `app f l`: use function `f` on each element in list `l` and returns `()`
- `foldr f z l`: returns `f(1,f(2,...,f(en,z),...))` where `l = [e1,e2,...,en]`
- `foldl f z l`: returns `f(en,...,f(e2,f(e1,z)))` where `l = [e1,e2,...,en]`
- `List.nth(l,n)`: `n`th element in list `l`, where `0 <= n < length l`
- `List.take(n,l)`: returns first `n` elements of list `l`
- `List.drop(n,l)`: returns what is left after dropping first `n` elements of list `l`
- `List.exists f l`: returns true if `f` is true for some element in list `l`
- `List.null l`: returns true if list `l` is empty
- `List.concat`: concatenates two lists `l1` and `l2`
- `mem x l`: returns true if element `x` is in the list `l`
- `remove l x`: removes duplicates from list `l`
- `rm 1`: removes the first appearance (if any) of element `x` from list `l`
- `rma l x`: removes all appearances (if any) of element `x` from list `l`
- `contains l 1 l2`: returns true if all elements in list `l2` are elements in list `l1`, ignoring the multiplicity
- `containsAll l1 l2`: returns true if all elements in list `l1` are elements in list `l2` and does not ignore multiplicity of elements in `l2`
- `inter l1 l2`: returns the intersection of lists `l1` and `l2`
- `union l1 l2`: returns the union, i.e., the concatenation of lists `l1` and `l2`

### Additional list functions

- `l1 ^ l2`: concatenate the two lists `l1` and `l2`
- `mem 1`: returns true if element `x` is in the list `l`
- `remove 1`: removes duplicates from list `l`
- `rm 1`: removes the first appearance (if any) of element `x` from list `l`
- `rma 1`: removes all appearances (if any) of element `x` from list `l`
- `contains 1 1 1 2`: returns true if all elements in list `l2` are elements in list `l1`, ignoring the multiplicity
- `containsAll 1 1 1 2`: returns true if all elements in list `l1` are elements in list `l2` and does not ignore multiplicity of elements in `l2`
- `inter 1 1`: returns the intersection of lists `l1` and `l2`
- `union 1 1`: returns the union, i.e., the concatenation of lists `l1` and `l2`
hd list  head (i.e., the 1st element) of list

tl list   tail (i.e., list without the 1st element) of list

length list length of list

rev list   reverse list

map f  list uses function f on each element in list and returns a list with all the results

foldr f z list returns f(e1, f(e2,…, f(en, z) …)), where list = [e1, e2, …, en]

List.nth(list,n) (n+1)th element in list, where 0 ≤ n< length list

List.take(list,n) returns 1st n elements of list

List.drop(list,n) returns what is left after dropping the 1st n elements of list

List.exists p list returns true if predicate p is true for some element in list

List.null list returns true if list is empty
Examples
Hints (list functions)

- `hd([])` and `tl([])` are **undefined** and raise an ML exception
- So first define the stop condition of a function
- Avoid calculating lists containing intermediate results
Higher order functions

• Function that has a function as an argument

color IntList = list INT;
val myIntList = [1,2,3,4]:IntList;
fun incr(i:int) = i+1;
map incr myList

Evaluates to [2,3,4,5]
Higher order functions (2)

color IntList = list INT;
val myIntList = [1,2,3,4]:IntList;
foldr op+ 0 myList

Evaluates to $1 + 2 + 3 + 4 = 10$

$op+(a,b) = a+b$
foldr results in $op+(1,op+(2,op+(3,op+(4,0))))$
Outline

• Recursion
• List functions and higher order functions
• Pattern matching
• Dealing with complex functions
• Sudoku example
Pattern matching

- More compact notation
- Does not extend expressive power of the language

```haskell
fun lenlist1(s:Stock) = if s = [] then 0
else 1 + lenlist1(tl(s));
```

```haskell
fun lenlist1([]) = 0 |
lenlist1(s::s) = 1 + lenlist1(s);
```
Pattern matching

fun  \text{lenlist1}([\text{\textbf{\textit{}}}}) = 0 | \\
\quad \text{lenlist1}({\text{\textbf{\textit{si::s}}}}) = 1 + \text{lenlist1}(s); \\
\text{Replaces } \text{\textbf{if}} \quad \text{\textbf{\textit{then}}} \quad \text{\textbf{\textit{else}}} \\
\text{Replaces } \text{\textbf{\textbf{hd}}} \quad \text{\textbf{\textbf{\textit{and}}} \quad \text{\textbf{\textbf{tl}}}}; \\
{\text{si}} \text{ \textbf{evaluates to } head \ of \ list; } \quad {s} \text{ \textbf{evaluates to } tail \ of \ list} \\
\text{Because } \text{\textbf{\textbf{hd}}([\text{\textbf{\textit{}}}])} \text{ \textbf{and } } \text{\textbf{\textbf{tl}}([\text{\textbf{\textit{}}}])} \text{ \textbf{are undefined,}} \\
\text{\textbf{\textbf{si::s}}} \text{ \textbf{is a list of at least length 1}}
fun lenlist1([]) = 0 |

lenlist1(si::s) = 1+lenlist1(s);

- lenlist1 accepts only lists (follows from pattern matching)
- Type of elements in the list not important
- Function is independent of the exact type
- can be applied to any list
fun maxstock(s:Stock) =
    if s=[] then 0
    else if (#number(hd(s))) >= maxstock(tl(s)) then #number(hd(s))
    else maxstock(tl(s));
Pattern matching (3)

fun incr(x:StockItem, []:Stock) = [x] |

incr (x,(si::s)) = if (#prod(si)) = (#prod(x))

then {prod=(#prod(si)),

number = ((#number(si)) + (#number(x))))

:: incr(x,s)

else si::incr(x,s);

<table>
<thead>
<tr>
<th>Prod:Product</th>
<th>Number:number</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;apple&quot;</td>
<td>301</td>
</tr>
<tr>
<td>&quot;orange&quot;</td>
<td>504</td>
</tr>
<tr>
<td>&quot;pear&quot;</td>
<td>423</td>
</tr>
<tr>
<td>&quot;banana&quot;</td>
<td>134</td>
</tr>
<tr>
<td>&quot;XX&quot;</td>
<td>20</td>
</tr>
</tbody>
</table>

x={prod="XX", number=20}
fun reverse([]) = [ ] |
reverse(x::y) = reverse(y)^^[x];

fun elt([], a) = false |
elt((x::xs), a) = a=x orelse elt(xs, a);

fun del(a,[]) = [] |
del(a,(x::xs)) = if a=x then xs else x::(del(a,xs));

fun intersect([], ys) = [] |
intersect(xs, []) = [] |
intersect((x::xs), ys) = if elt(ys,x)
then x::(intersect(xs,(del(x,ys)))))
else intersect(xs, ys);
Outline

- Recursion
- List functions and higher order functions
- Pattern matching
- Dealing with complex functions
- Sudoku example
Dealing with complex functions

**Approach:** decompose complex problem into smaller problems

**Example:** Remove a list $l_1$ of values from a list $l_2$

- Write function $\text{del}$ for removing an element from a list
- Write function $\text{delList}$ that iterates through $l_1$ and calls for every element function $\text{delList}$
fun del(a, []) = [] |
   del(a, (x::xs)) = if a = x then xs
                     else x::(del(a, xs));

fun delList(l1, []) = [] |
                      delList([], l2) = l2 |
                      delList((x::xs), l2) = delList(xs, del(x,l2));
Example: Signing documents

• Documents need to be signed by persons.
• Four persons: Tim, Sue, Clare and John.
• Each document requires three signatures.
• No two signatures of the same person.
• Work in progress is limited to five documents.
color Doc = string;
color Person = string;
color Signatures = list Person;
color SignedDoc = product Doc * Signatures;
color BlackToken = unit;
var d:Doc;
var p:Person;
var s:Signatures;
fun notin(p:Person,[]:Signatures) = true |
   notin(p,h::s) = if p=h then false else notin(p,s);
fun count([]) = 0 |
   count(h::s) = 1+count(s);
Signing documents: Network structure

(person: ("Tim"::d, "Sue"::d, "Clare"::d, "John"::d))

[\text{count}(s)<3 \text{ andalso } \neg \text{in}(p,s)]

\begin{align*}
\text{unsigned} \_ \text{doc} & \xrightarrow{d} \text{Doc} \\
\text{accept} & \xrightarrow{(d,[])} \text{SignedDoc} \\
\text{pile} & \xrightarrow{(d,s)} \text{release} \\
\text{free} & \xrightarrow{5()`} \text{BlackToken} \\
\text{SignedDoc} & \xrightarrow{(d,s)} \text{signed} \_ \text{doc}
\end{align*}
Exercise

- Replace place free by a place always holding one token.

```plaintext
color Doc = string;
color Person = string;
color Signatures = list Person;
color SignedDoc = product Doc * Signatures;
color BlackToken = unit;
var d:Doc;
var p:Person;
var s:Signatures;
fun notin(p:Person,[]:Signatures) = true |
    notin(p,h::s) = if p=h then false else notin(p,s);
fun count([]) = 0 |
    count(h::s) = 1+count(s);
```
Outline

• Recursion
• List functions and higher order functions
• Pattern matching
• Dealing with complex functions
• Sudoku example
Example: Sudoku

colset Index = int with 0..8;
colset Cell = int with ~1..9;
colset Pos = product Index * Index;
colset Val = product Pos * Cel;
colset Sudoku = list Val;

for 9 rows and columns

~1 and 0 have a technical reason, normal values are 1..9

Write an ML function to solve a Sudoku, assuming that in each step there is a "deterministic candidate" (i.e., no backtracking is needed).
Alternative input format : Sudoku_list

val v4 = [
  [6,0,0, 0,8,0, 0,0,9],
  [0,7,0, 4,0,6, 0,8,0],
  [0,0,0, 5,0,1, 0,0,0],
  [0,1,7, 2,0,9, 8,5,0],
  [2,0,0, 0,0,0, 0,0,1],
  [0,8,4, 1,0,3, 6,7,0],
  [0,0,0, 3,0,8, 0,0,0],
  [0,4,0, 9,0,5, 0,1,0],
  [8,0,0, 0,7,0, 0,0,5]
];

0 values are empty

0 values are not inserted

to map list of lists representation to list of ((i,j),c) values
colset Sudoku_string = string;

fun convert_to_string_h(s:Sudoku,i) = 
if i>= 81 then "------------------------\n" else 
(Int.toString(fmap(s,(i div 9, i mod 9)))) ^ 
(if i mod 9 = 8 then "\n" else " ")^ 
convert_to_string_h(s,i+1));

fun convert_to_string(s:Sudoku) = "\n------------------------
\n"^convert_to_string_h(s,0):Sudoku_string;
Useful functions

fun dom([ ] ) = [ ] | dom((x,y)::l) = x::dom(l);
fun elt([ ], a) = false |
    elt((x::xs), a) = a=x orelse elt(xs, a);
fun fmap([ ],z) = 0 |
    fmap((x,y)::l,z) = if x=z then y else fmap(l,z);
fun sdiff([ ],z) = [ ] |
    sdiff(x::y,z) = if elt(z,x) then sdiff(y,z) 
        else x::sdiff(y,z);

infix sdiff;

difference of two sets
Basic functions

fun row([],k) = [] |
    row(((i,j),c)::s,k) = if i=k then c::row(s,k)
    else row(s,k) : Cels;

fun column([],k) = [] |
    column(((i,j),c)::s,k) = if j=k then c::column(s,k)
    else column(s,k) : Cels;

fun de(i,j) = (i div 3) = (j div 3);

fun block([],i,j) = [] |
    block(((i1,j1),c)::s,i,j) = if de(i,i1) and
    de(j,j1) then c::block(s,i,j) else block(s,i,j) : Cels;

fun free(s,i,j) = (([1,2,3,4,5,6,7,8,9] sdiff row(s,i)) sdiff column(s,j)) sdiff block(s,i,j) : Cells;
all possible values of type Pos, i.e., list of all cells

\[ \text{colset } \text{Vals} = \text{product } \text{Pos} \]

\[ \text{colset } \text{Sudoku}_{\text{opt}} = \text{list } \text{Vals} \]

\[ \text{fun } \text{allpos}() = \text{Pos.all}() \]

\[ \text{fun } \text{undef}(s) = \text{allpos()} \text{ sdiff } \text{do} \]

\[ \text{fun analyze1}(s,[ ]) = [ ] | \]

\[ \text{analyze1}(s,(i,j)::l) = ((i,j),\text{free}(s,i,j))::\text{analyze1}(s,l); \]

\[ \text{fun analyze}(s) = \text{analyze1}(s,\text{undef}(s)): \text{Sudoku}_{\text{opt}}; \]
fun next_move([]) = [] | 
next_move(((i,j),[])::s) = ((i,j),~1)::next_move(s) | 
next_move(((i,j),[c])::s) = ((i,j),c)::next_move(s) | 
next_move(((i,j),c::cs)::s) = next_move(s)::Sudoku_moves;

fun solve(s) = if next_move(analyze(s)) = [] then s else solve(next_move(analyze(s))^^s):Sudoku;

repeatedly call solve until no entries can be added (done or non-deterministic choice needed)
Sudoku s can be solved by function solve(s), but let us first draw a Petri net

*** replace v1 by v2, v3, v4 or your own sudoku ***

1: \[\begin{bmatrix} 9 & 0 & 3 & 5 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 & 7 & 0 \end{bmatrix} \]
*** replace v1 by v2, v3, v4 or your own sudoku ***

```
1
---------------------
9 0 3 5 0 0 1 0 0
0 0 6 7 0 0 0 0 0
0 0 0 0 4 0 0 0
0 0 5 0 0 8 4 0 0
7 6 0 0 0 0 0 2 1
0 0 4 0 0 0 0 3 0 0
0 0 0 0 5 0 0 0
0 0 0 0 9 2 0 0
0 0 0 1 0 0 2 7 0 8
---------------------
```

```
1
---------------------
9 8 3 5 2 6 1 4 7
4 5 6 7 9 1 8 3 2
1 7 2 8 3 4 9 5 6
3 9 5 2 1 8 4 7 6
7 6 8 9 4 3 5 2 1
2 1 4 8 5 7 3 8 9
8 2 9 3 7 5 3 1 4
6 4 7 1 8 9 2 5 3
5 3 1 4 6 2 7 9 8
---------------------
```

```
1
---------------------
9 0 3 5 0 0 1 0 0
0 0 6 7 0 0 0 0 0
0 0 0 0 4 0 0 0
0 0 5 0 0 8 4 0 0
7 6 0 0 0 0 0 2 1
0 0 4 0 0 0 0 3 0 0
0 0 0 0 5 0 0 0
0 0 0 0 9 2 0 0
0 0 0 1 0 0 2 7 0 8
---------------------
```
Another example

*** replace v1 by v2, v3, v4 or your own sudoku ***
Needs to be deterministic

*** replace v1 by v2, v3, v4 or your own sudoku ***

---

---
val it = 
[((0,0),9),((0,1),8),((0,2),3),((0,3),5),((0,4),2),((0,5),6),((0,6),1),
((0,7),4),((0,8),7),((1,0),4),((1,1),5),((1,2),6),((1,3),7),((1,4),9),
((1,5),1),((1,6),8),((1,7),3),((1,8),2),((2,0),1),((2,1),7),((2,2),2),
((2,3),8),((2,4),3),((2,5),4),((2,6),9),((2,7),6),((2,8),5),((3,0),3),
((3,1),9),((3,2),5),((3,3),2),((3,4),1),((3,5),8),((3,6),4),((3,7),7),
((3,8),6),((4,0),7),((4,1),6),((4,2),8),((4,3),9),((4,4),4),((4,5),3),
((4,6),5),((4,7),2),((4,8),1),((5,0),2),((5,1),1),((5,2),4),((5,3),6),
((5,4),5),((5,5),7),((5,6),3),((5,7),8),((5,8),9),((6,0),8),((6,1),2),
((6,2),9),((6,3),3),((6,4),7),((6,5),5),((6,6),6),((6,7),1),((6,8),4),
((7,0),6),((7,1),4),((7,2),7),((7,3),1),((7,4),8),((7,5),9),((7,6),2),
((7,7),5),((7,8),3),((8,0),5),((8,1),3),((8,2),1),((8,3),4),((8,4),6),
((8,5),2),((8,6),7),((8,7),9),((8,8),8)) : ((int * int) * Cell) list

sort_sudoku(solve(convert_from_list(v1)));
fun sord(((x1,y1),z1),((x2,y2),z2)) =
  (x1 < x2) orelse (x1=x2 andalso y1 < y2);
fun sort_sudoku(s) = sort sord s;

sort function is built in
“sort lt_fun l” sorts list l using the
function lt_fun to determine when one
element in the list is less than another.

See
http://cpntools.org/documentation/concepts/colors/declarations/colorsets/
list_colour_sets
See sudoku.cpn
See sudoku.cpn

val it = 
[[9,0,3,5,0,0,1,0,0],[0,0,6,7,0,0,0,0,0],[0,0,0,8,0,4,0,0,0],
 [0,0,5,0,0,8,4,0,0],[7,6,0,0,0,0,0,2,1],[0,0,4,6,0,0,3,0,0],
 [0,0,0,3,0,5,0,0,0],[0,0,0,0,0,9,2,0,0],[0,0,1,0,0,2,7,0,8]] : Sudoku_list

val it = 
[((0,0),9),((0,1),8),((0,2),3),((0,3),5),((0,4),2),((0,5),6),((0,6),1),
 ((0,7),4),((0,8),7),((1,0),4),((1,1),5),((1,2),6),((1,3),7),((1,4),9),
 ((1,5),1),((1,6),8),((1,7),3),((1,8),2),((2,0),1),((2,1),7),((2,2),2),
 ((2,3),8),((2,4),3),((2,5),4),((2,6),9),((2,7),6),((2,8),5),((3,0),3),
 ((3,1),9),((3,2),5),((3,3),2),((3,4),1),((3,5),8),((3,6),4),((3,7),7),
 ((3,8),6),((4,0),7),((4,1),6),((4,2),8),((4,3),9),((4,4),4),((4,5),3),
 ((4,6),5),((4,7),2),((4,8),1),((5,0),2),((5,1),1),((5,2),4),((5,3),6),
 ((5,4),5),((5,5),7),((5,6),3),((5,7),8),((5,8),9),((6,0),8),((6,1),2),
 ((6,2),9),((6,3),3),((6,4),7),((6,5),5),((6,6),6),((6,7),1),((6,8),4),
 ((7,0),6),((7,1),4),((7,2),7),((7,3),1),((7,4),8),((7,5),9),((7,6),2),
 ((7,7),5),((7,8),3),((8,0),5),((8,1),3),((8,2),1),((8,3),4),((8,4),6),
 ((8,5),2),((8,6),7),((8,7),9),((8,8),8)] : (int * int) * Cell list

sort_sudoku(solve(convert_from_list(v1)));

solve(convert_from_list(v1));

note the two 1's in middle block
Solved correctly
More information

• About Standard ML:
  • http://www.standardml.org/Basis/ (for functions)

• About CPN:
<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Type</th>
<th>Topic</th>
<th>Reading Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17-5-2013</td>
<td>Exam</td>
<td>Pre-exam focusing on classical Petri nets (1 point)</td>
<td>Study Chapters 1-4 and all exercises in Sections 1-2.</td>
</tr>
<tr>
<td></td>
<td>(14.00-15.30)</td>
<td>Inst.</td>
<td>Explanation “CPN assignment” (3 points)</td>
<td>Start making exercises in Section 1-2.</td>
</tr>
<tr>
<td></td>
<td>24-5-2013</td>
<td>Inst.</td>
<td>Modeling in terms of CPN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30-5-2013</td>
<td></td>
<td>No lecture</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31-5-2013</td>
<td>Inst.</td>
<td>CPN modeling continued</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>3-6-2013</td>
<td>Lect.</td>
<td>Simulation (9)</td>
<td>Read Chapter 8.</td>
</tr>
</tbody>
</table>
After this lesson you should be able to

• Understand the concept recursion;
• Know how to define functions in the CPN language;
• Know the basic ML list functions or at least where to look them up;
• Apply pattern matching when defining functions;
• Know how to define a complex function (by splitting this function into simpler functions).