SML lectures

- 14.10. - The Algol Family and SML (Mitchell's chap. 5)
- **21.10. - More on ML & Recursive/Higher-order Functions (chap. 5)**
- 4.11. - Types, Type Inference and Polymorphism (chap. 6)
- 11.11. - Control in sequential languages, Exceptions and Continuations (chap. 8)

IN3040 - 2022

Functions and Pattern Matching

- Function declaration
  - Functions are as other values:
    - (5*6) ;
    > val it = 30 : int
    - fn x => x * 2 ; "anonymous function", in lambda notation λx . (x * 2)
    > val it = fn : int -> int
    - val dbl = fn x => x * 2 ; > val dbl = fn : int -> int
  - But we have a special syntax for defining functions:
    - fun dbl x = x * 2 ; > val dbl = fn : int -> int
- Function declaration, general form
  - fun f (<pattern>) = <expr>
    - fun f (x,y) = x+y; Actual par. must match pattern (x,y)
  - fn <pattern> => <expr>
    - fn (x,y) => x+y; Anonymous function
- Multiple-clause definition
  - fun <name> <pat_1> = <exp_1> | …

    | <name> <pat_n> = <exp_n>

  - fun length (nil) = 0

    | length (x::s) = 1 + length(s);

  > val length = fn ´a list -> int

  - length ["J", "o", "n"] > val it = 3 : int

IN3040 - 2022

Some functions on lists

- Insert an element in an ordered list

```
fun insert (e, nil)      = [e]
 |   insert (e, x::xs) = if e>x then x :: insert(e,xs)
                                 else e::(x::xs);
- insert (3,[1,2,5]) ;
> val it = [1,2,3,5] : int list
```

- Append lists

```
fun append(nil, ys) = ys
|    append(x::xs, ys) = x :: append(xs, ys);
- append ([3,4],[1,2]) ;
>val it = [3,4,1,2] : int list
```

Data-type declarations

- Enumeration types
  - datatype color = Red | Yellow | Blue;
    - elements are: Red, Yellow, Blue <- Constructors!
- Tagged union types
  - datatype value = I of int | R of real | S of string;
    - elements are: I(9) , R(8.3) , S("hello") ...
  - datatype keyval = StrVal of string * string | IntVal of string * int ;
    - elements are: StrVal("foo","bar") , IntVal("foo",55) ...
  - datatype mylist = Nil | Cons of value * mylist
    - elements are: Nil , Cons (I(8) ,Nil) , Cons (R(1.0), Cons (I (8), Nil))
- General form

  datatype <name> = <clause> | … | <clause>

  <clause> ::= <constructor> | <constructor> of <type>

Type abbreviations

- We use *datatype* to define new types
- The keyword *type* can be used to define a type *abbreviation:*

```
- type int_pair = int * int ;
```

- The type inference will not report types as the defined abbrev.:

```
- val a = (3,5);
> val a = (3,5) : int * int
```

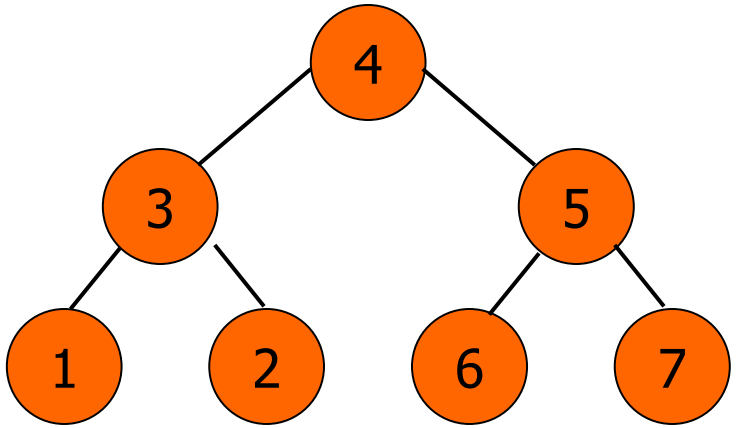
- We can force the use of type abbreviation:

```
- val a : int_pair = (3,5);
> val a = (3,5) : int_pair
```

Datatype and pattern matching

- Recursively defined data structure
  - datatype tree = Leaf of int | Node of int*tree*tree;

```
Node(4, Node(3,Leaf(1), Leaf(2)),
        Node(5,Leaf(6), Leaf(7))
     )
```



- Recursive function (sum)
  - fun sum (Leaf n) = n
    | sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2);

Case expression

- Datatype
  - datatype exp = Num of int | Var of string | Plus of exp*exp;
- Case expression

```
case e of Num(i)       =>  …  |
          Var(v)       => ….  |
          Plus(e1,e2) => …
- fun eval(e) = case e of  Num(i) => i
                         | Var(v) => lookUp(v)
                         | Plus(e1,e2) => eval(e1) + eval(e2)
```

- Case matching is done in order
- Use _ to catch all missing

```
- fun bintoString(i) = case i of 0 => "zero"
                               |    1 => "one"
                               |    _ => "illegal value";
> val bintoString = fn : int -> string
```

- Can also use _ in declarations if we don't care about the value being matched

```
- fun hd(x::xs) = x ;
- fun hd(x::_) = x ;
```

IN3040 - 2022

insert: Three ”different” declarations

1.
```
fun insert (e, ls) =
    case ls of nil  => [e]
             | x::xs => if e>x then x::insert(e, xs) else
e::ls ;
```

2.
```
fun insert (e, nil)    = [e]
  | insert (e, x::xs) = if e>x then x::insert(e, xs)
                                else e::(x::xs) ;
```

3.
```
fun insert (e: int, ls: int list) : int list =
  case ls of nil   => [e]
           | x::xs => if e>x then x::insert(e, xs) else
e::ls ;
```



UNIVERSITETET
I OSLO

More on ML & Types

Volker Stolz
stolz@ifi.uio.no

Department of Informatics – University of Oslo

Initially by Gerardo Schneider.
Based on John C. Mitchell's slides (Stanford U.)

IN3040 - 2022

Outline

- More recursive examples
- Higher-order functions
- Something about equality
- ~~Something on the ML module system~~
- Types in programming
- Type safety

IN3040 - 2022

More on list functions

- Writing a recursive function is not difficult, but what about efficiency?

- Example: Reverse a list
  (remember [1,2] @ [3,4] = [1,2,3,4])

```
fun  rev [] = []
   |  rev (x::xs)      = (rev xs) @ [x] ;
```

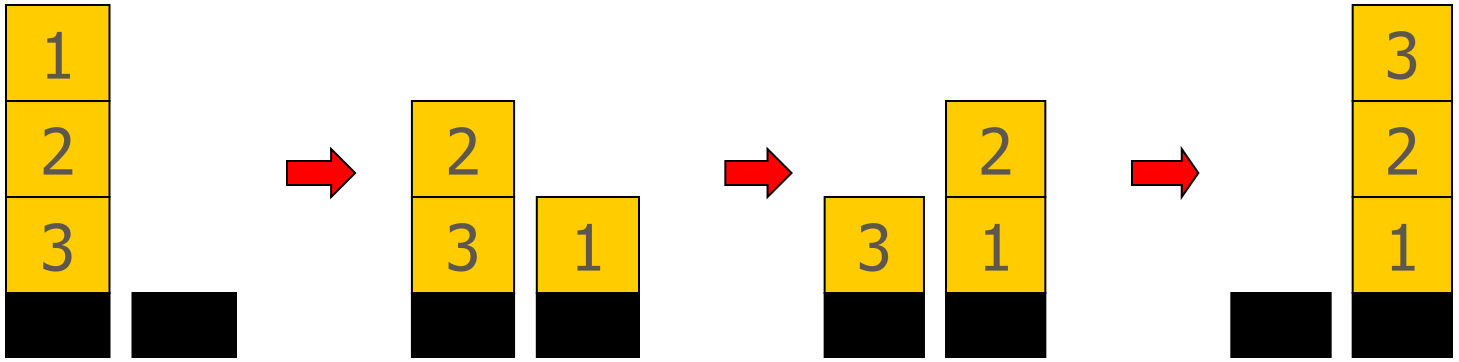
- Questions
  - How efficient is reverse?
  - Can you do this with only one pass through list?

More efficient reverse function

```
fun revAppend ([],ys) = ys
  | revAppend (x::xs,ys) = revAppend(xs,(x::ys)) ;

fun rev xs = revAppend(xs,[]);
```

Tail recursive function!



Two factorial functions

- Standard recursion

```
- fun fact n =
    if n = 0 then 1 else n * fact(n-1) ;
```

- Tail recursive (iteritative)

```
- fun facti(n,p) =
    if n = 0 then p else facti(n-1,n*p) ;
- fun fact n = facti(n,1) ;
```

- More examples in Pucella sec. 2.7

Outline

- More examples on recursion
- Higher-order functions
- Something about equality
- ~~Something on the ML module system~~
- Types in programming
- Type safety

IN3040 - 2022

Monomorphism vs. Polymorphism

- *Monomorphic* means ”having only one form”, as opposed to *Polymorphic*
- A type system is monomorphic if each constant, variable, etc. has unique type
- Variables, expressions, functions, etc. are polymorphic if they ”allow” more than one type

Example. In ML, the *identity* function fn x => x is polymorphic: it has infinitely many types!

```
- fn x => x
> val it = fn : 'a -> 'a
```

Warning! The term ”polymorphism” is used with different specific technical meanings (more on this in ML-lecture 3)

IN3040 - 2022

Higher-order functions (functionals)

- In ML functions are computational values ("first-class objects")
  - can be constructed during execution
  - stored in data structures
  - passed as arguments to other functions
  - returned as values

A *functional* is a function that operates on other functions

- Programs are more concise and clear when using functionals
- Functionals on lists have been very popular in Lisp
- The use of functionals is a powerful tool for *modularisation* which is what gives FPLs one of its conceptual advantages (Hughes 1984)

Higher-order functions (functionals)

- Map: apply a function to every element in a list

```
- fun map (f, nil) = nil
  |    map (f, x::xs) = f(x) :: map (f,xs);
> val map = fn : ('a -> 'b) * 'a list -> 'b list

- fun incr x = x+1 ;
> val incr = fn : int -> int

- map (incr, [1,2,3]);          [2,3,4]

- map (fn x => x*x, [1,2,3]);   [1,4,9]
```

Higher-order functions (functionals)

- Map: apply a function to every element in a list

```
- fun map (f, nil) = nil
  |    map (f, x::xs) = f(x) :: map (f,xs);
> val map = fn : ('a -> 'b) * 'a list -> 'b list
```

```
- fun bintoString(i) =
  case i of  0  => “zero”
           |    1  => “one”
           |    _  => “illegal value”;

> val bintoString = fn : int -> string
```

```
- map (bintoString , [1,0,2,0]);
> val it = ["one","zero","illegal value","zero"] : string list
```

Higher-order functions (functionals)

- filter: apply a predicate to every element of list

```
- fun filter (p, nil) = nil
    | filter (p, (x::xs))  = if p(x) then x :: (filter (p,xs))
                             else filter (p,xs) ;

- val odd = fn : int -> bool

- val mylist = [1,2,3,4,5,6,7,8];
- filter (odd, mylist);                    > val it = [1,3,5,7] : int list

- map (fn x => x*x, (filter(odd,mylist)));

                                  > val it = [1,9,25,49] : int list

- val pairs = [(1,2),(4,3),(8,9),(0,9),(0,0),(5,1)] ;

-  filter ((op <) , pairs);

            > val it = [(1,2),(8,9),(0,9)] : (int * int) list
```

Curried functions

- A function can have only one argument
  - tuples are used for more than one argument
- Multiple arguments may be realized by giving a function as a result
  - *Currying* -> after the logician Haskell B. Curry
- A function over pairs has type

  'a * 'b -> 'c

  while a curried function has type

  'a -> ('b -> 'c)
- A curried function allows *partial application*: applied to its 1st argument (of type 'a), it results in a function of type 'b -> 'c

Curried functions

- Example: function to add two numbers
  - fun pluss (x,y) = x + y ;
  > val pluss = fn : int * int -> int
  - pluss (2,3) ;
  ➢ val it = 5 : int

- Curried version of the same function
  - fun cPluss x y = x + y ;
  > val cPluss = fn : int -> int -> int
  - cPluss 2 3 ;
  > val it = 5 : int
  - val addTwo = cPluss 2 ;
  > val addTwo = fn : int -> int
  - addTwo 5 ;
  > val it = 7 : int

Curried functions

- Curry and uncurry

```
- fun curry f x y = f (x,y) ;
> val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

- fun uncurry f (x,y) = f x y  ;
> val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

Example: the map function

- Recall that map can be defined as

```
fun map (f, nil) = nil
   | map (f, x::xs) = f(x) :: map (f,xs);
   > val map = fn : ('a -> 'b) * 'a list -> 'b list

   - map (fn x => x+1, [1,2,3]);
   > val it = [2,3,4] : int list
```

- By currying it, we can define map as

```
fun map f nil = nil
   | map f (x::xs) = (f x) :: map f xs;
 > val map = fn : ('a -> 'b) -> 'a list -> 'b list

 - map (fn x => x+1)  [1,2,3];
   > val it = [2,3,4] : int list
```

More on the map function

- We can have a function having as argument a function which has another function as an argument
- Thanks to currying, we can combine functionals to work on lists of lists

  Example:

  - map (map (fn x => x+1)) [[1], [1,2], [1,2,3]];

  → [map (fn x => x+1) [1], map (fn x => x+1)[1,2], map (fn x => x+1)[1,2,3]]

  → [[2], [2,3], [2,3,4]]

What does it give as a result?

> val it = [[2],[2,3], [2,3,4]] : int list list

IN3040 - 2022

Outline

- More recursive examples
- Higher-order functions
- Something about equality
- ~~Something on the ML module system~~
- Types in programming
- Type safety

IN3040 - 2022

Equality

- Equality in (S)ML is defined for many types but not all – E.g., it is defined for:
  - Integers
  - Booleans
  - Strings
  - Characters

- What about floating points (reals), compund types (tuples, records, lists), functions, abstract data types, etc?

IN3040 - 2022

Equality

- When are two expressions equal?
  - The so-called *Leibniz's Principle of the Identity of Indiscernables*:

"e1 and e2 are **equal** iff they cannot be distinguished by any operation in the language"

"e1 and e2 are **distinct** iff there is some way to tell them apart"

- What is difficult about Leibniz's Principle?

Problems with Equality

- Equality, as defined by Leibniz's principle, is undecidable

In general, there is no program which determines whether two expressions are equal in Leibniz's sense.

Also:

- Problems with reference cells (aliasing)

```
- val s = ref 1 ; val t = ref 1 ;
- s = t ;                  > false
- !s = !t                  > true
- val s = t ;
- s = t ;                  > true
```

- Polymorphic equality complicates the compiler

Equality Types

- An equality type is a type admiting equality test
- Types admiting equality in (S)ML
  - *int, bool, char, string*
  - *tuples* and *records*, **if all their components admit equality**
  - *datatypes*, **if every constructor's parameter admits equality**
  - *lists* admit equality **if the underlying element type admits equality**
    - Two lists are equal if they have the same length and the same elements in corresponding positions

IN3040 - 2022

Equality Types (cont.)

- Do **not** admit equality in (S)ML
  - *reals*
  - *functions*
  - *tuples, records and datatypes* not mentioned in the previous slide
  - *abstract data types*
- Equality type variable: ”a

```
- fun equals (x,y) = if x = y then true else false ;
> stdIn:7.25 Warning: calling polyEqual
val equals = fn : ''a * ''a -> bool
```

IN3040 - 2022

Equality: Examples

- Equality tests on functions is not computable since
  f = g iff for all x, f(x) = g(x)
- No ”standard” notion of equality for an abstract type
  - What is supposed to be the equality on *trees*? Is it defined structurally? Is it over the list of their elements? By DFS or BFS?
- Ex:

```
fun find x nil = false
  | find x (y :: ys) =   x = y orelse find x ys ;

> = stdIn:30.31 Warning: calling polyEqual
val find = fn : ''a -> ''a list -> bool
```

(don’t worry, only a performance issue)