FUNCTIONAL PROGRAMMING

Contents

Contents of the first part

- Introduction
- Abstraction with functions (processes)
  - Elements of the Program
  - Functions and the Processes They Generate
  - Higher-Order Functions
- Abstraction with data
  - The idea of data-abstraction
  - Hierarchical data structures
  - Multiple Representations for Abstract Data
  - Polymorphic and generic operations

Irodalom: [SICP] Abelson, Sussman & Sussman: *Structure and Interpretation of Computer Programs,* The MIT Press, 1996

History of Functional Programming

Functional Programming Languages

- 1930-40s: *Alonzo Church: λ-calculus*
- LISP (LISt Processing), late 1950s, MIT, US, John McCarthy; typeless
  - for proving some logic expressions (recursive equations), handling symbolic expressions
- ML (Meta Language), Edinborough, GB, mid 1970s; strongly typed, type inference
- Scheme, based on LISP, 1975, MIT, US;
- SML (Standard ML), late 1980s
- Miranda, 1985;GB; strongly typed, non-strict semantics, pure functional, lazy evaluation
- Haskell, similar to Miranda, 1990s, US; static polymorphic typing, type classes, monadic I/O
- Common LISP, 1994, ANSI standard;
- Clean, similar to Miranda & Haskell 1994, Nijmegen, NL; uniqueness type system (for I/O)
- Mercury, based on Prolog, 1995, Melbourne, AU, functional & other extensions to Prolog
- OCaml, based on ML, 1996, INRIA, FR; object-oriented & other extensions to ML
- Alice, based on ML, 2003, Saarbrücken, DE; optional lazy evaluation, futures, concurrency

Functional Programming

What is common in functional languages?

- Recursive functions
- Recursive data structures
- Handling of functions as data

In the following weeks:

- We will discuss *computing processes* and *data* handled by them
- Our programs - the rule systems describing the processes - will be written in SML
- We will use the Moscow ML compiler & interpreter
- What we learn about abstraction, modelling and program structure will be useful with other programming languages as well

ABSTRACTIONS WITH FUNCTIONS (PROCESSES)

Program elements

Programming Language: more than just a means for instructing a computer to perform tasks. Framework within which we organize our ideas about processes, provides ways for combining simple ideas to form more complex ideas.

- primitive expressions, which represent the simplest entities the language is concerned with,
- means of combination, by which compound elements are built from simpler ones, and
- means of abstraction, by which compound elements can be named and manipulated as units.

Expressions in SML

- atomic: names and constants: ex. `apple`, `486`, `2.0`, `"text"`, `#"A"` `true`
- compound: pl. `482+pear`, `2.3-0.3`, `"te"^"xt"`, `op+(482,4,plum)`, `#"A"< #"a"`

How we combine:

- operators (operator, function)
- operand (formal parameter)
- argument (actual parameter)
- recursion

Examples for using SML

The SML interpreter works in a so called *read-eval-print* loop. The evaluation starts when "`;`" and `ENTER` is pressed.

```
Moscow ML version 2.00 (June 2000)
Enter ‘quit();’ to quit.
- 486;
> val it = 486 : int
- 2.3-0.3;
> val it = 2.0 : real
- "te"^"xt";
> val it = "text" : string
- op+(482,4);
> val it = 486 : int
- #"A"< #"a";
> val it = true : bool
- val it = 486;
> val it = 486 : int
```

Each expression is actually a *value declaration*: if we don’t specify a name, SML binds the name `it` to the value.

Name giving in the global environment

With *value declaration*, we bind a name to a value:

```
- val size = 2;
> val size = 2 : int
- 5*size;
> val it = 10 : int
- val ||| = 3;
> val ||| = 3 : int
- ||| * size;
> val it = 6 : int
```

Remark: `|` and `*` are adhesive symbols, so there must be a space between them.

A name can be:

- : alphanumeric, which consists of the small and capital letters, numbers the _ and the `'` symbols and starts with a letter
- consists of only symbols

Name giving is the simplest abstraction tool in programming languages.
The *name–value* pairs are stored in the „memory" of SML, the so called global environment. Later we'll see that there are local environments as well.

Construction rules of names

- Alphanumeric name: sequence of small letters, capitals, numbers, the apostrophe (`'`) and the underbar (_) symbols, starting with letter or apostrophe.
  - Examples: `agentSmith   Agent_3_Smith   agent'smith   'agent`
  - Names starting with an apostrophe denote type-variables (see later).
- Symbol-name: sequence of the following *adhesive* symbols:

```
! % & $ # + - / : < = > ? @ \ ~ ' ^ | *
```

  - Examples: `++  <->  |||  ##  |=|`
- The following reserved symbols have special roles:

```
( ) [ ] { } , ; . ...
```

- Reserved words (can't be used as names):

```
abstype and andalso as case do datatype else end eqtype exception
fn fun functor handle if in include infix infixr let local nonfix
of op open orelse raise rec sharing sig signature struct structure
then type val where with withtype while : :: :> _ | = => -> #
```

Atomic data types

Type name	*Description*	*Library*
`int`	signed integer	`Int`
`real`	rational (real)	`Real`
`char`	character	`Char`
`bool`	boolean	`Bool`
`string`	string	`String`
`word`	unsigned int	`Word`
`word8`	8 bit unsigned int	`Word8`

Built-in operators and their precedence

In this table, *`wordint`*, *`num`* és *`numtxt`* stand for the followings:

`wordint` = `int`, `word`, `word8`
`num` = `int`, `real`, `word`, `word8`
`numtxt` = `int`, `real`, `word`, `word8`, `char`, `string`

Prec.	*Operator*	*Type*	*Result*	*Exception*
7	`*`	*`num`* `*` *`num`* `->` *`num`*	product	`Overflow`
	`/`	`real * real -> real`	quotient	`Div`, `Overflow`
	`div, mod`	*`wordint`* `*` *`wordint`* `->` *`wordint`*	quotient, remainder	`Div`, `Overflow`
	`quot, rem`	`int * int -> int`	remainder, quotient	`Div`, `Overflow`
6	`+, -`	*`num`* `*` *`num`* `->` *`num`*	sum, difference	`Overflow`
	`^`	`string * string -> string`	concatenated string	`Size`
5	`::`	`'a * 'a list -> 'a list`	list with added element	
	`@`	`'a list * 'a list -> 'a list`	concatenated list	
4	`=, <>`	`''a * ''a -> bool`	equal, not equal	
	`<, <=`	*`numtxt`* `*` *`numtxt`* `-> bool`	less than, less or equal	
	`>, >=`	*`numtxt`* `*` *`numtxt`* `-> bool`	greater than, greater or equal	
3	`:=`	`'a ref * 'a -> unit`	assignment	
	`o`	`('b -> 'c) * ('a -> 'b)-> ('a -> 'c)`	function composition	
0	`before`	`'a * 'b -> 'a`	left argument	

`div` $-\infty$, `quot` rounds towards zero. Results of `div` and `quot`, `mod` and `rem` are equal only if their two operands have the same sign.

Constants

- Signed integer constant (int)

Examples:	0	~0	4	~04	999999	0xFFFF	~0x1ff
Counter-examples:	0.0	~0.0	4.0	1E0	-317	0XFFFF	-0x1ff

- Rational constant (real)

Examples:	0.7	~0.7	3.32E5	3E~7	~3E~7	3e~7	~3e~7
Counter-examples:	23	.3	4.E5	1E2.0	1E+7	1E-7	

- Unsigned integer constant (word)

Examples:	0w0	0w4	0w999999	0wxFFFF	0wx1ff	
Counter-examples:	0w0.0	~0w4	-0w4	0w1E0	0wXFFFF	0WxFFFF

- Character constant (char): the # symbol and a one-character string (see later).

Examples:	#"a"	#"\n"	#"\^Z"	#"\255"	#"\""
Counter-examples:	# "a"	#c	#"""	#'a'	

- Boolean constant (bool): only two constants

Examples:	true	false		
Counter-examples:	TRUE	False	0	1

Constants, escape sequences

- String constant: zero or more printable characters, spaces or *escape-sequences* beginning with the `\` symbol; between double quotes (`"`)
- Escape-sequences

`\a`	Bell (BEL, ASCII 7).
`\b`	Backspace (BS, ASCII 8).
`\t`	Horizontal tabulator (HT, ASCII 9).
`\n`	Newline (LF, ASCII 10).
`\v`	Vertical tabulator (VT, ASCII 11).
`\f`	Form feed (FF, ASCII 12).
`\r`	Carriage-return (CR, ASCII 13).
`\^`*c*	Control-character, where $64 \leq c \leq 95$ (@ … _), and the ASCII code of `\^`*c* is smaller by 64 than the ASCII code of *c*.
`\`*ddd*	The character having ASCII code *ddd* (*ddd* is decimal).
`\u`*xxxx*	The character having ASCII code *xx* (*xx* is hexadecimal).
`\"`	Double quotes (`"`).
`\\`	Backslash (`\`).
`\`$f \cdots f$`\`	Ignored characters $f \cdots f$ zero or more formatting characters (space, HT, LF, VT, FF, CR) symbools.

Evaluating compund expressions

A compund expression is evaluated in two steps (so called *eager or applicative evaluation*):

1. First, the operator is evaluated (operator or function), then the operands (arguments),
2. Second, the operator function is called with the arguments

Note that this startegy is simple because it is defined by recursion. =

Evaluation rules of atomic expressions:

1. Values of constants are the values which they stand for,
2. Built in operators (functions) activate the corresponding native operations
3. Values of names are the values which they are bound in the current environment

Remark: 2. is only a special case of 3.

Példa:

```
(2+4*6)*(3+5+7) = op*(op+(2,op*(4,6)),op+(op+(3,5),7))
```

Expressions can be represented as trees, (see Logic Programming).

Anonymous functions, lambda notation, defining functions

Anonymous function with λ notation: ex. `(fn x => x*x)`
Applying an anonymous function: pl. `(fn x => x*x) 2`

- The `fn` symbol is called *lambda*.
- `x` is the formal parameter of the function (local name).
- `x*x` is the body of the function.
- `2` is the argument (actual parameter) of the function.

Giving name to a function (function declaration):

```
val square = fn x => x * x

val sumOfSquares = fn (x, y) => square x + square y

val f = fn a => sumOfSquares(a+1, a*2)
```

Functions defined by the user can be used the same way as built-in functions.

Further examples on defining functions in SML

Function to produce the successive elements of the 2 bit 1 Hamming-distance code.

- We can define the function with a table:

00	01
01	11
11	10
10	00

```
fn 00 => 01
 | 01 => 11
 | 11 => 10
 | 10 => 00
```

- Variants („clauses”): one variant for each case.
- The `fn` (read: *lambda*) constructs an (anonymous) *function expression*.
- Some uses of the function:
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 10`
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 11`
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 111`
- Pattern-match: one-way unification
- Easily understandable, but not robust: the function is partial (not defined on every element of the domain)

Further examples on defining functions in SML

Incrementing integers modulo n (ex. $n = 5$)

- A function is usually defined with an algorithm, not a table, to avoid too much variants.
- `fn i => (i + 1) mod 5`
  - `i` is the formal parameter, or *bound variable*
- A few uses:
  - `(fn i => (i + 1) mod 5) 2`
  - `(fn i => (i + 1) mod 5) 4`
  - `(fn i => (i + 1) mod 5) 3.0` – Error!
- This function could be defined with two clauses:
  `fn 4 => 0 | i => i + 1`
- The order is important: SML (unlike Prolog) uses only the first matching clause!
- Neither of the functions are robust. Which one is better?

Binding a name to a function value (declaring function values)

- We have seen that names can be bound to function values the same way as to any other values.
  - `val nextCode = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
  - `val incMod = fn 4 => 0 | i => i + 1`
- With syntactic sweetener (`fun`):

  -
    ```
    fun nextCode 00 = 01
      | nextCode 01 = 11
      | nextCode 11 = 10
      | nextCode 10 = 00
    ```

  -
    ```
    fun incMod 4 = 0
      | incMod i = i + 1
    ```
- Applying them on some arguments
  - `nextCode 01`
  - `incMod 4`

Head comment

Let's write declarative *head comment* for all our functions!

-
  ```
  (* nextCode cc = the next element of the 2-bit 1-Hamming distance c
     PRE: cc ∈ {00, 01, 11, 10}
  *)
  fun nextCode 00 = 01
    | nextCode 01 = 11
    | nextCode 11 = 10
    | nextCode 10 = 00
  ```
- `PRE` = *pre*condition
- `PRE: cc ∈ {00, 01, 11, 10}` means: the `nextCode` function's `cc` argument must be in the set `{00, 01, 11, 10}`, else the result is undefined.

-
  ```
  (* incMod i = (i+1) modulo 5
     PRE: 5 > i >= 0
  *)
  fun incMod i = (i+1) mod 5
  ```

Function as a value

- Functions are „first-class citizens” in a functional language: they can freely be passed to other functions, returned as the result of functions, stored in data structures, and so on.
  - The type of a function value is: $\alpha \rightarrow \beta$, where α is the type of the argument, β is the type of the result.
  - The function itself is a value: *function value*
  - Important: the function value is NOT the result of the *application* of the function!
  - Examples:
    - sin (type: *real* $\rightarrow$ *real*)
    - round (type: *real* $\rightarrow$ *int*)
    - $\circ$ (function composition; type: $((\beta \rightarrow \gamma) * (\alpha \rightarrow \beta)) \rightarrow (\alpha \rightarrow \gamma)$)
  - Examples for function application:
    - $\mathrm{round}\ 5.4 = 5$, so the result of the application of the function is of type *int*
    - $\mathrm{round} \circ \mathrm{sin}$ (type: *real* $\rightarrow$ *int*)
    - $(\mathrm{round} \circ \mathrm{sin})1.0 = 1$ (type: *int*)

Functions with two or more arguments

- Functions always have only one argument, but:

  1. We can use compound arguments: pairs, records, list, etc.
     - ex. $f(1,2)$ is the f function applied to the *pair* $(1,2)$
     - ex. $f[1,2,3]$ is the f function applied to the *list* $[1,2,3]$
  2. OR we can apply the function with severan successive steps to arguments:
     ex. $f\ 1\ 2 \equiv (f\ 1)\ 2$ means that
     - in the first step, we apply f to 1, which results in a function
     - in the second step, we apply the result function $(f\ 1)$ to 2 and we get the result of $(f\ 1)\ 2$

- In $f\ 1\ 2$, f is a *partially applicable* function
- It is the programmer's choice to write a function with compound argument, or as a partially applicable function. The difference is only in the syntax ($f\ (1,2)\ \ <=>\ \ f\ 1\ 2$). As we will see later, partially applicable functions are more flexible: they can be applied on some subset of their arguments.
- Infix notation: $x \oplus y \equiv$ the application of the function $\oplus$ to the pair (x,y) as argument.

ABSTRACTION WITH FUNCTIONS AND PROCESSES

Application of functions in SML

- In SML the function name `f` and its argument `e` can be any expressions, which must be separated: `f e`, or `f(e)`, or `(f)e`, or `(f)(e)`
- Separator: zero, one or more *formatting characters* (␣, \t, \n etc.). No formatting character can be used only if using parentheses (i.e. before „(” or after a „)”).
- Important: the separator is the strongest operator which binds to the left.
  ex. `f 1+2` ≡ `(f 1)+2`, `f 1 2` ≡ `(f 1) 2`!!!
- Examples:

```
Math.sin 1.00  (Math.cos)Math.pi  round(3.17)
2 + 3          (real) (3 + 2 * 5)
```

- Classifying functions in SML:
  - Built-in functions, ex. `+`, `*` (both infix), `real`, `round` (both prefix)
  - Library functions, ex. `Math.sin`, `Math.cos`, `Real.fromInt`
  - User-defined functions, pl. `square`, `/\`, `head`

Evaluating Function Applications

An expression with user-defined functions is evaluated similarly to other compound expressions. When we defined that evaluation, de assumed that SML „knows” how to apply functions to arguments. Now we define how SML applies functions:

- All occurences of the formal parameters in the function body are replaced by the corresponding arguments, then
- the function call is replaced with the result of the evaluation of the prepared body

Let’s see how `f 5` is evaluated. Each step, a sub-expression is replaced by an equivalent expression.

`f 5` $\longrightarrow$ `sumOfSquares(5+1, 5*2)` $\longrightarrow$ `sumOfSquares(6, 5*2)` $\longrightarrow$
`sumOfSquares(6, 10)` $\longrightarrow$ `square 6 + square 10` $\longrightarrow$ `6*6 + square 10` $\longrightarrow$
`36 + square 10` $\longrightarrow$ `36 + 10*10` $\longrightarrow$ `36 + 100` $\longrightarrow$ `136`

(`val sumOfSquares = fn (x, y) => square x + square y`; `val square = fn x => x * x`)
This *substitution model – equals replaced by equals* – helps understanding how function application works. This model is applicable if the meaning of a function is always independent from its environment.
Interpreters/compilers usually work with other, more complex – more efficient – models.

Applicative order (eager evaluation), normal order (lazy evaluation)

- When evaluating compound expressions, SML first evaluates the operator, then the arguments, then calls the operator function with the arguments. This order is called *applicative order* or *eager evaluation*
- There are other ways, too. The most important is, when we postpone the evaluation of sub-expressions as long as possible. The evaluation of a sub-expression is needed when it is an argument of a built-in operator or when it is needed for pattern matching in a user-defined function, and of course, the function itself is also needed. This is called *normal order* or *call-by-need* or *lazy evaluation*.

Let's see how `f 5` is evaluated when using normal order evaluation.

`f 5` → `sumOfSquares(5+1, 5*2)` → `square(5+1) + square(5*2)` → `(5+1)*(5+1) + (5*2)*(5*2)` → `6*(5+1) + (5*2)*(5*2)` → `6*6 + (5*2)*(5*2)` → `36 + (5*2)*(5*2)` → `36 + 10*(5*2)` → `36 + 10*10` → `36 + 100` → `136`

- It is proven that for functions, for which the substituion model is applicable, these two strategies lead to the same result.
- Note that when using lazy evaluation, some sub-expressions must be evaluated multiple times.
- Compilers/interpreters help this situation with aliasing (references): identical sub-expressions aren't copied, only referenced: when one occurance is evaluated, so do the others.

Conditional expressions, boolean operators, predicates

- Type name: `bool`. Data constructors: `false`, `true`. Built-in function: `not`.
- *lazy* built-in operators (special language constructs)
  - With three parameters: `if b then e1 else e2`.
    It doesn't evaluate `e2`, if `b` evaluates to `true`, and doesn't evaluate `e1` otherwise.
  - With two parameters:
    `e1 andalso e2` : doesn't evaluate `e2`, if `e1` is `false`.
    `e1 orelse e2` : doesn't evaluate `e2`-t, if `e1` is `true`.
- All three operators are just syntactic sweeteners:
  - `if b then e1 else e2` ≡ `(fn true => e1 | false => e2) b`
  - `e1 andalso e2` ≡ `(fn true => e2 | false => false) e1`
  - `e1 orelse e2` ≡ `(fn true => true | false => e2) e1`
- Typical error: `if exp then true else false`

Conditional expressions, boolean operators, predicates

Let’s see some examples.

```
val absolute = fn x => if      x < 0 then ~x
                       else if x > 0 then x
                       else               0

val absolute = fn x => if      x < 0 then ~x
                       else               x

use "sumOfSquares.sml";

val sumOfSquaresOfTwoLarger =
      fn (x,y,z) =>
           if      x < y andalso x < z then sumOfSquares(y, z)
           else if y < x andalso y < z then sumOfSquares(x, z)
           else                             sumOfSquares(x, y);
```

Predicate is a function which’s return value is `bool`. Example:

```
val isAlphaNum = fn c =>
      #"A" <= c andalso c <= #"Z" orelse
      #"a" <= c andalso c <= #"z" orelse
      #"0" <= c andalso c <= #"9"
```

Conditional expressions, boolean operators, predicates

- Trivial: `andalso` and `orelse` can be expressed with `if-then-else`:
  - `if e1 then e2 else false` ≡ `e1 andalso e2`
  - `if e1 then true else e2` ≡ `e1 orelse e2`
- Let's use `andalso` and `orelse` instead of `if-then-else` where applicable, it makes the code more readable.
- In SML, user-defined functions can't be *lazy*. SML always evaluates the arguments before calling a function.
- Eager equivalents of `andalso` and `orelse`:

```
(* && (a, b) = a /\ b
   && : bool * bool -> bool
*)
fun op&& (a, b) = a andalso b;
infix 2 &&
```

```
(* || (a, b) = a \/ b
   || : bool * bool -> bool
*)
fun op|| (a, b) = a orelse b;
infix 1 ||
```

- `infix` *`prec name1 name2 ...`* : turns *`name1 name2 ...`* functions to *infix* operators with *`prec`* precedence and binding to the left.

Calculating square roots with Newton’s method

- Functions in a functional languages are similar to functions in mathematics: they return values depending on the values of one or more arguments. There is a big difference: functions in functional languages must be efficiently computable.
- Let’s see the definition of the square root function in math:
  $\sqrt{x} = y$, where $y \geq 0$ and $y^2 = x$.
- This equation system is adequate for checking whether a number is a square root of another, but is it adequate for computing square roots?
- Functions in mathematics declare a property, functions in functional languages also tell, *how to produce* the value. That is, declarative programming is declarative only when compared to imperative programming (WHAT? <=> HOW?).
- A well-known method for computing square roots is *Successive Approximation*: If y is an approximation for the square root of x, then the average of y and x/y is a better approximation. The process produces successive approximations of $\sqrt{x}$ and stops when the approximation is considered good enough.
- Let’s write this algorithm in SML.

Calculating square roots with Newton’s method

```
val rec sqrtIter =
      fn (guess, x) => if goodEnough(guess, x)
                       then guess
                       else sqrtIter(improved(guess, x), x)
```

- The `rec` keyword means that the value declaration is *recursive*: the declared value (name) will be used in its own declaration.
- Our strategy is a good example for top-down design. In the beginning, we don’t care about details, we assume that everything we need is already available, and we implement them later.

So, we have to define a few details:

```
val improved = fn (guess, x) => average(guess, x/guess)
val average = fn (x, y) => (x+y)/2.0
val goodEnough = fn (guess, x) => abs(square_r guess - x) < 0.001
val square_r = fn (x : real) => x * x
```

Finally, we have to call out `sqrIter` function with an initial approximation value:

```
val sqrt = fn x => sqrtIter(1.0, x);
```

Calculating square roots with Newton’s method

- Unfortunately, the order of declarations is not adequate: SML requires that the definition of names must precede their first use. (This is not required in some (lazy) functional languages!!!)
- We could reverse the order of our declarations, but then the code wouldn’t reflect our design, our way of thinking.
- We can use *simultaneous declaration*, which means, that several declarations are grouped together: read simultaneously, then processed simultaneously. The names declared together are separated by the `and` keyword.

```
val rec sqrtIter =
      fn (guess, x) => if goodEnough(guess, x)
                       then guess
                       else sqrtIter(improved(guess, x), x)
and improved = fn (guess, x) => average(guess, x/guess)
and average = fn (x, y) => (x+y)/2.0
and goodEnough = fn (guess, x) => abs(square_r guess - x) < 0.001
and square_r = fn (x : real) => x * x
val sqrt = fn x => sqrtIter(1.0, x)
```

Calculating square roots with Newton's method

- Up to this point, our abstraction methods (name giving for names and functions) are useful for handling complex things as units, but aren't useful in hiding details.
- There are several language constructs for hiding details. The most essential is the „expression with local declal declaration", or simply „`let`-expression",
- The `let`-expression is used also for defining (and evaluating) recurring expressions only once.
- Syntax:

```
let d
in e
end
```

where *d* is a non-empty declaration-sequence,
e is a non-empty expression.

Calculating square roots with Newton’s method

```
fun sqrt x =
  let fun sqrtIter (guess, x) = if goodEnough(guess, x)
                                then guess
                                else sqrtIter(improved(guess, x),x)
     and improved (guess, x) =  average(guess, x/guess)
     and average (x, y) = (x+y)/2.0
     and goodEnough (guess, x) = abs(square_r guess - x) < 0.001
     and square_r (x : real) = x * x
  in
      sqrtIter(1.0, x)
  end
```

- In SML the scope and visibility rules are similar to the rules in other languages
- For example, the `x` formal parameter of the `sqrt` function is visible in the functions defined inside `sqrt`, unless they’re covered by a local `x` name. `x` is used everywhere inside `sqrt` as a global name.
- Remark: the `:real` *type constraint* could be omitted: SML can derive the type from the environment.

Calculating square roots with Newton’s method

```
(* A simplified variant: *)
  fun sqrt x =
    let fun sqrtIter guess = if goodEnough guess
                             then guess
                             else sqrtIter(improved guess)
        and improved guess = average(guess, x/guess)
        and average (x, y) = (x+y)/2.0
        and goodEnough guess = abs(square_r guess - x) < 0.001
        and square_r x = x * x
    in
        sqrtIter 1.0
    end;
```

With giving meaningful names to parts of the program, it bacame simpler and easier to understand. With *separation of concerns*,

- programming,
- understanding (for future readers),
- modifying became easier.

Procedures (functions) and processes

- Procedures (functions) are patterns, which define, what the computations do, define the local behaviour of the processes.
- The global behaviour of a process (number of steps, execution time, space consumed) is more difficult to guess.

Linear recursion and iteration

- Simply transforming the mathematical definition of the factorial function to SML gives:

$$0! = 1$$
$$n! = n(n-1)!$$

```
(* PRE : n >= 0 *)
fun factorial 0 = 1
  | factorial n = n * factorial(n-1)
```

- If we apply our *substitution model*, we can see that the process produces and stores all the numbers between n and 1, before executing the first multiplication, it postpones the multiplications. This is a linear recursive process.
- Instead of this, we could multiply 1 with 2, then the partial result with 3, then with 4, and so on, when reaching n, the last partial result would be $n!$. For this, we need an auxiliary formal parameter (or a local variable in imperative languages), which stores the current partial result, and another, which counts from 1 to n. This would be a linear iterative process.

Linear recursion and iteration

```
fun factorial n =
      let fun factIter (product, counter) =
                   if counter > n
                       then product
                   else factIter(product*counter, counter+1)
      in
          factIter(1, 1)
      end
```

- We get a simpler, clearer version of `factIter`, if we decrement the counter from n

```
(* PRE : n >= 0 *)
fun factorial n =
      let fun factIter (product, 0) =
                      product
                 | factIter (product, counter) =
                      factIter(product*counter, counter-1)
      in
          factIter(1, n)
      end
```

Linear recursion and iteration

Linear recursive version:

```
factorial 5
5*factorial 4
5*(4*factorial 3)
5*(4*(3*factorial 2))
5*(4*(3*(2*factorial 1)))
5*(4*(3*(2*(1*factorial 0))))
5*(4*(3*(2*(1*    1       ))))
5*(4*(3*(2*      1      )))
5*(4*(3*       2     ))
5*(4*       6     )
5*     24
120
```

Linear iterative version:

```
factIter (1     ,5)
factIter (1   *5,4)
factIter (5     ,4)
factIter (5   *4,3)
factIter (20    ,3)
factIter (20  *3,2)
factIter (60    ,2)
factIter (60  *2,1)
factIter (120   ,1)
factIter (120 *1,0)
factIter (120   ,0)
120
```

Procedures (Functions) and Processes

- Don’t mix recursive processes and recursive procedures (functions).
- In the case of a recursive function, it is a matter of sintax: the function calls itself.
- In the case of a recursive process, we are talking about how the computation is executed.
- If the function is *right-recursive (tail-recursive, terminal-recursive)* the generated process can be iterative (depending on the goodness of the interpreter/compiler).

We’ll return to the topic of „abstraction with functions”, but now we switch topic: we study the concept of *parametric polymorhpism*, and a polymorph data structure, the list.

POLYMORPHISM

Polymorphism

- Let's examine the identity function: `fun id x = x`
- What is the type of `x`? It can be of any type, its type is denoted by a *type variable*:
  Moscow ML: `val 'a id = fn : 'a -> 'a`
  SML/NJ: `val id = fn : 'a -> 'a`
- `id` is a *polymorph* function, `x` and `id` are poly-typed names.
- Names beginning with an apostrophe are type names, and the one-letter type names are pronounced as the corresponding greek letter (ex. `'a`, *alpha*).
- Let's examine the equality function: `fun eq (x, y) = x = y`.
- What are the types of `x` and `y`? `val ''a eq = fn : ''a * ''a -> bool`.
- Names beginning with two apostrophes are equality-typenames, they stand for types which can be checked for equality. The atomic data types are equality types, except for `real`, and compound data types containing only equality types are equality types.

Types of Polymorphism

Polymorphism shows up in different forms in programming:

- If a polymorph name denotes one single algorithm which can be used on arguments of any type, it is *parametric polymorphism*.
- If an *overloaded* name denotes several different algorithms: one algorithm for each type it is defined for, it is *ad-hoc* or *overloaded polymorphism*.
- A third variation is *polymorphism via inheritence* (see object oriented programming).

LISTS

List: definition, data- and type-constructors

- Definition
  1. A list is a finite sequence of elements having the same type.
  2. A list is a linear recursive data structure, which can be
     - empty,
     - the first element and the list of the other elements.
- Constructors
  - The empty list is denoted by the `nil` name, which is a *data constructor constant*.
  - We usually use the `[]` symbol instead of `nil` (syntactic sugar).
  - The type of `nil` is: `'a list`.
  - `'a` is a *type variable*, `list` is a *type constructor function*.
  - The `::` name is a *data constructor function* (or a *data constructor operator*).
    It creates a new list from an element and a (possibly empty) list.
  - The type of `::` is: `'a * 'a list -> 'a list`.
    It is an infix operator, having precedence 5, binding to the right.
  - The `::` name is pronounced *four-dots* (or *cons* (constructor) for historical reasons).

List: notation, patterns

- Examples
  - Creating lists with data constructors:

```
nil                #"" :: nil
3 :: 5 :: 9 :: nil      = 3 :: (5 :: (9 :: nil))
```

  - Syntactic sweetener for lists:

```
[]                      = nil
[3, 5, 9]               =  3 :: 5 :: 9 :: nil
```

  - Caution! Prolog's list notation is similar, but there are differences:

SML	Prolog		SML	Prolog	
`[]`	`[]`	same	`(x::xs)`	`[X|Xs]`	different
`[1,2,3]`	`[1,2,3]`	same	`(x::y::ys)`	`[X,Y|Ys]`	different

- Patterns

  Expressions built with `[]` and `nil` data constructor constants and with the `::` data constructor operator, and the `[x1, x2, ..., xn]` list-notation can be used in patterns (in the head of functions).

List: head (`hd`), tail (`tl`)

- The first element of a (non-empty) list is its *head*.

```
(* hd : 'a list -> 'a
   hd xs = the first element of the non-empty list xs (head of xs)
*)
fun hd (x :: _) = x;
```

- The list containing the elements of a list, after the first (its *tail*).

```
(* tl : 'a list -> 'a list
   tl xs = the list containing the elements of a list xs, after the
   (the tail of x)
*)
fun tl (_ :: xs) = xs;
```

- `hd` and `tl` are partial functions. The `Lists.hd` and `List.tl` functions return an `Empty` exception when applied to empty lists.
- The _ (underbar) is the so called wildcard symbol, the match-anything pattern. In contrast to Prolog, it can only be used in function heads.

Handling lists: length (`length`), sum of elements (`isum`), product of elements (`rprod`)

- The `length` function returns the length of a list.

```
(* length : 'a list -> int
   length zs = the number of elements in zs *)
fun length []          = 0
  | length (_ :: zs) = 1 + length zs
```

- The `isum` function returns the sum of elements in an integer list.

```
(* isum : int list -> int
   isum ns = the sum of elements in ns *)
fun isum []          = 0
  | isum (n :: ns) = n + isum ns
```

- The `rprod` function returns the product of elements in a real list.

```
(* rprod : real list -> real
   rprod xs =  product of elements in xs *)
fun rprod []          = 1.0
  | rprod (x :: xs) = x * rprod xs;
```

Examples: hd, tl, length, isum, rprod

- hd, tl

Expression	Result of evaluation
`List.hd [1, 2, 3];`	`> val it = 1 : int`
`List.hd [];`	`! Uncaught exception:`
	`! Empty`
`List.tl [1, 2, 3];`	`> val it = [2, 3] : int list`
`List.tl [];`	`! Uncaught exception:`
	`! Empty`

- length, isum, rprod

Expression	Result of evaluation
`length [1, 2, 3, 4];`	`> val it = 4 : int`
`length [];`	`> val it = 0 : int`
`isum [1, 2, 3, 4];`	`> val it = 10 : int`
`isum [];`	`> val it = 0 : int`
`rprod [1.0, 2.0, 3.0, 4.0];`	`> val it = 24.0 : real`
`rprod [];`	`> val it = 1.0 : real`

map: Aplying a function to each element of a list

- Example: calculate the square root of each number in a list.

```
load "Math";
map Math.sqrt [1.0, 4.0, 9.0, 16.0] = [1.0, 2.0, 3.0, 4.0];
```

- In general: `map f` $[x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$
- The definition of `map` is (`map` is a polymorph function):

```
(* map : ('a -> 'b) -> 'a list -> 'b list
   map f xs = the list of elements in xs mapped by f
*)
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs;
```

- The type of `map` is (because the −> type operator binds to the right!):

```
('a -> 'b) -> 'a list -> 'b list ≡ ('a -> 'b) -> ('a list -> 'b list)
```

- The `map` function is a *partially applicable*, and *higher order* function: if applied to a `'a -> 'b` function, results to a `'a list -> 'b list` function. The resulted function, when applied to a `'a list`, results in a `'b list`.
- This function is pre-defined in SML.

Proving (informally) the correctness of recursive functions, with `map` as an example

- We have to prove that the recursive function is
  - functionally correct: the result is what we expect
  - the evaluation of the function is finite (it does not fall in an „infinite recursion”)
- The proof is with structural induction by length (similar to *mathematical induction*)

```
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs
```

  - Let’s assume that `map` works for lists of length $n-1$. (tail of the list, `xs`)
  - Let’s apply `f` to the first element of the list. (head of the list, `x`)
  - Let’s build a new list from `f x` and `map f xs`
  - The result is what we expected, we have proven that if the function works for lists of length $n-1$, then it works for lists of length n.
  - It trivially works for lists of length 0.
  - The evaluation is finite, because
    - every list is finite,
    - When recursively calling `map`, its argument is a one shorter list in every step.
    - The recursion is stopped when the empty list is reached.

A few built-in and library functions

- `explode : string -> char list` – the list of characters in the string

  pl. `explode "abc" = [#"a", #"b", #"c"]`
- `implode : char list -> string` – the string made of the characters in the list

  pl. `implode [#"a", #"b", #"c"] = "abc"`
- Variants of `map`, which work for other compound structures. Examples:
  - `String.map : (char -> char) -> string -> string`
  - `Vector.map : ('a -> 'b) -> 'a vector -> 'b vector`
- In the `Char` library, we can find many useful *predicate* functions. Examples:
  - `Char.isLower : char -> bool` – true for the lower case letters of the alphabet
  - `Char.isSpace : char -> bool` – true for the formatting characters
  - `Char.isAlpha : char -> bool` – true for letters of the alphabet
  - `Char.isAlphaNum : char -> bool` – true for letters of the alphabet and for numbers
  - `Char.isAscii : char -> bool` – true for characters having ASCII code smaller than 128

  pl. `Char.isSpace #"\t" = true; Char.isAlphaNum #"!" = false`

`filter`: elements of the list that satisfy a predicate

- Example: Collect the lower case letters from a string.

  `List.filter Char.isLower (explode "AlTeRnAtInG") = [#"l",#"e",#"n",#"t",#"n"];`

- In general, if `p` x_1 `= true, p` x_2 `= false, p` x_3 `= true, ..., p` x_{2k+1} `= true`, then `filter p [`x_1`,` x_2`,` x_3`, ...,` x_{2k+1}`] = [`x_1`,` x_3`, ...,` x_{2k+1}`]`.

- The definiton of `filter`:

```
(*  filter : ('a -> bool) -> 'a list -> 'a list
    filter p zs = The elements of zs satisfying p
*)
fun filter _ [] = []
  | filter p (x :: xs) =
      if p x then x :: filter p xs else filter p xs;
```

- The type of `filter` (a `->` binds to the right!):

  `filter : ('a -> bool) -> 'a list -> 'a list.`

  That is, if `filter` is applied to a `'a -> bool` function (an `'a` predicate), results in a `('a list -> 'a list)` function, which when applied to an `'a list`, results in an `'a list`.

Finding the maximal element in a list

- The empty list does not have a maximal element,
- The maximal element in a one-element-list is the only element,
- The maximal element of a list having at least two elements is:
  - The maximum of the first element and the maximal element in the tail of the list

```
load "Int";
(* maxl : int list -> int
   maxl ns = The maximal
             element in ns
*)
fun maxl [] = raise Empty
  | maxl [n] = n
  | maxl (n::ns) = Int.max(n, maxl ns)
```

`max` a variant for integers:

```
(* max: int * int -> int
   max (n,m) =
   the maximum of n and m
*)
fun max (n,m) = if n>m
                then n
                else m
```

  - The maximal element in the list with the smaller of the first two elements removed

```
fun maxl' [] = raise Empty
  | maxl' [n] = n
  | maxl' (n::m::ns) = maxl'(Int.max(n,m)::ns)
```

    - Unlike in `maxl`, here the order of clauses is unimportant (the patterns are disjunct).
    - `maxl'` is *tail-recursive*, its space consumption is constant.

Finding the maximal element in a list

- How can we make `maxl` a polymorph function? We define it as a *generic* function: It has an extra parameter, the function which choses the maximum of two elements.

```
(* maxl : ('a * 'a -> 'a) -> 'a list -> 'a
   maxl max zs = the maximal element in zs, according to max
*)
fun maxl max [] = raise Empty
  | maxl max [z] = z
  | maxl max (z::zs) = max(z, maxl max zs)
```

- `max` is always the same, even so we give it as an argument in each recursive call. We can improve efficiency (in some implementations), if we use a let-expression:

```
fun maxl max zs = let fun mxl [] = raise Empty
                        | mxl [y] = y
                        | mxl (y::ys) = max(y, mxl ys)
                  in
                      mxl zs
                  end
```

Variations of max

Variations of max

-
  ```
  (* charMax : char * char -> char
     charMax (a, b) = the maximum of a and b
  *)
  fun charMax (a, b) = if ord a > ord b then a else b;
  ```
  or simply without ord:
  ```
  fun charMax (a : char, b) = if a > b then a else b;
  ```

-
  ```
  (* pairMax : ((int * real) * (int * real)) -> (int * real)
     pairMax (n, m) = lexicografically greater of n and m
  *)
  fun pairMax (n as (n1 : int, n2 : real), m as (m1, m2)) =
        if n1 > m1 orelse n1 = m1 andalso n2 >= m2 then n else m;
  ```

-
  ```
  (* stringMax : string * string -> string
     stringMax (s, t) = the greater of s and t
  *)
  fun stringMax (s : string, t) = if s > t then s else t;
  ```

Concatenating (`append`) and revesring (`nrev`) lists

- Concatenation of two lists (`append` function, or infix operator `@`)

$$[x_1,\ldots,x_m]@[y_1,\ldots,y_n] = [x_1,\ldots,x_{m-1}]@(x_m::[y_1,\ldots,y_n]) = \ldots = [x_1,\ldots,x_m,y_1,\ldots,y_n]$$

  First, we decompose `xs` to its elements, then we append the elements to `ys` backwords, starting from the end of `xs`, because lists can be built on the left. Number of steps: $O(n)$.

```
(* append : 'a list * 'a list -> 'a list
   append(xs, ys) = the elements of xs added to the front of ys *)
fun append ([], ys) = ys
  | append (x::xs, ys) = x::append(xs, ys)
```

- Naive reverse of a list (`nrev`)

$$\texttt{nrev}[x_1,x_2,\ldots,x_m] = \texttt{nrev}[x_2,\ldots,x_m]@[x_1] = \texttt{nrev}[\ldots,x_m]@[x_2]@[x_1] = \ldots = [x_m,\ldots,x_1]$$

  We append the first element as a one-element list to the end. The number of steps: $O(n^2)$.

```
(* nrev : 'a list -> 'a list
   nrev xs = reverse of xs *)
fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x]
```

Reversing lists: example for using `nrev`

- Example for evaluating `nrev`:

  The operators `::` and `@` bind to the right, have precedence of 5.

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

 fun [] @ ys = ys
   | (x::xs) @ ys = x :: xs @ ys (* = (x :: xs) @ ys *)
```

```
nrev([1,2,3,4]) → nrev([2,3,4])@[1] → nrev([3,4])@[2]@[1]
   → nrev([4])@[3]@[2]@[1] → nrev([])@[4]@[3]@[2]@[1]
   → []@[4]@[3]@[2]@[1] → [4]@[3]@[2]@[1]
   → 4::[]@[3]@[2]@[1] → 4::[3]@[2]@[1])
   → [4,3]@[2]@[1]) → 4::([3]@[2])@[1])
   → []@[4]@(3::[2,1] → []@[4]@[3,2,1] → ...
```

`nrev` isn't efficient: the number of steps is $O(n^2)$.

Appending and reversing lists (`revApp` and `rev`)

- Appending the elements of a list in front of another, in reverse order (`revApp`)

```
(* revApp : 'a list * 'a list -> 'a list
   revApp(xs, ys) = the elements of xs in reverse order and ys
*)
fun revApp ([], ys) = ys
  | revApp (x::xs, ys) = revApp(xs, x::ys)
```

  The number of steps for `revApp` is proportional to the length of the list. With its help, we can implement `rev`:

```
(* rev : 'a list -> 'a list
   rev xs = xs reversed
*)
fun rev xs = revApp (xs, [])
```

  A list having 1000 elements is reversed in 1000 steps by `rev` and in $\frac{1000\cdot1001}{2} = 500500$ steps by `nrev`.

- `append` – `@` as an infix operator – and `rev` are available as built-in functions.

COMPOUND DATA STRUCTURES

Record and n-tuple

- From two different types, we can form a record or a pair. ex:

  `{x = 2, y = 1.0} : {x : int, y : real}` és `(2, 1.0) : (int * real)`

- A pair is just syntactic sweetener. ex:

  `(2, 1.0)` ≡ `{1 = 2, 2 = 1.0}` ≡ `{2 = 1.0, 1 = 2}` ≠ `{1 = 1.0, 2 = 2}`.

  In a pair, the order of fields is important! `1` and `2` are field names.

- We can form a record with more than two values. ex:

  `{nev = "Bob", tel = 3192144, age = 19} : {kor : int, nev : string, tel : int}`

  And similar record with numbers as field names:

  `{1 = "Bob", 3 = 3192144, 2 = 19}:  {1 : string, 2 : int, 3 : int}`

  The latter is equivalent of the following *n-tuple*

  `("Bob", 19, 3192144) : (string * int * int)`

  that is

  `(string * int * int)` ≡ `{1 = string, 2 = int, 3 = int}`

- In a record, the order of fields is unimportant, the fields are identified by the field names. In an n-tuple, the order is important, fields are identified by their position.

WEAK AND STRONG ABSTRACTION

Data types: weak and strong abstraction

- Weak abstraction: the name is just a synonim, the parts of the data structure are still visible and available
- Strong abstraction: the name denotes a new entity (object), availability of the parts of the data structures is limited
- `type`: weak abstraction; pl. `type rat = {num : int, den : int}`
  - Gives a new name to a type expression (compare with value declaration).
  - Helps reading the program.
- `datatype`: in combination with modules: strong abstraction

  ex: `datatype 'a option = NONE | SOME of 'a`
  - Creates a new entity.
  - Can be recursive and polymorph.

Data types: enumerated and polymorph types with `datatype` declaration

`datatype answer = Yes | No`	Enumerated type.
`datatype answer3 = yes | no | maybe`	Enumerated type.
`datatype 'a option = NONE | SOME of 'a`	Polymorph type.

- The new entities: `Yes`,`No`, `yes`, `no`, `maybe`, `NONE` are values, *data constructor (constant)s*. `SOME` is *data constructor function*. Data constructors are in the same *namespace* as the other (value) names.

- The new entities: `answer` and `answer3` are *type constructor (constant)s*. `option` is a *type constructor function (postfix operator)*. Type constants (`answer`) and type functions applied to other types (`int option`, `'a option`) are type expressions. Type constructors are in a different *namespace* as value names.

- Of course, data constructors have a type as well. ex:

`Yes   : answer`	`NONE   : 'a option`
`No    : answer`	`SOME : 'a -> 'a option`

- Example for function handling a user-defined datatype:

```
fun invert No = Yes | invert Yes = No
```

WORTH TO MENTION

Worth to mention

- Unit value and type

  The `()` or `{}` symbol is a *0-tuple*, its type is `unit`. The `0-tuple` is the only value of the type `unit`. The `unit` type is the identity element of type operations.

- The `print` function

  If the `print` function (`string -> unit`) when applied to a `string`, the result is a `0-tuple`, and a side effect is that the string is written to the standard output.

- The `(e1;e2;...;en)` sequential expression's result is the value of `en`. If `e1`, `e2`, ... have side effects, it will be carried out. `(e1; e2; e3)` is equivalent to the following `let`-expression:

```
let val _ = e1 val _ = e2 in e3 end
```

- The value of the `e1 before e2 ...  before e3` expression is equivalent to thr value of `e1`. If `e1`, `e2`, ... have side effects, it will be carried out. `e1 before e2 before e3` is equivalent to the following `let`-expression:

```
let val e = e1 val _ = e2 val _ = e3 in e end
```

ABSTRACTION WITH FUNCTIONS

Tree recursion

- So far, we have met linear recursive and linear iterative processes (calculating factorials in several ways).
- Now let's see examples for *tree recursion*: let's generate the sequence of Fibonacci-numbers.
- A Fibonacci-number is the sum of the two previous Fibonacci-numbers:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

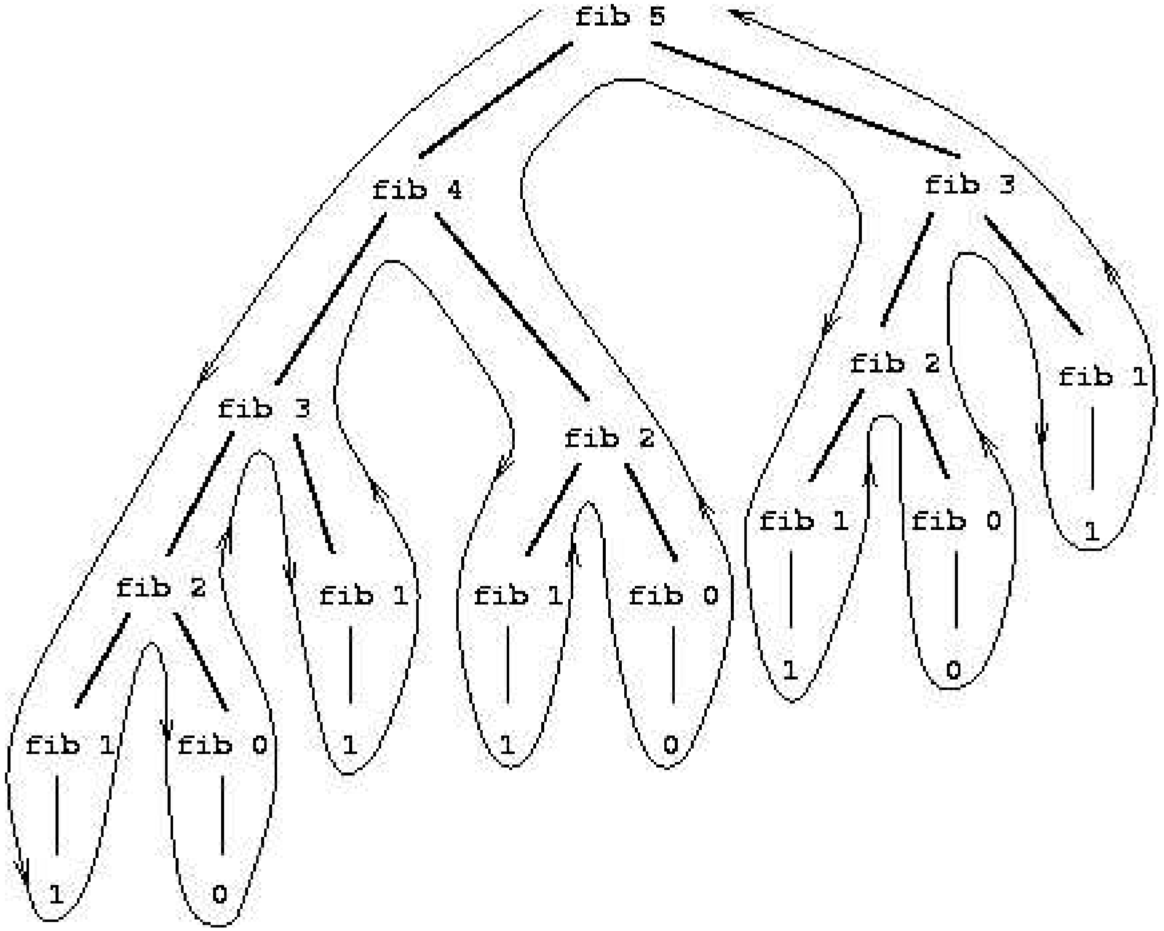
- The definition of Fibonacci-numbers can be easily transformed into an SML function:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2), \text{ ha } n > 1$$

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

  Remember: the third clause of the `fib` function must be the last, because the `n` pattern matches anything.
- A recursive function (procedure) with more than one recursive call is called *tree recursion*.
- The figure on the next slide will show how this tree-recutsive function evaluated.

Tree recursion



- We get `fib 5` with calculating `fib 4` and `fib 3`, `fib 4` with `fib 3` and `fib 2` and so on.

Tree recursion

- The previous program is good for introducing tree recursion, but almost totally unusable for generating Fibonacci-numbers.
- Note that for example we calculated `fib 3` two times, doing about the third of the work unnecessarely.
- It can be proven that for $F(n)$, a tree with exactly $F(n+1)$ leafs must be fully explored, where the leafs are the $F(0)$ and $F(1)$ calls.
- $F(n)$ is an exponential function of n. To be more precise, $F(n)$ is an integer close to $\Phi^n/\sqrt{5}$, where $\Phi = (1-\sqrt{5})/2 \approx 1.61803$, the so called *golden section* ratio number. Φ satisfies the $\Phi^2 = \Phi + 1$ equation.
- The number of required steps grows togerher with $F(n)$, exponentially with n. In the meanwhile, the memory consumed is only proportional to n, because only one root-leaf path has to be kept in memory.
- In general, it is true that the number of steps is proportional to the leafs and nodes of the tree, while the memory usage is only proportional to the maximal depth of it.

Tree recursion

- Fibonacci-numbers can be generated by a linear-iterative process.

  If there are a and b variables with initial values $F(1) = 1$ and $F(0) = 0$ respectively, and we iteratively apply the $a \leftarrow a + b$ and $b \leftarrow a$ transformations, after n iterations, $a = F(n + 1)$ and $b = F(n)$ will hold.

- It is an imperative algorith, which is straightforward to implement in imperative languages. Let's see how it can be implemented in SML.

```
fun fib n = let fun fibIter (i, b, a) =
                      if i = n
                         then b
                         else fibIter(i+1, a, a+b)
            in
                fibIter(0, 0, 1)
            end
```

- Note that the transformation is the same straigtforward: loops (iterations) are (tail-)recursive auxiliary functions, local variables are parameters of the functions, initial values appear as arguments of the auxiliary functions.

Tree recursion

- *Pattern matching* can be used if we decrement `i` from n to 0.

```
fun fib n = let
                fun fibIter (0, b, a) = b
                  | fibIter (i, b, a) = fibIter(i-1, a, a+b)
            in
                fibIter(n, 0, 1)
            end
```

- Warning: the order of clauses is important, as the pattern aren't disjunct.
- Note that in contrast to imperative style, the place where the "variables" are "changed" is strict: in the recursive function call. In an iteration in an imperative language, you can scatter your variable assignments anywhere in the loop, and if a variable is unchanged, then there is no point where this can be seen: the absence of the assignment is hard to notice.

Tree recursion

- In the Fibonacci-example, the numer of steps was exponential to n in the tree-recursion example, and was proportional to n in the linear-iterative version.
- It would be a mistake to conclude that tree recursion is useless. When dealing with hierarchical data structures, for example, working with trees, tree recursion is natural and useful.
- Tree recursion can be also useful when impementing a first version of the solution for a problem: it is easy to implement, easy to reason about the program.
- In our example, it was easy to transform the mathematical definition to a program, and after examined and understood, was easy to change it to become efficient, too. Tree recursion helps understanding a problem and solution.

There was a need for only a small idea to transform our program to iterative.
For this example, it isn't that simple:

- How many ways can you change one dollar to 50-, 25-, 10-, 5- and 1-cent coins?
- In general: How many ways can you change a given amount of money to given coins?

Tree recursion: changing money

Let's assume that we have n different type of coins, in a descending order. Then the number of ways in we can change a dollars is, we calculate

- how many ways we can change a without using the first (the biggest) coin (having value d), and we add
- how many ways we can change $a - d$ with all the coins we have. In other words: how many ways we can change a in such a way that we use the first coin at least once.

This problem can be solved by recursion, as the problem can be reduced to smaller problems: changing smaller amount of money with less coins. The base cases can be the following:

- If $a = 0$, the number of ways is 1.

  (If we have 0 dollars, it can be changed only one way: with 0 coins)
- If $a < 0$, the number of ways is 0.
- If $n = 0$, the number of ways is 0.

In our example, we implemented the `firstDenomination` function with enumeration.
It would be more flexible to implement it with using a list.

Tree recursion: changing money

```
fun countChange amount =
  let (* cC amount kindsOfCoins = the number of ways amount can be changed with
                                  using kindsOfCoins coins *)
      fun cC (amount, kindsOfCoins) =
            if amount < 0 orelse kindsOfCoins = 0 then 0
            else if amount = 0 then 1
            else cC (amount, kindsOfCoins - 1) +
                 cC (amount - firstDenomination kindsOfCoins, kindsOfCoins)
      and firstDenomination 1 = 1
        | firstDenomination 2 = 5
        | firstDenomination 3 = 10
        | firstDenomination 4 = 25
        | firstDenomination 5 = 50
  in
     cC(amount, 5)
  end;

countChange 10 = 4; countChange 100 = 292;
```

Practice:

- Change `firstDenomination` function to have the values of coins in a list.
- Change `cC` function to use pattern matching.

Exponents

- In processes seen so far, the number of evaluation (execution) steps grew linearly, or exponentially with n, the number of data units. In the next example, the number of steps is proportional to the logarithm of n.
- The definition of b raised to the n-th power is also easy to transform into SML.

$$\begin{aligned} b^0 &= 1 \\ b^n &= b \cdot b^{n-1} \end{aligned}$$

```
fun expt (b, 0) = 1
  | expt (b, n) = b * expt(b, n-1)
```

- The outcome process is linear recursive. Execution requires $O(n)$ steps and memory of size $O(n)$.
- It is similarly easy to write the linear iterative version of computing the factorial.

```
fun expt (b, n) =
      let fun exptIter (0, product) = product
            | exptIter (counter, product) =
                                    exptIter (counter-1, b * product)
      in
          exptIter(n, 1)
      end
```

- Execution requires $O(n)$ steps and memory of size $O(1)$ i.e. constant.

Exponents (cont’d.)

- Fewer steps suffice, provided we take advantage of the equations below:

$b^0 = 1$

$b^n = (b^{n/2})^2$, if n is even

$b^n = b \cdot b^{n-1}$, if n is odd

```
fun expt (b, n) =
  let fun exptFast 0 = 1
        | exptFast n =
            if even n
            then square(exptFast(n div 2))
            else b * exptFast(n-1)
      and even i = i mod 2 = 0
      and square x = x * x
  in  exptFast n  end
```

- Steps and memory needed is proportional to $O(\lg n)$. Iterative version with constant storage req.:

```
fun expt (b, 0) = 1  (* Not to be omitted! Why not?  *)
  | expt (b, n) = let fun exptFast (1, r) = r
                        | exptFast (n, r) =
                          if even n then exptFast(n div 2, r*r)
                          else           exptFast(n-1, r*b)
                      and even i = i mod 2 = 0
                  in  exptFast(n, b)  end
```

LISTS

A given number of elements from the beginning and end of the list (`take`, `drop`)

- Let $xs = [x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}]$, then `take(xs, i)` $= [x_0, x_1, \ldots, x_{i-1}]$ and `drop(xs, i)` $= [x_i, x_{i+1}, \ldots, x_{n-1}]$.
- An implementation of `take` (is it tail-recursive? can it be made tail-recursive? is it robust?)

```
(* take : 'a list * int -> 'a list
   take (xs, i) = if i < 0, xs;, if i >= 0,
                  list consisting of the first i elements of xs *)
fun take (_, 0)      = []
  | take ([], _)     = []
  | take (x::xs, i) = x :: take(xs, i-1)
```

- An implementation of `drop` (is it tail-recursive? can it be made tail-recursive? is it robust?)

```
(* drop : 'a list * int -> 'a list
   drop(xs, i) = if i < 0, xs; if i >= 0,
                 list obtained by leaving the first i elements of xs *)
fun drop ([], _)     = []
  | drop (x::xs, i) = if i > 0 then drop (xs, i-1) else x::xs
```

- Their library versions – `List.take`, and `List.drop` – when applied to the list `xs`, raise an exception named `Subscript` if $i < 0$ or $i >$ `length xs`.

Reduction of a list by dyadic operations

Let’s recall the two versions of the `maxl` function, which finds the maximal element of an integer list:

- The version of `maxl` performing right-to-left reduction (non-tail-recursive)

```
(* maxl : int list -> int
   maxl ns = the maximal element of the integer list ns
*)
fun maxl [] = raise Empty
  | maxl [n] = n
  | maxl (n::ns) = Int.max(n, maxl ns)
```

- The version of `maxl` performing left-to-right reduction (tail-recursive)

```
(* maxl' : int list -> int
   maxl' ns = the maximal element of the integer list ns
*)
fun maxl' [] = raise Empty
  | maxl' [n] = n
  | maxl' (n::m::ns) = maxl'(Int.max(n,m)::ns)
```

- As seen in this example, reducing a list by a dyadic operation is a recurring task.
- They have in common, that we need to get one value from n values, that’s why we’re talking about *reduction*.

Reduction of a list by dyadic operations (`foldr`, `foldl`)

- A dyadic operation (more precisely, a *function* of *prefix* position, *applicable to a pair*) is performed on a list from right to left by `foldr`, from left to right by `foldl`. Examples for computing sum and product:

```
foldr op* 1.0 [] = 1.0;                          foldl op+ 0 [] = 0;
foldr op* 1.0 [4.0] = 4.0;                       foldl op+ 0 [4] = 4;
foldr op* 1.0 [1.0, 2.0, 3.0, 4.0] = 24.0;       foldl op+ 0 [1, 2, 3, 4] = 10;
```

- Let $\oplus$ denote an arbitrary dyadic infix operator. Then

$$\texttt{foldr op}\oplus\ \texttt{e [}x_1\texttt{, }x_2\texttt{, ..., }x_n\texttt{] = (}x_1 \oplus \texttt{(}x_2 \oplus \ldots \oplus \texttt{(}x_n \oplus \texttt{e) ...))}$$
$$\texttt{foldr op}\oplus\ \texttt{e [] = e}$$
$$\texttt{foldl op}\oplus\ \texttt{e [}x_1\texttt{, }x_2\texttt{, ..., }x_n\texttt{] = (}x_n \oplus \ldots \oplus \texttt{(}x_2 \oplus \texttt{(}x_1 \oplus \texttt{e)) ...)}$$
$$\texttt{foldl op}\oplus\ \texttt{e [] = e}$$

- The operand `e` of operation $\oplus$ acts as the (right) identity element in some frequently used operations such as addition, multiplication, conjunction (logical ”and”), and alternation (logical ”or”).

- In the case of associative operations, the results of `foldr` and `foldl` are identical.

Examples for using `foldr` and `foldl`

- `isum` returns the sum of an integer list, `rprod` returns the product of a real list.

```
val isum = foldr op+ 0;                    val rprod = foldr op* 1.0;
val isum = foldl op+ 0;                    val rprod = foldl op* 1.0;
```

- The `length` function can also be defined by `foldl` or `foldr`. As dyadic operation we use an auxiliary function that *doesn't use* its first parameter.

```
(* inc : 'a * int -> int
   inc (_, n) = n + 1 *)
fun inc (_, n) = n + 1;

(* lengthl, lengthr : 'a list -> int *)
val lengthl = fn ls => foldl inc 0 ls;
fun lengthr ls = foldr inc 0 ls;
val lengthl = foldl inc 0;

lengthl (explode "tengertanc");
lengthr (explode "hajdu sogor");
```

List: The definition of `foldr` and `foldl`

- `foldr op`$\oplus$ `e [`x_1`, `x_2`, ..., `x_n`] = (`$x_1 \oplus$ `(`$x_2 \oplus$ `...` $\oplus$ `(`$x_n \oplus$ `e) ...))`

  `foldr op`$\oplus$ `e [] = e`

```
(* foldr f e xs = the result of the dyadic operation f with
                  identity element e, applied to the elements
                  of xs, proceeding from right to left
   foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldr f e (x::xs) = f(x, foldr f e xs)
  | foldr f e [] = e;
```

- `foldl op`$\oplus$ `e [`x_1`, `x_2`, ..., `x_n`] = (`$x_n \oplus$ `...` $\oplus$ `(`$x_2 \oplus$ `(`$x_1 \oplus$ `e)) ...)`

  `foldl op`$\oplus$ `e [] = e`

```
(* foldl f e xs = the result of the dyadic operation f with
                  identity element e, applied to the elements
                  of xs, proceeding from left to right
   foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldl f e (x::xs) = foldl f (f(x, e)) xs
  | foldl f e [] = e;
```

Further examples for using `foldr` and `foldl`

- `foldr` and `foldl` appends elements of one list before another when the constructor function *cons*, i.e. `op::` is used as the dyadic operation.

  `foldr op:: ys` $[x_1, x_2, x_3]$ = $(x_1$ `::` $(x_2$ `::` $(x_3$ `:: ys)))`

  `foldl op:: ys` $[x_1, x_2, x_3]$ = $(x_3$ `::` $(x_2$ `::` $(x_1$ `:: ys)))`

- `::` is not associative, so the results of `foldl` and `foldr` are different!

```
(* append : 'a list -> 'a list -> 'a list
   append xs ys = list obtained by appending xs before ys *)
fun append xs ys = foldr op:: ys xs;

(* revApp : 'a list -> 'a list -> 'a list
   revApp xs ys = list obtained by appending
                  the reversed xs before ys *)
fun revApp xs ys = foldl op:: ys xs;
```

`append [1, 2, 3] [4, 5, 6] = [1, 2, 3, 4, 5, 6];` (cf. Prolog: `append`)

`revApp [1, 2, 3] [4, 5, 6] = [3, 2, 1, 4, 5, 6];` (cf. Prolog: `revapp`)

Further examples for using `foldr` and `foldl`

- Two implementations of `maxl`

```
(* maxl : ('a * 'a -> 'a) -> 'a list -> 'a
   maxl max ns = maximal element of the list ns according to max
*)

(* non-tail-recursive *)

fun maxl max []        = raise Empty
  | maxl max (n::ns) = foldr max n ns

(* tail-recursive *)

fun maxl' max []        = raise Empty
  | maxl' max (n::ns) = foldl max n ns
```

Example for using lists: creating runs

- Run is a list whose elements satisfy a given condition.
- The given condition is passed to the function creating the run as a *predicate* to be applied to the previous and current element.
- Our task: to write an SML function that returns a list of runs composed of subsequent elements of a list (preserving the original sequence of elements).
- In the first version, we write an auxiliary function to create the first (prefix) run of a list, and another one to create the rest of the list.
- The auxiliary function `run` has two arguments: the first one is a predicate implementing the desired condition, the second one is a pair. The first member of the pair is the previous element, the second member is the list whose run beginning with the previous element `run` must create.
- The two arguments of the auxiliary function `rest` are the same as the arguments of `run`. It must return the list it obtains by removing the first run from the list passed as the second member of the pair.
- On the next slides, the auxiliary functions `run` and `rest`, as well as different versions of the function `runs` can be seen.

Example for using lists: creating runs (cont’d.)

- First version: creating run and rest with two functions

```
(* run : ('a * 'a -> bool)  -> ('a * 'a list) -> 'a list
   run p (x, ys) = the first (prefix) run of x::ys satisfying p *)
fun run p (x, [])     = [x]
  | run p (x, y::ys) = if p(x, y) then x :: run p (y, ys) else [x]

(* rest : ('a * 'a -> bool) ->  ('a * 'a list) -> 'a list
   rest p (x, ys) = the rest of x::ys after its run satisfying p *)
fun rest p (x, [])              = []
  | rest p (x, yys as y::ys) =
                    if p(x ,y) then rest p (y, ys) else yys

(* runs1 : ('a * 'a -> bool) -> 'a list -> 'a list list
   runs1 p xs = list consisting of runs of xs satisfying p *)
fun runs1 p []        = []
  | runs1 p (x::xs) =
      let val rns = run p (x, xs)
          val rts = rest p (x, xs)
      in
          if null rts then [rns] else rns :: runs1 p rts
      end
```

Example for using lists: creating runs (cont'd.)

- Factors decreasing efficiency

  1. `runs1` goes through the list twice: first `run`, then `rest`,
  2. though `p` never changes, it is passed as a parameter to `run` and `rest`,
  3. none of the functions uses an accumulator.

- Ways to improve

  1. `run` should return a pair, with the run as the first element and the rest as the second; we should use an accumulator for collecting the elements of the run,
  2. `run` should be local inside `runs1`,
  3. the text `if null rts then [rns] else` can be deleted: the recursion terminates at the next call anyway.

Example for using lists: creating runs (cont’d.)

- Second version: creating run and rest with one local function

```
(* runs2 : ('a * 'a -> bool) -> 'a list -> 'a list list
   runs2 p xs = list consisting of runs of xs satisfying p
*)
fun runs2 p []       = []
  | runs2 p (x::xs) =
      let (* run : ('a * 'a list) -> 'a list * 'a list
             run (x, ys) zs = a pair with first member being the first
                               (prefix) run of x::ys satisfying p, appended
                              before zs, second member being the rest of x::ys
          *)
          fun run (x, []) zs           = (rev(x::zs), [])
            | run (x, yys as y::ys) zs = if p(x, y)
                                              then run (y, ys) (x::zs)
                                          else (rev(x::zs), yys);
          val (fs, ms) = run (x, xs) []
      in
          fs :: runs2 p ms
      end
```

Example for using lists: creating runs (cont'd.)

- Third versoin: collecting each run and the list of runs as well

```
(* runs3 : ('a * 'a -> bool) -> 'a list -> 'a list list
   runs3 p xs = list consisting of runs of xs satisfying p
*)
fun runs3 p []       = []
  | runs3 p (x::xs) =
      let (* runs : ('a * 'a list) -> 'a list -> 'a list * 'a list
             runs (x, ys) zs zss = a list consisting of runs of x::ys
                                   satisfying p, appended before zzs
          *)
          fun runs (x, []) zs zss             = rev(rev(x::zs)::zss)
            | runs (x, yys as y::ys) zs zss =
                if p(x, y)
                    then runs (y, ys) (x::zs) zss
                else runs (y, ys) [] (rev(x::zs)::zss)
      in
          runs (x, xs) [] []
      end;
```

Example for using lists: creating runs (cont'd.)

- Examples for applying the functions:

```
run   op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]) =
    [1,9,19];

rest  op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]) =
    [3,4,24,34,4,11,45,66,13,45,66,99];

runs1 op<= [1,9,19,3,4,24,34,4,11,45,66,13,45,66,99] =
    [[1,9,19], [3,4,24,34], [4,11,45,66], [13,45,66,99]];

runs1 op<= [99,1] = [[99], [1]];

runs1 op<= [99] = [[99]];

runs1 op<= [] = [];
```