**Funktionsprogrammering noter**

**HR1 – Getting started.**

* 1. **Values, types, identifiers and declarations**

Interactive interface allows users to enter ex. Arithmetic expressions in line:

2\*3 + 4;;

Note it is followed by 2 semicolons and is terminated with the enter key

The answer contains the value and type of the expression

val it : int = 10

“> “ is output when the system is awaiting input from the user, it is called a **promt.**

The reserved word **val** indicates a value has been computed.

Identifier **it** is a name for the computed value (10 above)

The **type** of the result is **int**

User can name a value with a **declaration**:

**let** price = 125;;

system: val price : int = 125

The identifier is now **price** which has the integer value of 125, -> price is bound to 125

price \* 20;;

val it : int = 2500

**it** is now bound to 2500 which now can be used in expressions

it / price = 20 //this is a question to the system

val it : bool = true

**it** is now bound to the answer true of type bool.

“=” sign in is part of an expression of type bool (true/false) whereas “=” sign in the answer expresses a binding of the identifier **it** to a value. (“=” if **let** is in the input line, it then assigns the value with “=” and is not a bool: let price = 125).

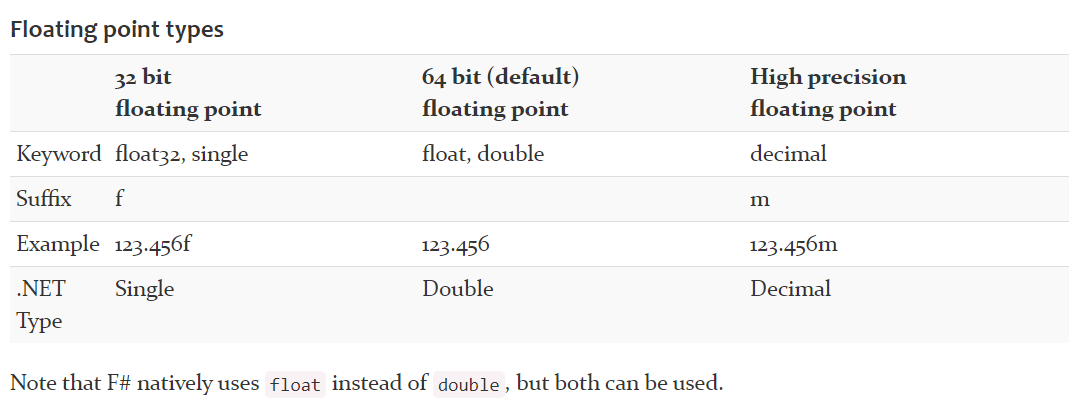
* 1. **Simple function declarations**

A function can be named like an integer constant.

System.Math.PI;;

val it : float = 3.141592654

The type **float** denotes the subset of the real numbers that can be represented in the system, where System.Math.PI is bound to a value of this type.



(float in F# is the same as double)

let circleArea r = System.Math.PI \* r \* r;;

val circleArea : float -> float

The value is a function with the type float -> float, where -> indicates a function type and argument as well as the value of the function has type float

The function circleArea can be applied to different arguments, but the arguments must have the type float, and the result has type float aswell.

circleArea 1.0;;

val it : float = 3.141592654

circleArea (2.0);;

val it : float = 12.56637061

(Note that () around argument is optional).

The identifier System.Math.PI is a composite identifier. The identifier System denotes a *namespace* where the identifier Math is defined. The System.Math denotes a *namespace* where the identifier PI is defined. System and System.Math also denote parts of the .NET library.

* 1. **Anonymous functions. Function expressions**

A function can be created without getting a name, by evaluating a *function expression (also called lambda expression)*. That is an expression where the value is a function.

fun r -> System.Math.PI \* r \* r;;

val it : float -> float = <fun:clo@10-1>

it 2.0;;

val it : float = 12.56647061

The expression *fun* *r -> System.Math.PI \* r \* r* denotes the circle area function and it reads:

“The function of r given by Pi\*r^2”

The reserved word *fun* indicates that a function is defined, the identifier r occurring to the left of the -> is a pattern for the argument of the function, and the *System.Math.PI \* r \* r* is the expression for the value of the function.

**Function expressions with patterns:**

function

| 1 -> 31 //January

| 2 -> 28 //February

| 3 -> 31 //March

| 4 -> 30 //April

| 5 -> 31 //May

| 6 -> 30 //June

| 7 -> 31 //July

| 8 -> 31 //August

| 9 -> 30 //September

| 10 -> 31 //October

| 11 -> 30 //November

| 12 -> 31;; //December

val it : int -> int = <fun:clo@17-2>

This shows the computed value it is a function with the type int -> int

It 2;;

val it : int = 28

The function can be expressed more compactly using a *wildcard pattern “\_”:*

funtion

| 2 -> 28 //February

| 4 -> 30 //April

| 6 -> 30 //June

| 9 -> 30 //September

| 11 -> 30 //November

| \_ -> 31;; //All other months

This function is defined using six clauses. The first clause 2 -> 28 consists of a pattern 2 and a corresponding expression 28.

The last clause contains a wildcard pattern.

A constant, like 2, only matches itself, while the wildcard pattern \_ matches any value.

Example applying 4 gives 30, while applying 7 gives 31.

The function can be compressed even more the the use of an *or-pattern |:*

function

| 2 -> 28 //February

| 4 | 6 | 9 | 11 -> //April, June, September, November

| \_ -> 31 //All other months

;;

Where the or pattern matches any of the values 4, 6, 9, 11, and no others.

let daysOfMonth = function

| 2 -> 28 //February

| 4 | 6 | 9 | 11 -> //April, June, September, November

| \_ -> 31 //All other months

;;

val daysOfMonth : int -> int

daysOfMonth 3;;

val it : int = 30

daysOfMonth 9;;

val it : int = 30

* 1. **Recursion**

When creating a recursion formula, it needs to contain a *base case*, example for factorial base case is !0 = 1

let rec fact = function

| 0 -> 1

| n -> n \* fact(n-1)

val fact : int -> int

The above code contains 2 clauses, and is the recursion formula for n!.

The pattern of the first clause is the constant 0, while the pattern of the second clause is the identifier n.

The patterns are *matched* with integer arguments during the *evaluation* of the function values.

The only value matching the pattern 0 is 0, while in the other pattern every value matches the pattern.

**Evaluation:**

The system uses the declaration fact to evaluate function values in a way that resemble 4!.

To evaluate the above code, the system first searched for a clause that matches the input, in this case 4. First clause is skipped, since 4 != 0, so the second clause is investigated. The value 4 is bound to n, and then *substituted* for n in the right hand side of the clause. So fact 4 evaluates to 4 \* fact (4 – 1).

**Unsuccessfull evaluations:**

The evaluation may not evaluate to a value because of:

* System running out of memory due to long expressions
* The evaluation may involve bigger integers than the system can handle
* The evaluation of an expression may not terminate.

Example applying *fact* to a negative number leads to an infinite evaluation.

It is important when writing recursive functions, that the clauses gets closer and closer to the base case, which stops the recursion, else you will get stuck in infinite loops.

* 1. **Pairs**

Example:

let a = (2.0,3);;

val a = (2.0,3) : float \* int

And patterns pat1 and pat2 is a composite pattern (pat1,pat2). It matches a pair (a1,a2) exactly when pat1 matches a1 and pat2 matches a2:

Let (x,y) = a;;

val y : int = 3

val x : float = 2.0

A pair is a special case of tuples.

x^n is represented as a function power with a pair (x,n) as the argument.

let rec power = function

| (x,0) -> 1.0

| (x,n) -> x \* power(x,n-1);;

val power : float \* int -> float

(DO NOT: let zero = 0;; -> (x,zero), this will match x to u and zero to i from (u,i) pair and match the clause will match any pair.)

The argument is a pair of type float \* int, while the value of the function is of type float.

power a;;

val it : float 8.0

power(4.0,2);;

val it : float 16.0

A function in F# has **one** value and **one** argument. In this case the argument is a pair (u,i) of type float \* int, while the value of the function is type float.

* 1. **Types and type checking**

F# will try to infer a type for each value, expression and declaration entered. If the system can infer a type for the input, then the input is accepted by the system, otherwise it will reject the input with error message: “This expression was expected to have type X but here has type Y”.

Example circleArea has type float -> float, so if we give it circleAre 2 it gives the error with X = float and Y = int, since it expects a float but gets an int.

* 1. **Bindings and environments**

Identifiers can be bound to denote an integer, a floating-point value, a pair or a function. The notions *bindings* and *environments* are used to explain that entities are bound by identifiers.

The *execution* of a declaration, let x = e, causes the identifier x to be bound to the value of the expression e.

Let a = 3;;

val a : int = 3

causes the identifier a to be bound to 3. Denoted by a -> 3.

Further declarations gives extra bindings.

Let b = 7.0;;

Val b : float = 7.0 gives a b-> binding.

|  |
| --- |
| A -> 3 |
| B -> 7.0 |

A collection of bindings is called an environment, which is written like a matrix. 1 x amount of bindings.

Env1 =

Let c = (2, 8);;

Val c : int \* int = (2,8)

Let circleArea r = System.Math.PI \* r \* r;;

Val circleArea : float -> float

|  |
| --- |
| A -> 3 |
| B -> 7.0 |
| C -> (2,8) |
| circleArea -> “the circle area function |

Causes the extension of the environment env1 giving the environment env2 =

The value of an expression is always evaluated in the *actual environment* that contains the bindings of identifiers that are valid at evaluation time.

* 1. **Euclid**
  2. **Evaluations with environments**

The system may create and use temporary bindings of identifiers during the evaluation of expressions.

Let rec gcd = function

| (0,n) -> n

| (m,n) -> gcd(n%m,m);;

Val gcd : int \* int -> int

Gcd contains 2 clauses. Pattern 1: (0,n) with expression n and pattern 2: (m,n) with expression gcd(n%m,m).

There is then 2 cases in the evaluation of an expression gcd(x,y) corresponding to the two clauses.

Gcd(0,y) -> pattern 1 -> binding (n->y)

Gcd(x,y) with x!=0 -> pattern 2 -> binding (m->x) and (n->y) and the system evaluates the righthand side with these bindings.

When pattern 1 is hit, after several recursions, note that m and n’s bindings have changed on every recursion, the last bindings will disappear, and the evaluation will have finished with n->y.

* 1. **Free-standing program**

A free-standing program contains a *main* function of type:

String[] -> int

Preceded by the *entry point attribute:*

…

[<EntryPoint>]

Let main (param: string[]) =

…

*Param* consists of k strings

**HR2 – Values, operators, expressions and functions.**

**2.1 Numbers, Truth values. The *unit* type**

In math, the set of natural numbers are a subset of the set of integers, that is a subset of the rational numbers (i.e., fractions). In F# the type *int* is considered disjoint from the set of values with the type *float*. Floating point numbers are part of the real numbers that are representable in the computer.

Encoding integer and float values in the computer are different. (computer has no instructions for adding integer values to float values).

*Int* values is written as a sequence of digits, possibly prefixed with the minus sign “-“. Real numbers (float) is written using decimal point notation, exponential notation, or both.

0;; -> int = 0

0.0;; -> float = 0.0

0123;; -> int = 123

-7.235;; -> float = -7.235

-388890;; -> int = -388890

1.23e-17;; -> float = 1.23\*10^-17

**Operators:**

*Operator* is a synonym for function.

*Operands* is the components of an operator.

*Monadic operator* is an operator with one operand. (Most monadic operators are used in *prefix* notation where the operator is written in front of the operand.)

*Dyadic operator* is an operator with 2 operands.

Examples of operators on numbers are monadic minus “ – “, and the dyadic operators addition “ + “, subtraction “ – “, multiplication “ \* “ and division “ / “.

The relations “ = “, “ < > “, between numbers are considered to be operators on numbers computing a truth value.

“ – “ has 3 purposes in F#. One in number constants, “-2” where it denotes the sign of the constant. Second in expressions like “- 2” and “-(2+1)” it denotes an application of the monadic minus operator. And last in the expression “1-2” it denotes the dyadic subtraction operator.

1. - - - 1;;

Val it : int = 1

Division is **not** defined on integers. Instead we have the operators “ / “ for quotient and “ % “ for the remainder.

13 / -5;;

Val it : int = -2

13 % -5;;

Val it : int = 3

**Truth values**

There are 2 truth values, *true* and *false* with the type *bool*.

True;;

|  |  |
| --- | --- |
|  | **Logical operators** |
| Not | (unary) negation |
| && | Logical and (conjunction) |
| || | Logical or (disjunction) |

Val it : bool = true

False;;

Val it : bool = false

Functions can have truth values as results. Example function *even* determining whether an integer is even. (n % 2 = 0).

Let even n = n % 2 = 0;;

Val even : int -> bool

A truth-valued function like *even* is called a *predicate.*

Functions on truth values are often called *logical operators*.

The *negation* **not** applies to truth values, and the comparison operators = and <> are defined for truth values.

Not true <> false;; (<> returns true if the left side is not equal to the right side, else returns false)

Val it : bool = false

Evaluations of e1 || e2 and e1 && e2 will only evaluate the expression e2 when needed. So in the expression e1 ||e2, e2 is not evaluated if e1 evaluates to true, and in e1 && e2, e2 is not evaluated if e1 evaluates to false.

1 = 2 && fact -1 = 0;;

Val it : bool = false

Here, the system stops evaluating after 1 = 2, since it is false, and we don’t evaluate fact -1, which would result in a non-termination evaluation.

**The *unit* type:**

There is only one value, written “( )”, of type *unit*. It belongs to the basic types in F#. It is used in the imperative part of F# as a “dummy” result of a computation consisting solely of side effects like input-output or modification of mutable data. There are no operators on the value ( ) of type *unit.*

**2.2 Operator precedence and association**

The monadic operator is written like other function names in front of the argument.

The dyadic operators are written in *infix* notation. That is where the operator is placed between the operands.

Omitting brackets in F# follow two concepts for dyadic operators:

*Operator precedence* – Operators in the same row have the same precedence. Which is higher than that of operators occurring in succeeding rows. Example: \* and / have the same precedence, that is higher than that of +.

A monadic operator has higher precedence than any dyadic operator.

Higher precedence means earlier evaluation:

-2 – 5 \* 7 > 3 – 1 -> ((-2) – (5 \* 7)) > (3-1)

|  |  |
| --- | --- |
| **Operator** | **Association** |
| \*\* | Associates to the right |
| * / % | Associates to the left |
| + - | Associates to the left |
| = <> >= <= | No association |
| && | Associates to the left |
| || | Associates to the left |

*Operator association –*

An operator associating to the left means that operators of the same precedence are applied starting from the left.

1 – 2 – 3 -> (1-2)-3

**2.3 Characters and strings**

A character is a letter, digit or a special character (comma, semicolon etc.).

Characters are encoded in the computer as integer values using the *Unicode alphabet (international standard for encoding characters)*.

‘a’;;

Val it : char = ‘a’

Type is char.

New line, apostrophe, quote and backslash characters are written by means of the *escape sequences*.

|  |  |
| --- | --- |
| **Sequence** | **Meaning** |
| \’ | Apostrophe |
| \” | Quote |
| \\ | Backslash |
| \b | Backspace |
| \n | Newline |
| \r | Carriage return |
| \t | Horizontal tab |

^Character escape sequences^

**Strings**

A string is a sequence of characters. Strings are values of type **string**.

“abcd------“;;

Val it : string = “abcd------”

*Empty string* (“”) is a string that contains no characters.

Strings can be written using *verbatim* string notation where the character @ is placed in front of the first quote.

@”\\\\”;;

Val it : string = “\\\\”

“\\\\”;;

Val it : string = “\\”

Without the verbatim string, the escape sequence \\ for backslash is converted.

**Functions on strings**

String.length “1234”;;

Val it : int = 4

String.length “\”1234””;;

Val it : int = 6

You can use the concatenation function + to join two strings together. The operator + is used in infix mode.

Let text = “abcd---”;;

Val it : string = “abcd

Text + text;;

Val it : string “abcd---abcd---”

Note that the same operator symbol + is used for integer addition and string concatenation. This *overloading* of operator is treated in section 2.5.

A strings numbering start at 0.

s.[i] can be used to extract the the I’th character in the string:

“abc”.[0];;

Val it : char = a

“abc”.[3];;

System.IndexOutOfRangeException:…

To concatenate a string and a character, first the character must be converted into a string with the *string* function:

“abc” + string ‘d’;;

Val it : string = “abcd”

Same is done for converting integer, float and Boolean types into strings.

String -4;;

Val it : string = “-4”

String true;;

Val it : string = “True”

Simple application example:

Let nameAge(name,age) =

Name + “ is “ + (string age) + “ years old”;;

nameAge(“Diana”,15+4);;

val it : string = “Diana is 19 years old”

**2.4 If-then-else expressions**

If-then-else expression has form:

If *exp1* then *exp2* else *exp3*

Where *exp1* is an expression of type *bool*¸ where *exp2* and *exp3* are expressions of the same type.

It evaluates as follows:

If *exp1* evaluates to true, then *exp2* is evaluated. Otherwise, if *exp1* is false, *exp3* is evaluated.

Note that either *exp2 or exp3* will be evaluated. (None will be evaluated if the evaluation of *exp1* does not terminate).

**2.5 Overloaded functions and operators**

A name or symbol for a function or operator is *overloaded* if it has different meanings when applied to arguments or operands of different types.

Examples are + like mentioned earlier, and the \* operator that denotes multiplication on integers (type int) or multiplication on real numbers (type float).

F# solves these ambiguities in the following way:

* If the type can be inferred from the context, then an overloaded operator symbol is interpreted as denoting the function in the inferred type.
* If the type cannot be inferred from the context, then an overloaded operator symbol with a default type will default to this type. The default type is *int* if the operator can be applied to integers.

Example:  
let square x = x \* x;;

Val square : int -> int

You can specify the type of the argument you want it to recognize:

Let square (x:float) = x \* x;;

Val square : float -> float

Or specify the result:

Let square x : float = x \* x;;

Val square : float -> float

Or

Let square x = x \* x : float;;

Val square : float -> float

**2.6 Type inference**

When an expression is entered, the system will try to determine a unique type using so called *type inference*. If this fails, the expression wont be accepted, and an error message is issued.

**Function** is a keyword that indicates to the system, that the type is a function.

**2.7 Functions are first-class citizens**

In functional languages like F#, functions are called *first-class citizens.*

A function can be an argument of another function.

The value of a function can be a function.

This is also known as *higher-order* functions.

**The value of a function can be a function**

Example:

“+” is and infix operator, with a version that is not written between the operands. This *non-fix* version is written “(+)”:

(+);;

Val it : (int -> int -> int) = <fun:it@1>

The type operator “->” associates to the right, so “(+)” has the type:

(+) : int -> (int -> int)

Which shows, that the value of the function (+) is another function with the type int -> int.

Applying (+) to an int then gives a function:

(+) n : int -> int

Example:  
let plusThree = (+) 3;;

Val plusThree : (int -> int)

plusThree 5;;

val it : int = 8

plusThree -7;;

val it : int = -4

The sum of 2 integers m and n can be computed as ((+) m ) n, where the brackets can be omitted, since the *function application associates to the left*:

(+) 1 3;;

Val it : int = 4

**The argument of a function can be a function**

There is an infix operator “<<” denoting the function composition:

Let f = fun y -> y+3;; //f(y) = y+3

Val f : int -> int

Let g = fun x -> x\*x;; //g(x) = x\*x

Val g : int -> int

Let h = f << g;; //h = (f o g)

Val h : int -> int

h 4;; //h(4) = (f o g) (4)

val it : int = 19

* h(4)=4\*4+3=19

Could look like:

((fun y -> y+3) << (fun x -> x\*x)) 4;;

Val it : int = 19

**Declaration of higher-order functions**

We have the higher-order built-in functions like (+) and <<. But one can also declare such functions.

Let weight ro = fun s -> ro \* s \*\* 3.0;;

Val weight : float -> float -> float

A function value weight ro is a function as the expression on the right-hand side of the declaration is a fun-expression. The property of the function value is also visible in the type of weight.

You can make *partial evaluations* of the function weight:

Let waterWeight = weight 1000.0;;

Val waterWeight : (float -> float)

waterWeight 1.0;;

val it : float = 1000.0

waterWeight 2.0;;

val it : float = 8000.0

Higher-order functions may alternatively be defined by supplying the arguments as follows in the let declaration:

Let weight ro s = ro \*s \*\* 3.0;;

Val weight : float -> float -> float

**2.8 Closures**

A *closure* gives the means of explaining a value that is a function. A closure is a triple:

(x, exp, env)

Where x is an argument identifier, exp is the expression to evaluate to get a function value, while env is an environment giving bindings to be used in such an evaluation.

|  |
| --- |
| Ro -> 1000.0 |
| \* -> “the product function” |
| \*\* -> “the power function” |

Closure example from last example:

(s, ro\*s\*\*3.0, )

The environment contains bindings of all identifiers in the expression ro\*s\*\*3.0, except for the argument s.

Note that a closure is a value in F# - functions are first-class citizens.

Example of the environment’s role in the closure:

Let pi = System.Math.PI;;

Let circleArea r = pi \* r \* r;;

Val circleArea : float -> float

This binds the identifier pi to a float value and circleArea to a closure.

Pi -> 3.14159

circleArea -> (r, pi\*r\*r, [pi -> 3.14159])

A fresh binding of pi does not affect the meaning of circleArea that uses the binding pi in the closure:

Let pi = 0;;

circleArea 1.0;;

val it : float = 3.14159

This feature is called *static binding* of identifiers occurring in functions.

**2.9 Declaring prefix and infix operators**

Expressions containing functions on pairs can often be given a more readable form by using infix notation where the function symbol is written between the two components of the argument. Infix form is used for dyadic arithmetic operators.

The *bracket notation* converts from infix or prefix operator to (prefix) function.

* The corresponding (prefix) function for an infix operator *op* is denoted by *(op).*
* The corresponding (prefix) function for a prefix operator *op* is denoted by *(~op)*.

An infix operator is declared using the bracket notation as in the following declaration of an infix exclusive-or operator .||. on truth values:  
let (.||.) p q = (p || q) && not(p &&q);;

Val (.||.) : bool -> bool -> bool

(1 > 2) .||. (2+3 < 5);;

Val it : bool = false

System determines precedence and association of declared operators on the basis of the characters In the operator.

True .||. False && true;;

Is the same as

True .||. (false && true);;

As && has higher precedence than .||.

A prefix character is declared using a leading ~ character.

Let (~%%) x = 1.0 / x;;

Val (~%%) : float -> float

%% 0.5;;

Val it : float = 2.0

**2.10 Equality and ordering**

The *equality* and *inequality* operators “=” and “<>” are defined on any basic type and on strings:

3.5 = 2e-3;;

Val it : bool = false

“abc” <> “ab”;;

Val it : bool = true

But it is not defined on functions (closures):

Cos = sin;;

Stdin(5,1): error FS0001: The type ‘( ^a -> ^a)… does not support the ‘equality’ constraint because it is a function type.

The equality function is automatically extended by F# whenever the user defines a new type:

Let eqText x y =

If x = y then “equal” else “not equal”;;

Val eqText : ‘a -> ‘a -> string when ‘a : equality

This contains a *type variable* ‘a with the *constraint*: when ‘a : equality.

This means that eqText will accept parameters x and y of any type equipped with equality.

eqText 3 4;;

val it : string = “not equal”

eqText ‘ ‘ (char 32);;

val it : string = “equal”

**Ordering**

The *ordering* operators: >, >=, < and <= are defined on values of basic types and on strings.

They correspond to the usual ordering of numbers. The ordering of characters is given by the ordering of the Unicode values, while true > false in the ordering truth values.

Strings are ordered in the *lexicographical* ordering. That is, for two strings s1 and s2 we have that s1 < s2 if s1 would occur before s2 in a lexicon:

‘A’ < ‘a’;; //Uppercase letters precede lower case letters

Val it : bool = true

“automobile” < “car”;;

Val it : bool = true

“” < “ “;;

Val it : bool = true

Empty string precedes any other string in the lexicographical odering.

Let ordText x y = if x > y then “greater”

Else if x = y then “equal”

Else “less”;;

Val ordText : ‘a -> ‘a -> string when ‘a : comparison

The type of x and y contains a *type variable* ‘a with the constraint:

When ‘a : comparison

Indicating that x and y can be of any type equipped with an ordering.

**2.11 Function application operators |> abd <|**

The operator |> means “send the value as argument to the function on the right” while <| means “send the value as argument to the function on the left”.

These two operators have lower precedence than the arithmetic operators

**3. Tuples, records and tagged values (s.43-66)**

Tuples, records and tagged values are compound values obtained by combining values of other types. They are all treated as “first-class citizens”.

**3.1 Tuples (43)**

A tuple is an ordering of n values (v1,v2,…,vn) where n>1, which is called an *n-tuple*.

Example:

(10,true);;

Val it : int \* bool = (10,true)

((“abc”,1),-3);;

Val it : (string\*int) \* int = ((“abc”,1),-3)

2-tuple is also called a pair.

A pair can have components, that also are pairs. Tuples can in general have arbitrary values as components.

A 3-tuple is called a *triple*  and a 4-tuple is called a *quadruple*.

An expression like (true) is not a tuple. -> there is no 1-tuples.

A (true,”abc”,1,-3) is not the same quadruple as ((true,”abc”),1,-3) <-Which is now a triple. (See graph on page 44).

**Tuple expressions (44)**

A *tuple expression* (exp1,exp2,…,exp2) is obtained by enclosing n expressions in parentheses. Each expression has its own type, which corresponds to the *Cartesian Product*.

A tuple expression is evaluated from left to right.

Let tp1 = ((1<2,”abc),1,1-4);;

Val tp1 : (bool\*string) \* int \* int = ((true,”abc”),1,-3)

Let tp2 = (2>1,”abc”,3-2,-3);;

Val tp2 : bool \* string \* int \* int = (true, “abc”, 1,-3)

**Tuples are individual values (44)**

A tuple expression may contain identifiers that are already bound to tuples.

Let t1 = (true,”abc”);;

Val t1 : bool \* string = (true,”abc”)

(See figure 3.2 s. 45)

Let t2 = (t1,1,-3);;

Val t2 : (bool \* string) \* int \* int = ((true,”abc”),1,-3)

The value bound to the identifier t1 is then found as a subcomponent of the value bound to t2.

A fresh binding of t1 is not going to change the value of t2, since the subcomponent (true,”abc”) is its own value and doesn’t depend on future t1 bindings.

**Equality (45)**

Equality is defined for n-tuples of the same type, provided that equality is defined for the components. The equality is defined componentwise.

(“abc”,2,4,9) = (“ABC”,2,4,9);;

Val it : bool = false

(1, (2,true)) = (2-1,(2>1));;

Val it : bool = true

(1, (2, true)) = (1, 2, 2>1);;

Error FS0001: Type mismatch

The types in the last tuples are different, since 1 is int \* (int \* bool) and the other is int \* int \* bool.

So the system recognizes different types, and cant compute the equality of different types.

**Ordering (46)**

The *ordering* operators: >, >=,<,<= and the compare function are defined on n-tuples of the same type, provided ordering is defined on the components.

Tuples are ordered lexicographically: (x1,x2,…,xn) < (y1,y2,…,yn)

(1,”a”) < (1,”ab”);;

Val it : bool = true

(2,”a”) < (1,”ab”);;

Val it : bool = false

Since “a” < “ab” holds, while 2 < 1 does not.

**Tuple patterns (46)**

A tuple pattern represents a graph.

A (x,n) is a *tuple pattern.*

Patterns can be used on the left hand side in a let declaration, which binds the identifiers in the pattern to the values obtained by the pattern matching.

Let (x,n) = (3,2);;

Val x : int = 3

Val n : int = 2

Patterns can contain constants, like (x,0), which is containing the constant 0. This matches any pair (v1,v2) where v2 = 0 and the binding x -> v1 is then obtained.

Let (x,0) = ((3,”a”), 0);;

Val x : int \* string = (3,”a”)

However, the pattern (x,0) is *incomplete* in the sense that it just matches pairs where the second component is 0 and there are other pairs of type t\*int that do not match the pattern.

The system issues a warning when an incomplete pattern is used.

Let (x,0) = ((3,”a”),0);;

Waning FS0025, incomplete pattern matches on this expression.

This warning can be ignored, since the second component of ((3,”a”),0) is indeed 0.

The wildcard pattern can be used I tuple patterns. Every value matches this pattern, but the matching provides no binding:

Let ((\_x),\_,z) = ((1,true),(1,2,3),false);;

Val z : bool = false

Val x : bool = true

A pattern cannot contain multiple occurrences of the same identifier. (x,x) is an illegal pattern.

**3.2 Polymorphism (48)**

*Swap* function interchanging the components of a pair:

Let swap (x,y) = (y,x);;

Val swap : ‘a \* ‘b -> ‘b \* ‘a

Swap (‘a’,”ab”);;

Val it : string \* char = (“ab”,’a’)

Swp ((1,3),(“ab”,true));;

Val it : (string \* bool) \* (int \* int) = ((“ab”,true),(1,3))

Example shows that the function can be applied to all pairs. Which is reflected in the type of swap: ‘a \* ‘b -> ‘b \* ‘a

The type of swap contains two *type variables* ‘a and ‘b. A type containing type variables is called a *polymorphic type* and a function with polymorphic type like swap is called a *polymorphic function*. (Polymorphic means “of many forms”.

Polymorphism is related to overloading, as we in both cases can apply the same function name or operator to arguments of different types. But an overloaded operator denotes *different* F# functions for different argument types. (+ operator both adding int and float types).

**3.3 Example: Geometric vectors (48)**

**3.4 Records (50)**

A *record* is a generalized tuple where each component is identified by a *label* instead of the position in the tuple. The record type must be declared before a record can be made. We may declare a type person as follows:

Type person = {age : int; birthday : int\*int; name : string; sex : string};;

The keyword type indicates it is a *type declaration* and the braces { } indicates a record type. Each identifier (age, birthday…) are called *record labels* and they are considered part of the type.

Let john = {name = “John”; age = 29; sex = “M”; birthday = (2,11);;

Val john : Person = {age = 29; birthday = (2,11); name = “John”; sex = “M”}

This creates the following binding of the identifier john:

John -> {age -> 29, birthday -> (2,11), name -> “John”, sex -> “M”}

A record is a *local environment* packaged in a certain way, which contains a binding of each record label to a corresponding value.

Suffixing the identifier john with the corresponding record label gives a field:

john.birthday;;

val it : int \* int = (2,11)

john.sex;;

val it : string = “M”

**Equality and ordering (51)**

The equality of two records with the same type is defined componentwise from the equality of values associated with the same labels, so the ordering of components in the record is of no importance when entering values:

john = {age = 29; name = “John”; sex = “M”; birthday = (2,11)};;

val it : bool = true

That means two records are equal, if they are of the same type and contain the same local bindings of the labels.

Ordering a record is based on a lexicographical ordering using the ordering of the labels in the record type declaration.

**Record patterns (51)**

A *record pattern* is used to decompose a record into its fields. The pattern:

{name = x; age = y; sex = s; birthday = (d,m)}

Denotes the graph (figure 3.3 s 51).

It generates bindings of the identifiers x,y,s,d and m when matched with a person record.

Let sue = {name=”Sue”; age = 19; sex = “F”; birthday = (24,12)};;

Let {name = x; age = y; sex = s; birthday = (d,m)} = sue;;

Val y : int = 19

Val x : string = “Sue”

Val s : string = “F”

Val m : int = 12

Val d : int = 24

Record patterns are used when defining functions.

Let age {age = a; name = \_; sex =\_; birthday = \_} = a;;

Val age : Person -> int

Let isYoungLady {age=a,sex=s,name=\_;birthday=\_} = a < 25 && s = “F”;;

Val isYoungLady : Person : bool

age john;;

val it : int = 29

isYoungLady john;;

val it : bool = false

isYoungLady sue;;

val it : bool = true

**3.5 Example: Quadratic equations (52)**

**Error handling (53)**

An error can be signaled by using an *exception*. An exception is named by an *exception declaration*.

Exception Solve;;

Exception Solve

Declaration:

Let solve(a,b,c) =

If b\*b-4.0\*a\*c < 0.0 || a = 0.0 then raise Solve

Else ((-b + sqrt(b\*b-4.0\*a\*c))/(2.0\*a), (-b – sqrt(b\*b-4.0\*a\*c))/(2.0\*a);;

Val solve : float \* float \* float -> float \* float

The then branch contains the expression: raise Solve. An evaluation of this expression gives the: FSI\_0015+Solve exception.

The built in exception for error handling is: *failwith: string -> ‘a*

Instead of: then raise Solve -> then failwith “explanation”.

**3.6 Locally declared identifiers (54)**

It is often convenient to use *locally declared* identifiers in function declarations.

In above example, the expression b\*b-4.0\*a\*c is evaluated 3 times (and written 3 times), which in inefficient and harder to read.

Let solve(a,b,c) =

Let sqrtD =

Let d = b\*b-4.0\*a\*c

If d < 0.0 || a = 0.0

Then failwith “Discriminant is negative or a = 0.0”

Else sqrt d

((-b+ sqrtD)/(2.0\*a),(-b-sqrtD)/(2.0\*a));;

Val solve : float \* float \* float -> float \* float

Note that indentation is playing a big role. The 4 lines within “Let sqrtD =” is terminated with the less indented line “((-b+ sqrtD)/(2.0\*a),(-b-sqrtD)/(2.0\*a));;”, which also ends the lifetime binding of d. The let expression constitutes the *scope* of the declaration of d.

(The ((-b+sqrtD..)) must be on the same indentation level as the let sqrtD. This let is terminated by the double ;;).

So a let declaration can contain more than one local declaration.

**3.7 Example: Rational numbers. Invariants (56)**

**Representation. Invariant (57)**

We use the *representation* (a,b) where b>0 and where the fraction a/b is irreducible, that is, gcd(a,b)=1, to represent the rational number a/b. Thus a value (a,b) of type int\*int represents a rational number if b>0 and gcd(a,b) = 1, and we name this condition the *invariant* for pairs representing rational numbers. Any rational number has a unique *normal form* of this kind.

Type Qnum = int \* int //(a,b) where b>0 and gcd(a,b) = 1

Where the invariant is stated as a comment to the declaration.

**Operators (57)**

**3.8 Tagged values. Constructors (58)**

Tagged values are used when we group together values of different kinds to form a single set of values.

Example, representing a circle by its radius r, square by side length a and a triangle by the triple (a,b,c) of its side lengths a,b and c. These 3 can be grouped together to form a *single* collection of *shapes* if we put a *tag* on each representing value.

Type shape = | Circle of float

| Square of float

| Triangle of float\*float\*float;;

Type Shape =

| Circle of float

| Square of float

| Triangle of float\*float\*float

**Constructors and values (59)**

Shape names a type, and Circle, Square and Triangle are bound to *value constructors*. These value constructors are functions and the give a *tagged value* of type shape when applied to an argument.

Example: Circle is a value constructor with type float -> Shape.

This means that Circle r denotes a value of type Shape, for every float r.

It can be observed that Circle 1.2 is a value which is not evaluated further by F#:

Circle 1.2;;

Val it : Shape = Circle 1.2

Since the value in the answer is equal to the expression being evaluated.

**Equality and ordering (59)**

Equality and ordering are defined for tagged values provided that are defined for their components. Two tagged values are equal if they have the same constructor and their components are equal. This corresponds to equality of the graphs represented by the tagged values.

Circle 1.2 = Circle(1.0 + 0.2);;

Val it : bool = true

Circle 1.2 = Square 1.2;;

Val it : bool = false

The sequence of the tags occurring is significant for the ordering. Our example, any circle is smaller than any square, which again is smaller than any triangle due to the order in which the corresponding tags are declared.

Circle 1.2 < Circle 1.0;;

Val it : bool = false

Circle 1.2 < square 1.2;;

Val it : bool = true

Triangle(1.0,1.0,1.0) > Square 4.0;;

Val it : bool = true

**Constructors in patterns (60)**

Constructors can be used in patterns, for example and area function for shapes is declared by:  
let are = function

| Circle r -> System.Math.PI \* r \* r

| Square a -> a \* a

| Triangle(a,b,c) ->

Let s = (a + b + c)/2.0

Sqrt(s\*(s-a)\*(s-b)\*(s-c);;

Val area : Shape -> float

A constructor matches itself only in a pattern match, while other identifiers match any value. Example, Circle 1.2 will match the pattern Circle r but not the other patterns in the function, which will bind r to the value 1.2.

area (Circle 1.2)

(Math.PI \* r \* r, [r -> 1.2])

**Invariant for the representation of shapes (60)**

Some values of type Shape does not represent geometric shapes, as Circle -1.2 does not represent a circle, since a circle cant have a negative radius.

Therefore, there is an *invariant* for this representation of shapes: The real numbers have to be positive, and the triangle inequalities must be satisfied.

This invariant can be declared as a predicate:

Let isShape = function

| Circle r -> r > 0.0

| Square a -> a > 0.0

| Triangle(a,b,c) ->

a > 0.0 && b > 0.0 && c > 0.0 && a < b + c && b < c + a && c < a + b

val isShape : Shape -> bool

**3.9 Enumeration types (62)**

Value constructors doesn’t need to have any argument, so we can create special type declarations:

Type Colour = Red | Blue | Green | Yellow | Purple;;

Type Colour =

| Red

| Blue

| Green

| Yellow

| Purple

Types like Colour are called *enumeration types*, as the declaration of Colour just enumerates five constructers Red, Blue, Green, Yellow, Purple. Where each constructor is a value of type Colour:

Green;;

Val it : Colour = green

The Boolean type is a predefined enumeration type:

Type bool = false | true

**3.10 Exceptions (63)**

Raising an exception terminates the evaluation of a call of a function.

It is possible to *catch* an exception using a try…catch expression:

Let solveText eq =

Try

String(solve eq)

with

| Solve -> “No solutions”;;

Val solveText : float \* float \* float -> string

Library functions (e.g. performing I/O) may raise exceptions that can only be captured using a *match on type*.

**3.11 Partial functions. The option type (64)**

A function f is a *partial* function on a set A if the domain of f is a proper subset of A.

**4. Lists (s. 67 – 91)**

List concept, including list values, patterns and basic operations.

The concept of a list is a special case of a collection.

**4.1 The concept of a list (67)**

A *list* is a finite sequence of values [v0;v1;…;vn-1] of the same type.

[2], [3;2] and [2;3;2] are all lists of integers. A list can contain an arbitrary amount of elements.

The first element of a list, v0, is called its *head* and the rest [v1;…;vn-1] is called its *tail.*

The list [2;3;2] is a tagged pair with tag : : where the head of the list is 2, and the second component, tail, is the list [3;2]. This list is also a tagged pair, with head 3 and tail [2]. Finally the head of the list is [2] and the tail is the empty list [].

**List constants in F# (68)**

Lists can be entered as values:

Let xs = [2;3;2];;

Val xs : int list = [2;3;2]

Let ys = [“Big”; “Mac”]

Val ys : string list = [“Big”; “Mac”]

The types int list and string list, containing the *type constructor* list, indicates that the value of xs is a list of integers and the value of ys is a list of strings.

Lists can have any element type, example a *list of pairs*:

[(“b”,2);(“c”,3);(“e”;5)];;

Val it : (string \* int) list = [(“b”,2);(“c”,3);(“e”;5)]

*List of records:*

Type P = {name : string; age : int}

[{name = “Brown”; age = 25}; {name = “Cook”; age = 45}];;

Val it : P list = [{name = “Brown”; age = 25}; {name = “Cook”; age = 45}]

*List of functions:*

[sin;cos];;

Val it : (float -> float) list = [<fun:it@7>; <fun:it@7-1>]

*Or even list of lists:*

[[2;3];[3];[2;3;2]];;

Val it : int list list = [[2;3];[3];[2;3;2]]

A list can also be components of other values. Example: Pairs containing lists:

(“bce”, [2;3;2]);;

Val it : string \* int list = (“bce”, [2;3;2])

**The type constructor list (68)**

The type constructor list has higher *precedence* than \* and -> in *type expressions*. So string \* int list means string \* (int list). The type constructor list is used in postfix notation, like the factorial function \_! And associates to the left. Int list list means (int list) list.

All elements in a list must have the same type, this is *not* legal:

[“a”;2];;

Error FS0001: This expression was expected to have type string but here has type int

**Equality of lists (69)**

Two lists [x0;x1;…;xm-1] and [y0;y1;…;yn-1] of the same type are *equal* when m = n and xi = yi for all i 0 <= I <= m. The order of the elements as well as repetitions of the same value are significant in a list.

The “=” equality operator can be used on lists, provided the lists are of the same type and provided that the equality operator can be used on values of that element type.

[2;3;2] = [2;3];;

Val it : bool = false

[2;3;2] = [2;3;3];;

Val it : bool = false

Lists containing functions cannot be compared, since equality is not defined for functions in F#.

**Ordering of lists (70)**

Lists of the same type are ordered *lexicographically,* provided there is an ordering defined in the elements.

[x0;x1;…;xm-1] < [y0;y1;…;yn-1] exactly when [x0;x1;…;xk] = [y0;y1;…;yk]

and (k = m- 1 < n-1 or k < min{m-1,n-1} and xk+1 < yk+1) for some k; where 0 <=k <= min{m-1,n-1}.

1. The list xs is a *proper prefix* of ys:

[1;2;3] < [1;2;3;4];;

Val it : bool = true

[´1´; ´2´;´3´] < [´1´; ´2´; ´3´; ´4´];;

Val it : bool = true

The empty list is smaller than eny non-empty list:

[] < [1;2;3];;

Val it : bool = true

[] < [[];[(true,2)]];;

Val it : bool = true

1. The lists agree on the first k elements and xk+1 < yk+1:

[1;2;3;0;9;10] < [1;2;3;4];;

Val it : bool = true

Because 0 < 4.

[“research”; ”articles”] < [“research”; “books”];;

Val it : bool = true

Because “articles” < “books” (a < b).

**4.2 Construction and decomposition of lists (71)**

**The cons operator (71)**

The infix operator : : (called “cons”) builds a list from its head and its tail as shown in figure 4.2 and 4.3. So it adds an element at the front of the list.

Let x = 2 :: [3;4;5];;

Val x : int list = [2;3;4;5];;

Let y = “”::[];;

Val y : string list = [“”]

The cons operator *associates* to the right, so x0::x1::x2 means x0::(x1::xs) where x0 and x1 have the same type and xs is a list with elements of that same type.

Let z = 2::3::[4;5];;

Val z : int list = [2;3;4;5]

**List patterns (71)**

While the cons operator can be used to construct a list from a (head) element and a (tail) list, it is also used in *list patterns*.

There is a list pattern [] for the empty list while patterns for non-empty lists are constructed using the cons operator, that is x::xs matches a non-empty list.

The x::xs gives the binding x->x0 and xs-> [x1;…;xn-1] of the identifiers x and xs.

Let x::xs = [1;2;3];;

Val xs : int list = [2;3]

Val x : int = 1

The :: operator in patterns denotes either decomposing a list with a pattern like x0::x1::xs, or it denotes building a list from smaller parts in an expression like 0::[1;2].

**Simple list expressions (73)**

There is special constructs that can generate lists. First the two simple forms of expressions called *range expressions*: [b..e] [b..s..e] , where b, e and s are expressions having number types.

The range expression [b..e] where e >= b, generates the lists of consecutive elements:

[b;b+1;b+2;…;b+n], where n is chosen such that b + n <= e < b + n + 1.

The range expression generates the empty list when e < b.

Example, the list of integers from -3 to 5 is generated by:

[ -3 .. 5];;

Val it : int list = [-3;-2;-1;0;1;2;3;4;5]

And a list of floats:

[2.4 .. 3.0 \*\*1.7];;

Val it : float list = [2.4;3.4;4.4;5.4;6.4]

Not that 3.0\*\*1.7 = 6.47300784

The expression s in the range [b .. s .. e] is called a step, it can be positive or negative, but not 0.

The generated list will either be ascending or descending depending on the sign of s:

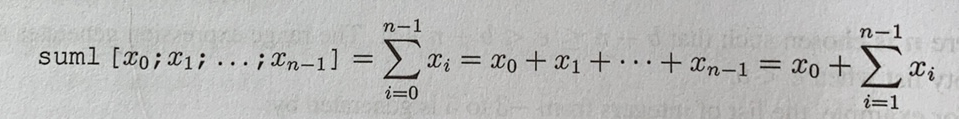
[6 .. -1 .. 2];;

Val it : int list = [6;5;4;3;2]

**4.3 Typical recursions over lists (74)**

**Function declarations with two clauses (74)**

Let us consider the function suml that computes the sum of a list of integer:



We get the recursion formula:

Suml [x0;x1;…;xn-1] = x0 + suml [x1;…;xn-1]

We define the value of the “empty” sum, that is suml[] to be 0 and we arrive at a recursive function declaration with two clauses.

Let rec suml = function

| [] -> 0

|x::xs -> x + suml xs;;

Val suml : int list -> int

When F# evaluates the above function, the system scans the clauses and selects the first clause where the argument matches the pattern:

Suml [1;2]:

1 + suml [2] (x::xs matches [1;2] with x -> 1 and xs -> [2])

1 + (2 + suml[]) (x::xs matches [2] with x -> 2 and xs -> [1])

1 + (2 + 0) (the pattern [] matches the value [])

1 + 2

3

Patterns are convenient in order to split up a function declaration into clauses covering different forms of the argument.

**Function declarations with several clauses (75)**

You can have function declarations with any number (>1) of clauses.

Example: alternative sum of an integer list:

Altsum[x0;x1;…;xn-1] = x0 – x1 + x2 - …. + ((-1)^n-1)\*xn-1

In declaring this function we consider three different forms of the argument:

1. Empty list: altsum[] = 0
2. List with one element: altsum [x0] = x0
3. List with two or more elements: altsum [x0;x1;…;xn-1] = x0-x1+altsum[x2;….;xn-1]

We can then cover these 3 cases with patterns:

Let rec altsum = function

| [] -> 0

| [x] -> x

| x0::x1::xs -> x0 – x1 + altsum xs;;

Val altsum : int list -> int

Altsum [2;-1;3];;

Val it : int = 6

**Layered patterns (76)**

We want to define a function succPairs such that:

succPairs [] = []

succPairs [x] = []

succPairs [x0;x1;…;xn-1] = (x0,x1);(x1,x2);….;(xn-2,xn-1)]

With the pattern from above we get the declaration:

Let rec succPairs = function

| x0 :: x1 :: xs -> (x0,x1) :: succPairs(x1::xs)

| \_ -> [];;

Val succPairs : ‘a list -> (‘a \* ‘a) list

This works, but a smarter declaration can be made avoiding the cons expression x1::xs in the recursive call in the following way:

Let rec succPairs = function

| x0::(x1::\_ as xs) -> (x0,x1) :: succPairs xs

| \_ -> [];;

Val it : ‘a list -> (‘a \* ‘a) list

succPairs [1;2;3];;

val it : (int \* int) list = [(1,2);(2,3)]

The pattern x1::\_ as xs is a *layered pattern*. A layered pattern has the general form:

(pat as id) with pattern pat and identifier id. A value val matches this pattern exactly when the value matches the pattern pat. The matching binds identifiers in the pattern pat as usual with the addition that the identifier id is bound to val.

Matching the list [x0;x1;…;xn-1] with the pattern x0::(x1::\_ as xs) will give the following bindings:

X0 -> x0

X1 -> x1

Xs -> [x1;….;xn-1]

**Pattern matching on result of recursive call (77)**

Pattern matching can be used to split the result of a recursive call into components.

The function sumProd computes the pair consisting of the sum and the product of the elements in a list of integers:

sumProd [x0;x1;….;xn-1] = (x0 + x1 + … xn-1,x0\*x1\*…\*xn-1)

sumProd [] = (0,1)

The declaration is based on the recursion formula:

sumProd [x0;x1;…;xn-1] = (x0 + rSum, x0 \* rProd),

where (rSum,rProd) = sumProd[x1;…;xn-1]

This gives the declaration:

Let rec sumProd = function

| [] -> (0,1)

| x::rest ->

Let (rSum,rProd) = sumProd rest

(x+ rSum, x\*rProd);;

Val sumProd : int list -> int \* int

sumProd [2;5];;

val it : int \* int = (7,10)

**Pattern matching on pairs of lists (78)**

**4.4 Polymorphism (78)**

Studying general kinds of polymorphism appearing frequently in connection with lists. This is done on the basis of three useful list functions that all can be declared using the same structure of recursion as shown in section 4.3

**List membership (79)**

The member function for lists determines whether a value x is equal to one of the elements in a list [y0;y1;…;yn-1], that is:

isMember x [y0;y1;…;yn-1] = (x = y0) or (x=y1) or … or (x=yn-1) = (x = y0) or (isMember x [y1;….;yn-1])

Since no x can be a member of the empty list, we get the declaration:

Let rec isMember x = function

| y::ys -> x=y || (isMember x ys)

| [] -> false;;

Val isMember : ‘a -> ‘a list -> bool when ‘a : equality

isMember can be useful, but is not included in F# library.

The annotation “‘a : equality” indicates that ‘a is an *equality type variable* (section 2.10).

**Append and reverse. Two built-in functions (79)**

The infix operator @(called ‘append’) joins two lists of the same type:

[x0;x1;…;xm-1] @ [y0;y;1;…;yn-1] = [x0;x1;…;xm-1;y0;y1;…;yn-1]

The function List.rev (called reverse) reverses a list:

List.rev [x0;x1;…;xn-1] = [xn-1;…;x1;x0]

These functions are predefined in F#.

(The @ operator is actually the infix operator corresponding to the library function List.append).

Let rec (@) x sys =

Match xs with

| [] -> ys

| x::xs’ -> x::(xs’ @ ys);;

Val (@) : ‘a list -> ‘a list -> ‘ a list

Evaluation:

[1;2]@[3;4]

-> 1::([2]@[3;4])

-> 1::(2::([] @ [3;4]))

-> 1 :: (2:: [3;4]))

-> 1:: [2;3;4]

-> [1;2;3;4]

The evaluation of xs @ ys comprises m+1 pattern matches plus m con’es where m is the length of xs.

The operators :: and @ have the same precedence (5) and both associate to the right. A mixture of these operators also associates to the right, so [1] @ 2 :: [3] is interpreted as [1] @ (2::[3]), while 1 :: [2] @ [3] is interpreted as 1 :: ( [2] @ [3]):

[1] @ 2 :: [3];;

Val it : int list = [1;2;3]

1 :: [2] @ [3];;

Val it : int list = [1;2;3]

**4.5 The value restrictions on polymorphic expressions (81)**

The type system and type inference of F# is very general and flexible. It has, however, been necessary to make a *restriction* on the *use of polymorphic expressions* in order to ensure type correctness in all situations.

The formulation of this restriction is based on the concept of *value expressions*. A value expression is an expression that is not reduced further by an evaluation, that is, it has already the same form as its value. The following expressions are hence value expressions:

[] Some[] (5,[]) (fun x -> [x])

While

List.rev[] [] @ []

Do not qualify as a value expression as they can be further evaluated.

Note that a function expression (a closure) is considered a value expression because it is only evaluated further when applied to an argument.

The restriction applies to the expression exp in declarations:

Let id = exp

And states the following:

At top level, polymorphic expressions are allowed only if they are value expressions. Polymorphic expressions can be used freely for intermediate results.

The restriction on polymorphic expressions may be paraphrased as follows:

1. All monomorphic expressions are OK, even non-value expressions.
2. All value expressions are OK, even polymorphic one.
3. At top-level, polymorphic non-value expressions are forbidden.

Where the type of a *monomorphic expression* does not contain type variables, that is, it is a *monomorphic type.*

**4.6 Examples. A model-based approach (82)**

**Example: Cash register (82)**

**Example: Map colouring (85)**

1. **Collections: Lists, maps and sets (s. 93-120)**

Lists, maps and sets are all part of the collection library in F#.

**5.1 Lists (93)**

List.map, List.exists, List.forall, List.tryFind, List.filter, List.fold, List.foldBack and List.collect are all operations that exists (see table 5.1 page 94).

**The map function (94)**

The library function List.map:

List.map: (‘a -> ‘b) -> ‘a sist -> ‘b list

Works as follows:

List.map f [x0;x1;…;xn-1] = [fx0; fx1;…;fxn-1].

The function application List.map f is the function that applies the function f to each element x0,x1,…,xn-1 in a list [x0;x1;…;xn-1].

**Functions using a predicate on the list elements (95)**

The f# library contains a large number of functions using a predicate of type ‘a -> bool on elements in a list of type ‘a list.

Some of them are:

List.exists : (‘a -> bool) -> ‘a list -> bool

List.forall : (‘a -> bool) -> ‘a list -> bool

List.tryFind: (‘a -> bool) -> ‘a list -> ‘a option

List.filter : (‘a -> bool) -> ‘a list -> ‘a list

The value of the expression:

List.exists p [x0;x1;….;xn-1]

is true if p(xk) = true holds for some list element xk, and false otherwise.

The value of the expression:

List.forall p [x0;x1;…;xn-1]

Is true, if p(xk) = true holds for all list elements xk, and false otherwise.

The value of the exression:

List.tryFind p [x0;x1;…;xn-1]

Is some xk for a list element xk with p(xk) = true, or none if no such element exists.

The value of the expression:

List.filter p [x0;x1;…;xn-1]

Is the list of those list elements xk where p(xk)=true.

Note that the evaluation of the expression List.exists p [x0,x1,….xi-1,xi….,xn-1]

Does not terminate if the evaluation of the expression p(xk) does not terminate for some k, where 0 <= k <=n-1 and if p(xj) = false for all j where 1 <=j<=k.

**The functions fold and foldback (97)**

The evaluation of List.fold f e [x0;x1;…;xn-1] accumulates the list elements x0,x1,…,xn-1 using the accumulation function f and the start value e.

The type of List.fold is:

List.fold: (‘a -> ‘b -> ‘a) -> ‘a -> ‘b list -> ‘a

The List.fold function basically takes a function f and a start value e and a list. It then applies the function f to every element in the list starting from e.

Example:

List vs = [v0;….;vn-1]

vi is a pair (xi,yi) of floats for 0 <= i < n.

We want to compute the sum of the norms of the vectors in vs using the norm function:

Let norm(x:float,y:float) = sqrt(x\*x+y\*y);;

Val norm : float \* float -> float

This is a case for applying List.fold with:

List element type: float \* float

Accumulator type: float

Accumulator function: fun s (x,y) -> s + norm(x,y)

Start value: 0.0

Declaration:

Let sumOfNorms vs =

List.fold (fun (x,y) -> s + norm(x,y)) 0.0 vs;;

Val sumOfNorms : (float \* float) list -> float

Let vs = [(1.0,2.0);(2.0,1.0);(2.0,5.5)];;

Val vs = (float \* float) list = [(1.0,2.0);(2.0,1.0);(2.0,5.5)]

sumOfNorms vs;;

val it : float = 10.32448591

Applying List.fold to the following version of “cons”:

Fun rs x -> x::rs

Where the parameters are interchanged, gives a declaration of the reverse function for lists:

Let rev cs = List.fold (fun rs x -> x::rs) [] xs;;

Val rev : ’a list -> ’a list

Rev [1;2;3];;

Val it : int list = [3;2;1]

The function List.foldBack is similar to List.fold but the elements are accumulated in the opposite order. The type of List.foldBack is:

List.foldBack: (‘a -> ‘b -> ‘b) -> ‘a list -> ‘b -> ‘b

And the general formula is:

List.foldBack g [x0;x1;…;xn-1] e = g x0 ( g x1 (…(g xn-1 e) …))

The evaluation of List.foldBack g [x0;x1;…;xn-1] e accumulates the list elements in reverse order xn-1,…,x1,x0 using the accumulation function g and the start value e.

Using List.foldBack on the “sum of norms” function:

Element and accumulater types can be used unchanged, but the parameters in the accumulator function must be interchanged:

Let backSumOfNorms vs =

List.foldBack (fun (x,y) s -> s + norm(x,y)) vs 0.0;;

Val backSumOfNorms : (float \* float) list -> float

This function works like the previous sumOfNorms function, but the norms are added in the opposite order, so starting with the norm of the last vector in the list.

Applying List.foldBack on the “cons” operator:

Fun x xs -> x::xs

Gives the append function:

Let app ys zs = List.foldBack (fun x xs -> x::xs) ys zs;;

Val app : ‘a list -> ‘a list -> ‘a list

App [1;2;3] [4;5;6];;

Val it : int list = [1;2;3;4;5;6]

The prefix version of an infix operator can be used as argument in fold and foldback:

List.fold (+) 0 [1;2;3];;

Val it : int = 6 //computed: ((0+1)+2)+3)

List.foldBack (+) [1;2;3] 0;;

Val it : int = 6 //computed: (1+(2+(3+0))

The results are equal, because + is a *commutative operator*: a+b=b+a

If using a non-commutative operator the results will be different:

List.fold (-) 0 [1;2;3];;

Val it : int = -6 //computed: ((0-1)-2)-3)

List.foldBack (-) [1;2;3] 0;;

Val it : int = 2 //computed: (1-(2-(3-0))

The expressions use the functions:

Fun e x -> e – x

Fun x e - > x – e

Note that a function declared by means of fold and foldback will always scan the whole list. Thus, the following declaration for the exists function:

Let existsF p =

List.fold (fun b -> (fun x -> p x || b)) false;;

Val existsF : (’a -> bool) -> (’a list -> bool)

Will not behave like the function List.exists with regard to non-termination.

It is not good to use fold or foldback to declare functions like exists or find, because those functions terminate after finding their value, and thus don’t scan the whole list.

**Declarations of fold and foldback (102)**

Let rec fold f e = function

| x::xs -> fold f (f e x) xs

| [] -> e;;

Let rec foldBack g xlst e =

Match xlst with

| x::xs -> g x (foldBack g xs e)

| [] -> e;;

See evaluations on page 103.

The evaluation of fold is much more efficient than the evaluation of foldback.

**5.2 Finite sets (104)**

Finite sets are of form {a1,a2,…,an} with elements a1,…,an from some set A.

A set provides a useful abstraction in cases where we have an unordered collection of elements where repetitions among the elements are of no concern.

**The mathematical set concept (104)**

A *set* (in math) is a collection of elements like:

{Bob,Bill,Ben} or {1,3,5,7,9}

Where it is possible to decide whether a given value is in the set.

The empty set containing no elements is written {} or Ø

The order in which elements are enumerated and repetitions among the elements both are of no concern, the following expressions denote the same set:

{Bob,Bill,Ben} {Bob,Bill,Ben,Bill} {Bill,Ben,Bill,Bob}

The above examples are all finite sets.

Sets may be infinite, like a set of all natural numbers, or a set of all real numbers.

A set A is a *subset* of B if all the elements of A also are elements in B.

Two sets are equal if they are both subsets of each other, which means they are equal if the contain exactly the same elements.

Some of the standard operations on sets are: *Union, Intersection and difference*.

Union is the set of elements that are in at least one of the sets:

{Bob,Bill,Ben} union {Alice, Bill, Ann} = {Alice, Ann, Bob, Bill, Ben}

Intersection is the set of elements that are in both A and B:

{Bob,Bill,Ben} intersection {Alice,Bill,Ann} = {Bill}

Difference is the subset of the elements from A that are not in B:  
{Bob,Bill,Ben} difference {Alice, Bill, Ann} = {Bob,Ben}

**Sets in F# (105)**

The Set library in f# supports finites sets of elements of a type where ordering is defined.

Example:

Set [“Bob”; “Bill”; “Ben”];;

Val it : Set<string> = set [“Ben”; “Bill”; “Bob”]

Set = the set builder function. (Note that the answer from F# creates the set in lexicographical ordering.)

Set [3;1;9;5;7;9;1];;

Val it : set<int> = set [1;3;5;7;9]

The equality “=” operator can be used on sets:  
set [“Bob”; “Bill”; “Ben”] = set [“Bill”; “Ben”; “Bill”;”Bob”];;

Val it : bool = true

**Basic properties and operations on sets (106)**

The functions Set.ofList and Set.toList are conversion functions between lists and sets. The resulting list is ordered and contains no repeated elements.

An element can be inserted in a set with the function Set.add, and removed with the Set.remove.

The add and remove operations do not change the original set, that is, they have no side effect.

Also Set.contains, Set.isSubset, Set.minElement, Set.maxElement are all operations that can be done on sets (page 107).

Set.count gives the amount of elements in a set:

Set.count ( set [“Bob”; “Bill”; “Ben”]);;

Val it : int = 3

**Fundamental operations on sets (107)**

We illustrate set operations for union, intersection and difference using an example where males are supposed to be all the males at a golf club, and boardMembers are the members of the board for that club.

Let boardMembers = Set.ofList [ “Alice”; “Bill”; “Ann”];;

Val boardMembers : Set<string> = set [“Alice”;”Ann”;”Bill”]

Set.union males boardMembers;;

Val it : Set<string> = set [“Alice”;”Ann”;”Ben”;”Bill”;”Bob”]

Set.intersect males boardMembers;;

Val it : Set<string> = set [“Bill”]

Set.difference males boardMembers;;

Val it : Set<string> = set[“Ben”;”Bob”]

A function can be applied to every member of a set using Set.map in the same manner it can be applied to every element of a list using List.map.

Let setOfCounts s = Set.map Set.count s;;

Val setOfCounts : Set<Set<’a>> -> Set<int> when ‘a : comparison

Consider the value for the set of sets {{1,3,5},{2,4},{7,8,9}}:

Let ss = set [set[1;3;5];set[2;4];set[7;8;9]];;

Val it : Set<Set<int>> = set [set[1;3;5];set[2;4];set[7;8;9]]

setOfCounts ss;;

val it : Set<int> = set [2;3]

Set.exists, Set.forall and Set.filter works in a similar manner to the functions on List.

Set.fold and Set.foldBack also works the same on sets as on lists.

Notice that it is more natural to base a declaration of setOfCounts on Set.map rather than basing it on one of the fold functions.

**Recursive functions on sets (109)**

The functions Set.map, Set.filter, Set.fold and Set.foldBack will traverse the complete set before they terminate, unless the evaluation is aborted by raising an exception, and this may undesirable in some situations.

Consider the function that finds the least element in a set satisfying a given predicate:

tryFind: (‘a – bool) -> Set<’a> -> ‘a option

when ‘a : comparison

This function can be declared by repeated extraction of the minimal element from a set until an element satisfying the predicate is found:

Let rec tryFind p s =

If Set.isEmpty s then None

Else let minE = Set.minElement s

If p minE then Some minE

Else tryFind p (Set.remove minE s);;

For example, the least three-element set from a set of sets is extracted as follows:

Let ss = set [set[1;3;5]; set [2;4]; set [7;8;9]];;

tryFind (fun s -> Set.count s = 3) ss;;

val it : Set<int> option = Some (set[1;3;5])

**Example: Map colouring (110)**

**5.3 Maps (113)**

In the modelling and solution for many problems it is often convenient to use finite functions to uniquely associate *values* with *keys*. Such finite functions from keys to values are called *maps.*

**The mathematical concept of a map (113)**

A *map* from a set A to a set B is a *finite* subset A’ of A together with a function m defined on A’:

m : A’ -> B

The set A’ is called the *domain* of m and we write: dom m = A’

An element ai in the set A’ is called a *key* for the map m. A pair (ai,bi) is called an *entry*, and bi is called the *value* for the key ai.

The order of the entries is of no significance, as the map only expresses an association of values to keys.

Any two keys ai and aj in different entries are different, as there is only one value for each key.

A map may be represented as a finite set of its entries:  
entriesOf(m) = {(a0,b0),…,(an-1,bn-1)}

**Maps in f# (114)**

The map library supports maps of polymorphic types Map<’a,’b>, where ‘a and ‘b are the types of the keys and values, respectively, of the map.

The map functions exists, forall, map, fold and foldback are similar to their list and set siblings.

**Example: Cash register (116)**