02157 Functional Programming

Lecture 1: Introduction and Getting Started

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

WELCOME to
02157 Functional Programming

Teacher: Michael R. Hansen, DTU Compute mire@dtu.dk

Teaching assistants:

Jonas Dahl Larsen
Mathias Spezia
Mikael Hjermitslev Hoffmann
Oliwia Pindel
Shuokai Ma

Homepage: www.compute.dtu.dk/courses/02157

About functions

Advanced Engineering Mathematics 1

- eNotes: `https://01006.compute.dtu.dk/enoter`

About functions

Advanced Engineering Mathematics 1

- eNotes: `https://01006.compute.dtu.dk/enoter`

For a function, like

$$f(x) = x^2$$

we often mention its domain an range:

$$f : \mathbb{R} \to \mathbb{R}$$

About functions

Advanced Engineering Mathematics 1

- eNotes: `https://01006.compute.dtu.dk/enoter`

For a function, like

$$f(x) = x^2$$

we often mention its domain an range:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

For a typed functional language like F#, a function like:

```
let f x = x ** 2.0;;
```

has an associated type:

```
f:float -> float
```

where float is the type of both the domain and the range.

A Simple Functional Programming Setting

A program *f* is a function

$$f : \mathit{Argument} \rightarrow \mathit{Result}$$

that takes one argument and produces one result.

A Simple Functional Programming Setting

A program *f* is a function

$$f : Argument \rightarrow Result$$

that takes one argument and produces one result.

Consider

```
let f x = 2*x + 3;;
```

A Simple Functional Programming Setting

A program *f* is a function

$$f : \mathit{Argument} \rightarrow \mathit{Result}$$

that takes one argument and produces one result.

Consider

```
let f x = 2*x + 3;;
```

Every function has a type specifying types of argument and result:

```
f: int -> int
```

- argument and result of `f` have type `int` (for integers).

A Simple Functional Programming Setting

A program *f* is a function

$$f : Argument \rightarrow Result$$

that takes one argument and produces one result.

Consider

```
let f x = 2*x + 3;;
```

Every function has a type specifying types of argument and result:

```
f: int -> int
```

- argument and result of `f` have type `int` (for integers).

Computation is governed by function application

$$\begin{array}{lll} & \texttt{f}(1+2) & \\ = & \texttt{f}(3) & \text{evaluate argument} \\ = & 2*3+3 & \text{substitute 3 in for } x \text{ in } f\text{'s body} \\ = & 9 & \end{array}$$

A Simple Functional Programming Setting

A program *f* is a function

$$f : Argument \rightarrow Result$$

that takes one argument and produces one result.

Consider

```
let f x = 2*x + 3;;
```

Every function has a type specifying types of argument and result:

```
f: int -> int
```

- argument and result of `f` have type `int` (for integers).

Computation is governed by function application

	f(1 + 2)	
=	f(3)	evaluate argument
=	2 * 3 + 3	substitute 3 in for *x* in *f*'s body
=	9	

F# has eager evaluation: Compute argument before making the call

Context of 02157

Prerequisites

- You have used an editor to create programs
- You have installed a program on your laptop
- You have had (or have in the same semester) a course on Discrete Mathematics

Context of 02157

Prerequisites

- You have used an editor to create programs
- You have installed a program on your laptop
- You have had (or have in the same semester) a course on Discrete Mathematics

The course is a part of educations leading to the MSc programme in Computer Science and Engineering.

- candidates contributing to the development of high-quality, advanced software products

Context of 02157

Prerequisites

- You have used an editor to create programs
- You have installed a program on your laptop
- You have had (or have in the same semester) a course on Discrete Mathematics

The course is a part of educations leading to the MSc programme in Computer Science and Engineering.

- candidates contributing to the development of high-quality, advanced software products

It is an aim to contribute to the fundament for educations leading to the MSc education in CS&E

Context of 02157

Prerequisites

- You have used an editor to create programs
- You have installed a program on your laptop
- You have had (or have in the same semester) a course on Discrete Mathematics

The course is a part of educations leading to the MSc programme in Computer Science and Engineering.

- candidates contributing to the development of high-quality, advanced software products

It is an aim to contribute to the fundament for educations leading to the MSc education in CS&E

May sound good; but what does it mean?

There is no magic

It is possible to understand everything:

There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language

There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language
- The semantics (meaning) of programs

There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language
- The semantics (meaning) of programs
- The evaluation of programs

There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language
- The semantics (meaning) of programs
- The evaluation of programs
- The properties of programs

There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language
- The semantics (meaning) of programs
- The evaluation of programs
- The properties of programs

Functional programming is a simple setting supporting

- declaration of clear, concise programs at a high level of abstraction
- understanding and analysis of programs

due to the basis on mathematical functions (no side-effects)

An archetypical example $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition: recursion formula

$$
\begin{aligned}
0! &= 1 && (i) \\
n! &= n \cdot (n-1)!, \quad \text{for } n > 0 && (ii)
\end{aligned}
$$

- $n!$ is defined recursively in terms of $(n-1)!$ when $n > 0$

An archetypical example $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition: recursion formula

$$\begin{array}{rcll} 0! & = & 1 & (i) \\ n! & = & n \cdot (n-1)!, \quad \text{for } n > 0 & (ii) \end{array}$$

- $n!$ is defined recursively in terms of $(n-1)!$ when $n > 0$

Computation:

$$\begin{array}{cll} & 3! & \\ = & 3 \cdot (3-1)! & (ii) \\ = & 3 \cdot 2 \cdot (2-1)! & (ii) \\ = & 3 \cdot 2 \cdot 1 \cdot (1-1)! & (ii) \\ = & 3 \cdot 2 \cdot 1 \cdot 1 & (i) \\ = & 6 & \end{array}$$

Declaring recursive functions: `let` *rec f x* $=$ *e*

- the function *f* occurs in the body *e* of a *recursive declaration*

Declaring recursive functions: `let` *rec f x* = *e*

- the function *f* occurs in the body *e* of a *recursive declaration*

A recursive function declaration:

```
let rec fact n =
    if n=0 then 1                        (*  i *)
    else n * fact(n-1);;                 (* ii *)
val fact : int -> int
```

Declaring recursive functions: `let` *rec f x* = *e*

- the function *f* occurs in the body *e* of a *recursive declaration*

A recursive function declaration:

```
let rec fact n =
    if n=0 then 1                    (*  i *)
    else n * fact(n-1);;             (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lll}
& \texttt{fact}(3) & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) \quad [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) \\
\rightsquigarrow & 6 &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

Declaring recursive functions: `let` *rec f x* = *e*

- the function *f* occurs in the body *e* of a *recursive declaration*

A recursive function declaration:

```
let rec fact n =
    if n=0 then 1                    (*  i *)
    else n * fact(n-1);;             (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{llll}
 & \texttt{fact}(3) & & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) & [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) & [\texttt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) & \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & \\
\rightsquigarrow & 6 & &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

- An environment is used to bind the formal parameter *n* to actual parameters $3, 2, 1, 0$ during evaluation

Declaring recursive functions: `let` *rec f x* = *e*

- the function *f* occurs in the body *e* of a *recursive declaration*

A recursive function declaration:

```
let rec fact n =
    if n=0 then 1                   (*  i *)
    else n * fact(n-1);;            (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{llll}
 & \texttt{fact}(3) & & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) & [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) & [\texttt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) & [\texttt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & \\
\rightsquigarrow & 6 & &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

- An environment is used to bind the formal parameter *n* to actual parameters $3, 2, 1, 0$ during evaluation

Declaring recursive functions: `let` *rec f x* = *e*

- the function *f* occurs in the body *e* of a *recursive declaration*

A recursive function declaration:

```
let rec fact n =
    if n=0 then 1                    (*  i *)
    else n * fact(n-1);;             (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lllr}
 & \texttt{fact}(3) & & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) & [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) & [\texttt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) & [\texttt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [\texttt{n} \mapsto 0] \\
\rightsquigarrow & 6 & &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

- An environment is used to bind the formal parameter *n* to actual parameters $3, 2, 1, 0$ during evaluation

Some functional programming background

- The λ-calculus was introduced around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.

Some functional programming background

- The λ-calculus was introduced around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.

- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.

Some functional programming background

- The λ-calculus was introduced around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) were introduced in the 1970s.

Some functional programming background

- The λ-calculus was introduced around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) were introduced in the 1970s.
- Functional languages (SML, Haskell, OCAML, F#, . . .) have now applications far away from their origin: Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .

Some functional programming background

- The λ-calculus was introduced around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) were introduced in the 1970s.
- Functional languages (SML, Haskell, OCAML, F#, ...) have now applications far away from their origin: Compilers, Artificial Intelligence, Web-applications, Financial sector, ...
- Declarative aspects are now sneaking into ”main stream languages”

Some functional programming background

- The λ-calculus was introduced around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) were introduced in the 1970s.
- Functional languages (SML, Haskell, OCAML, F#, ...) have now applications far away from their origin: Compilers, Artificial Intelligence, Web-applications, Financial sector, ...
- Declarative aspects are now sneaking into ”main stream languages”
- Functional programming should be a mandatory element of every BSc. education in Computer Science according to ACM’s and IEEE’s curricula recommendations, 2013.

Lambda Calculus

The untyped Lambda Calculus has just three kinds of expressions e:

- variables x
- abstractions $\lambda x.e$
- applications $e_1\ e_2$

Lambda Calculus

The untyped Lambda Calculus has just three kinds of expressions e:

- variables x
- abstractions $\lambda x.e$
- applications $e_1\ e_2$

where

- $\lambda x.e$ reads: “the function of x given by e”

Lambda Calculus

The untyped Lambda Calculus has just three kinds of expressions e:

- variables x
- abstractions $\lambda x.e$
- applications $e_1\ e_2$

where

- $\lambda x.e$ reads: “the function of x given by e”

An application like $(\lambda x.e)\ e_2$ may be evaluated as follows:

$$(\lambda x.e)\ e_2 \quad \rightsquigarrow \quad e_2'$$

where e_2' is obtained from e_2 by

- substituting every free occurrence of x in e by e_2

like we did in the previous examples

Lambda Calculus

The untyped Lambda Calculus has just three kinds of expressions e:

- variables x
- abstractions $\lambda x.e$
- applications $e_1\ e_2$

where

- $\lambda x.e$ reads: "the function of x given by e"

An application like $(\lambda x.e)\ e_2$ may be evaluated as follows:

$$(\lambda x.e)\ e_2 \quad \rightsquigarrow \quad e_2'$$

where e_2' is obtained from e_2 by

- substituting every free occurrence of x in e by e_2

like we did in the previous examples

No magic: A full explanation can be given in terms of few concepts

Lambda Calculus

The untyped Lambda Calculus has just three kinds of expressions e:

- variables x
- abstractions $\lambda x.e$
- applications $e_1\ e_2$

where

- $\lambda x.e$ reads: “the function of x given by e”

An application like $(\lambda x.e)\ e_2$ may be evaluated as follows:

$$(\lambda x.e)\ e_2 \quad \rightsquigarrow \quad e_2'$$

where e_2' is obtained from e_2 by

- substituting every free occurrence of x in e by e_2

like we did in the previous examples

No magic: A full explanation can be given in terms of few concepts

The part of F# we will use is based on typed lambda calculus

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns:
  x _ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns:
  x _ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions:
  x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $e_1 \, e_2$ $e_1 \oplus e_2$ `let` $p_1 = e_1$ `in` e_2 $e : t$

  `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses*   ...

where the construct *clauses* has the form:

$| \; p_1 \; \text{->} \; e_1 \; | \; \ldots \; | \; p_n \; \text{->} \; e_n$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns:
  $x \quad \_ \quad (p_1, \ldots, p_n) \quad p_1 :: p_2 \quad p_1 | p_2 \quad p \;\texttt{when}\; e \quad p \;\texttt{as}\; x \quad p : t \ldots$
- Expressions:
  $x \quad (e_1, \ldots, e_n) \quad e_1 :: e_2 \quad e_1\, e_2 \quad e_1 \oplus e_2 \quad \texttt{let}\; p_1 = e_1 \;\texttt{in}\; e_2 \quad e : t$

  $\texttt{if}\; e \;\texttt{then}\; e_1 \;\texttt{then}\; e_2 \qquad \texttt{match}\; e \;\texttt{with}\; clauses$

  $\texttt{fun}\; p_1 \cdots p_n \texttt{->} e \qquad \texttt{function}\; clauses \qquad \ldots$
- Declarations $\texttt{let}\; f\; p_1 \ldots p_n = e \quad \texttt{let rec}\; f\; p_1 \ldots p_n = e$, $n \geq 0$

where the construct *clauses* has the form:

$\texttt{| p}_1 \texttt{ -> e}_1 \texttt{ | ... | p}_n \texttt{ -> e}_n$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns:
  x _ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions:
  x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $e_1 \, e_2$ $e_1 \oplus e_2$ `let` $p_1 = e_1$ `in` e_2 $e : t$

  `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses*   ...
- Declarations `let` $f \; p_1 \ldots p_n = e$ `let rec` $f \; p_1 \ldots p_n = e$, $n \geq 0$
- Types
  `int`   `float`   `bool`   `string` $'a \ldots$
  $t_1 * t_2 * \cdots * t_n$ t `list` t_1`->`$t_2 \ldots$

where the construct *clauses* has the form:

| p_1 `->` e_1 | ... | p_n `->` e_n

Overview: Syntactical constructs in "our part of" F#

- Constants: 0, 1.1, true, ...
- Patterns:
  x _ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions:
  x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $e_1 e_2$ $e_1 \oplus e_2$ `let` $p_1 = e_1$ `in` e_2 $e : t$

  `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses*   ...
- Declarations `let` $f\ p_1 \ldots p_n = e$ `let rec` $f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  `int` `float` `bool` `string` $'a \ldots$
  $t_1 * t_2 * \cdots * t_n$ t `list` t_1`->`$t_2 \ldots$

where the construct *clauses* has the form:

| p_1 -> e_1 | ... | p_n -> e_n

In addition to that

- type declarations, precedence and associativity rules, parenthesis around p and e and type correctness

Have a look at

- `http://homepages.inf.ed.ac.uk/wadler/realworld/`
- `https://fsharp.org/testimonials/`

concerning use of functional programming in the ”real world”.

Practical Matters

- General information:
  http://courses.compute.dtu.dk/02157

Practical Matters

- General information:
  `http://courses.compute.dtu.dk/02157`

- Practical Information:
  `http://courses.compute.dtu.dk/02157/PracticalInfo.html`

  Exam form: Written exam, 4 hour – no aid allowed

Practical Matters

- General information:
  http://courses.compute.dtu.dk/02157
- Practical Information:
  http://courses.compute.dtu.dk/02157/PracticalInfo.html

  Exam form: Written exam, 4 hour – no aid allowed
- Course plan:
  http://courses.compute.dtu.dk/02157/plan.html

Practical Matters

- General information:
  http://courses.compute.dtu.dk/02157

- Practical Information:
  http://courses.compute.dtu.dk/02157/PracticalInfo.html

  Exam form: Written exam, 4 hour – no aid allowed

- Course plan:
  http://courses.compute.dtu.dk/02157/plan.html

On DTU Learn you can find some material

- A brief course introduction
- A mini-project on polynomials
- Slides
- ....

Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and "steady progress" throughout the semester

Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and “steady progress” throughout the semester

Mini-projects: Exercise FP concepts and techniques while

- telling a coherent story on a specific topic
- relating FP to neighbouring courses
- introducing fundamental CS concepts

Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and "steady progress" throughout the semester

Mini-projects: Exercise FP concepts and techniques while

- telling a coherent story on a specific topic
- relating FP to neighbouring courses
- introducing fundamental CS concepts

Nothing is mandatory

Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and "steady progress" throughout the semester

Mini-projects: Exercise FP concepts and techniques while

- telling a coherent story on a specific topic
- relating FP to neighbouring courses
- introducing fundamental CS concepts

Nothing is mandatory

It is your own responsibility to achieve a good use of Fridays' teaching slot

- no online support
- no hotline support

Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and "steady progress" throughout the semester

Mini-projects: Exercise FP concepts and techniques while

- telling a coherent story on a specific topic
- relating FP to neighbouring courses
- introducing fundamental CS concepts

Nothing is mandatory

It is your own responsibility to achieve a good use of Fridays' teaching slot

- no online support
- no hotline support

You are always welcome to visit my office: Room 112, Building 322

Overview

Part 1 Getting Started:

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Main ingredients of F#

Overview

Part 1 Getting Started:

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Main ingredients of F#

Part 2 Lists:

- Lists: values and constructors
- Recursions following the structure of lists
- Polymorphism

A value-oriented approach

The Interactive Environment

```
2*3 + 4;;
val it : int = 10
```

The Interactive Environment

```
2*3 + 4;;
val it : int = 10
```

⇐ Input to the F# system

⇐ Answer from the F# system

The Interactive Environment

```
2*3 + 4;;              ⇐ Input to the F# system
val it : int = 10      ⇐ Answer from the F# system
```

- The *keyword* `val` indicates a value is computed
- The *integer* 10 is the computed value
- `int` is the *type* of the computed value
- The *identifier* `it` names the (last) computed value

The Interactive Environment

```
2*3 + 4;;                    ⇐ Input to the F# system
val it : int = 10            ⇐ Answer from the F# system
```

- The *keyword* `val` indicates a value is computed
- The *integer* 10 is the computed value
- `int` is the *type* of the computed value
- The *identifier* `it` names the (last) computed value

The notion *binding* explains which entities are named by identifiers.

`it` $\mapsto$ 10 reads: "`it` is bound to 10"

Value Declarations

A value declaration has the form: `let` *identifier* = *expression*

```
let price = 25 * 5;;
```
⇐ A declaration as input

```
val price : int = 125
```
⇐ Answer from the F# system

The effect of a declaration is a binding: `price` $\mapsto$ 125

Value Declarations

A value declaration has the form: `let` *identifier* = *expression*

```
let price = 25 * 5;;            ⇐ A declaration as input
val price : int = 125           ⇐ Answer from the F# system
```

The effect of a declaration is a binding: `price` ↦ 125

Bound identifiers can be used in expressions and declarations, e.g.

```
let newPrice = 2*price;;
val newPrice : int = 250

newPrice > 500;;
val it : bool = false
```

Value Declarations

A value declaration has the form: `let` *identifier* $=$ *expression*

```
let price = 25 * 5;;
val price : int = 125
```

⇐ A declaration as input

⇐ Answer from the F# system

The effect of a declaration is a binding: `price` $\mapsto$ 125

Bound identifiers can be used in expressions and declarations, e.g.

```
let newPrice = 2*price;;
val newPrice : int = 250

newPrice > 500;;
val it : bool = false
```

A collection of bindings

$$\begin{bmatrix} \texttt{price} & \mapsto & 125 \\ \texttt{newPrice} & \mapsto & 250 \\ \texttt{it} & \mapsto & \text{false} \end{bmatrix}$$

is called an environment

Function Declarations 1: `let` $f\,x = e$

- x is called the *formal parameter*
- the defining expression e is called the *body* of the declaration

Function Declarations 1: `let` $f\,x = e$

- x is called the *formal parameter*
- the defining expression e is called the *body* of the declaration

Declaration of the circle area function:

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

- `System.Math` is a program library
- `PI` is an identifier (with type `float`) for π in `System.Math`

The type is automatically inferred in the answer:

```
val circleArea : float -> float
```

Function Declarations 1: `let` $f\,x = e$

- x is called the *formal parameter*
- the defining expression e is called the *body* of the declaration

Declaration of the circle area function:

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

- `System.Math` is a program library
- `PI` is an identifier (with type `float`) for π in `System.Math`

The type is automatically inferred in the answer:

```
val circleArea : float -> float
```

Applications of the function:

```
circleArea 1.0;; (* this is a comment *)
val it : float = 3.141592654

circleArea(3.2);; // A comment: optional brackets
val it : float = 32.16990877
```

Function Declarations 1: `let` $f\,x = e$

- x is called the *formal parameter*
- the defining expression e is called the *body* of the declaration

Declaration of the circle area function:

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

- `System.Math` is a program library
- `PI` is an identifier (with type `float`) for π in `System.Math`

The type is automatically inferred in the answer:

```
val circleArea : float -> float
```

Applications of the function:

```
circleArea 1.0;; (* this is a comment *)
val it : float = 3.141592654

circleArea(3.2);; // A comment: optional brackets
val it : float = 32.16990877
```

`1.0` and `3.2` are also called *actual parameters*

Patterns

A pattern is composed from identifiers, constants and the wildcard pattern: _ using constructors (considered soon)

Examples of patterns are: `3.1`, `true`, `n`, `x`, `5`, `_`

Patterns

A pattern is composed from identifiers, constants and the wildcard pattern: _ using constructors (considered soon)

Examples of patterns are: `3.1`, `true`, `n`, `x`, `5`, `_`

- A pattern may match a value, and if so it results in an environment with bindings for every identifier in the pattern.
- The wildcard pattern _ matches any value (resulting in no binding)

Patterns

A pattern is composed from identifiers, constants and the wildcard pattern: _ using constructors (considered soon)

Examples of patterns are: `3.1, true, n, x, 5, _`

- A pattern may match a value, and if so it results in an environment with bindings for every identifier in the pattern.
- The wildcard pattern _ matches any value (resulting in no binding)

Examples:

- Value `3.1` matches pattern `x` resulting in environment: $[x \mapsto 3.1]$

Patterns

A pattern is composed from identifiers, constants and the wildcard pattern: _ using constructors (considered soon)

Examples of patterns are: `3.1, true, n, x, 5, _`

- A pattern may match a value, and if so it results in an environment with bindings for every identifier in the pattern.
- The wildcard pattern _ matches any value (resulting in no binding)

Examples:

- Value `3.1` matches pattern `x` resulting in environment: $[x \mapsto 3.1]$
- Value `true` matches pattern `true` resulting in environment $[\,]$

Patterns

A pattern is composed from identifiers, constants and the wildcard pattern: _ using constructors (considered soon)

Examples of patterns are: `3.1`, `true`, `n`, `x`, `5`, `_`

- A pattern may match a value, and if so it results in an environment with bindings for every identifier in the pattern.
- The wildcard pattern _ matches any value (resulting in no binding)

Examples:

- Value `3.1` matches pattern `x` resulting in environment: $[x \mapsto 3.1]$
- Value `true` matches pattern `true` resulting in environment $[\,]$
- The pair `(1,true)` matches pattern `(x,y)` resulting in environment $[x \mapsto 1, y \mapsto \mathit{true}]$

Match expressions

A match expression e_m has the following form:

$$\texttt{match}\ e\ \texttt{with}$$
$$\mid pat_1 \rightarrow e_1$$
$$\vdots$$
$$\mid pat_n \rightarrow e_n$$

Match expressions

A match expression e_m has the following form:

$$\begin{array}{l}\texttt{match}\ e\ \texttt{with} \\ \mid pat_1 \rightarrow e_1 \\ \quad\vdots \\ \mid pat_n \rightarrow e_n\end{array}$$

A match expression e_m is evaluated as follows:

1. evaluate e to a value, say v
2. search for the first pattern pat_i matching v
3. evaluate e_i in an environment enriched with the bindings from the pattern matching

Match expressions

A match expression e_m has the following form:

$$\texttt{match}\ e\ \texttt{with}\\ \mid pat_1 \rightarrow e_1\\ \vdots\\ \mid pat_n \rightarrow e_n$$

A match expression e_m is evaluated as follows:

1. evaluate e to a value, say v
2. search for the first pattern pat_i matching v
3. evaluate e_i in an environment enriched with the bindings from the pattern matching

If no pattern matches v, then the evaluation terminates abnormally.

Example: Match on a pair

Let e_1 be given by:

```
match (3+5, 3<5) with
| (0, _)       -> 0
| (n,false)    -> -n
| (n,_)        -> 2*n
```

Example: Match on a pair

Let e_1 be given by:

```
match (3+5, 3<5) with
| (0, _)       -> 0
| (n,false)    -> -n
| (n,_)        -> 2*n
```

Evaluation:

$$
\begin{array}{ll}
 & e_1 \\
\rightsquigarrow & (2 * \mathtt{n}, [\mathtt{n} \mapsto 8]) \\
\rightsquigarrow & (2 * 8, [\mathtt{n} \mapsto 8]) \\
\rightsquigarrow & 16
\end{array}
$$

Example: Match expression in a declaration

Function declaration:

```
let rec fact n =
   match n with
   | 0 -> 1                (*  i *)
   | n -> n * fact(n-1)    (* ii *)
val fact : int -> int
```

Example: Match expression in a declaration

Function declaration:

```
let rec fact n =
   match n with
   | 0 -> 1                  (*  i *)
   | n -> n * fact(n-1)      (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lll}
 & \mathtt{fact}(3) & \\
\rightsquigarrow & 3 * \mathtt{fact}(3-1) & (ii) \\
\rightsquigarrow & 3 * 2 * \mathtt{fact}(2-1) & (ii) \\
\rightsquigarrow & 3 * 2 * 1 * \mathtt{fact}(1-1) & (ii) \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) \\
\rightsquigarrow & 6 &
\end{array}
$$

Example: Match expression in a declaration

Function declaration:

```
let rec fact n =
   match n with
   | 0 -> 1                (*  i *)
   | n -> n * fact(n-1)    (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lll}
 & \texttt{fact}(3) & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) \quad [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) \\
\rightsquigarrow & 6 &
\end{array}
$$

Example: Match expression in a declaration

Function declaration:

```
let rec fact n =
   match n with
   | 0 -> 1              (*  i *)
   | n -> n * fact(n-1)  (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{llll}
 & \mathtt{fact}(3) & & \\
\rightsquigarrow & 3 * \mathtt{fact}(3-1) & (ii) & [\mathtt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \mathtt{fact}(2-1) & (ii) & [\mathtt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \mathtt{fact}(1-1) & (ii) & \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & \\
\rightsquigarrow & 6 & &
\end{array}
$$

Example: Match expression in a declaration

Function declaration:

```
let rec fact n =
   match n with
   | 0 -> 1                (*  i *)
   | n -> n * fact(n-1)    (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{llll}
 & \mathtt{fact}(3) & & \\
\rightsquigarrow & 3 * \mathtt{fact}(3-1) & (ii) & [\mathtt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \mathtt{fact}(2-1) & (ii) & [\mathtt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \mathtt{fact}(1-1) & (ii) & [\mathtt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & \\
\rightsquigarrow & 6 & &
\end{array}
$$

Example: Match expression in a declaration

Function declaration:

```
let rec fact n =
   match n with
   | 0 -> 1                (*  i *)
   | n -> n * fact(n-1)    (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{llll}
 & \texttt{fact}(3) & & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) & [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) & [\texttt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) & [\texttt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [\texttt{n} \mapsto 0] \\
\rightsquigarrow & 6 & &
\end{array}
$$

Example: Match expression in a declaration

Function declaration:

```
let rec fact n =
   match n with
   | 0 -> 1                 (*  i *)
   | n -> n * fact(n-1)     (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{llll}
 & \texttt{fact}(3) & & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) & [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) & [\texttt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) & [\texttt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [\texttt{n} \mapsto 0] \\
\rightsquigarrow & 6 & &
\end{array}
$$

A match with a when clause and an exception:

```
let rec fact n =
   match n with
   | 0          -> 1
   | n when n>0 -> n * fact(n-1)
   | _          -> failwith "Negative argument"
```

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$x^0 = 1 \quad (1)$$

$$x^n = x \cdot x^{n-1}, \quad \text{for } n > 0 \quad (2)$$

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$x^0 = 1 \qquad (1)$$
$$x^n = x \cdot x^{n-1}, \quad \text{for } n > 0 \qquad (2)$$

Function declaration:

```
let rec power(x,n) =
   match (x,n) with
   | (_,0) -> 1.0                      (* 1 *)
   | (x,n) -> x * power(x,n-1)          (* 2 *)
```

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$x^0 = 1 \qquad (1)$$
$$x^n = x \cdot x^{n-1}, \quad \text{for } n > 0 \qquad (2)$$

Function declaration:

```
let rec power(x,n) =
   match (x,n) with
   | (_,0) -> 1.0                   (* 1 *)
   | (x,n) -> x * power(x,n-1)      (* 2 *)
```

Patterns:

`(_,0)` matches any pair of the form $(u, 0)$.

`(x,n)` matches any pair (u, i) yielding the bindings

$$\texttt{x} \mapsto u, \texttt{n} \mapsto i$$

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$
\begin{aligned}
x^0 &= 1 && (1)\\
x^n &= x \cdot x^{n-1}, \quad \text{for } n > 0 && (2)
\end{aligned}
$$

Function declaration:

```
let rec power(x,n) =
   match (x,n) with
   | (_,0) -> 1.0                       (* 1 *)
   | (x,n) -> x * power(x,n-1)          (* 2 *)
```

Patterns:

`(_, 0)` matches any pair of the form $(u, 0)$.

`(x, n)` matches any pair (u, i) yielding the bindings

$$\texttt{x} \mapsto u, \texttt{n} \mapsto i$$

Can you simplify the program?

Evaluation. Example: `power(4.0, 2)`

Function declaration:

```
let rec power(x,n) =
  match (x,n) with
  | (_,0) -> 1.0                    (* 1 *)
  | (x,n) -> x * power(x,n-1)       (* 2 *)
```

Evaluation:

	$\mathtt{power}(4.0, 2)$	
$\rightsquigarrow$	$4.0 * \mathtt{power}(4.0, 2-1)$	Clause 2, $[\mathtt{x} \mapsto 4.0, \mathtt{n} \mapsto 2]$
$\rightsquigarrow$	$4.0 * \mathtt{power}(4.0, 1)$	
$\rightsquigarrow$	$4.0 * (4.0 * \mathtt{power}(4.0, 1-1))$	Clause 2, $[\mathtt{x} \mapsto 4.0, \mathtt{n} \mapsto 1]$
$\rightsquigarrow$	$4.0 * (4.0 * \mathtt{power}(4.0, 0))$	
$\rightsquigarrow$	$4.0 * (4.0 * 1)$	Clause 1
$\rightsquigarrow$	16.0	

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	~27, 0, 15, 21000
Floats	`float`	~27.3, 0.0, 48.21
Booleans	`bool`	true, false

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	~27, 0, 15, 21000
Floats	`float`	~27.3, 0.0, 48.21
Booleans	`bool`	true, false

Pairs:

If $e_1 : \tau_1$ and $e_2 : \tau_2$
then $(e_1, e_2) : \tau_1 * \tau_2$ pair (tuple) type constructor

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	~27, 0, 15, 21000
Floats	`float`	~27.3, 0.0, 48.21
Booleans	`bool`	true, false

Pairs:
If $e_1 : \tau_1$ and $e_2 : \tau_2$
then $(e_1, e_2) : \tau_1 * \tau_2$ pair (tuple) type constructor

Functions:
if $f : \tau_1 \texttt{->} \tau_2$ and $a : \tau_1$ function type constructor
then $f(a) : \tau_2$

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	~27, 0, 15, 21000
Floats	`float`	~27.3, 0.0, 48.21
Booleans	`bool`	true, false

Pairs:

If $e_1 : \tau_1$ and $e_2 : \tau_2$
then $(e_1, e_2) : \tau_1 * \tau_2$ — pair (tuple) type constructor

Functions:

if $f : \tau_1$ -> τ_2 and $a : \tau_1$ — function type constructor
then $f(a) : \tau_2$

Examples:

```
(4.0, 2):  float*int
power:  float*int -> float
power(4.0, 2):  float
```

* has higher precedence that ->

Type inference: power

```
let rec power (x,n) =
   match (x,n) with
   | (_,0) -> 1.0                    (* 1 *)
   | (x,n) -> x * power(x,n-1)       (* 2 *)
```

Type inference: `power`

```
let rec power (x,n) =
   match (x,n) with
   | (_,0) -> 1.0                    (* 1 *)
   | (x,n) -> x * power(x,n-1)      (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.

Type inference: `power`

```
let rec power (x,n) =
   match (x,n) with
   | (_,0) -> 1.0                  (* 1 *)
   | (x,n) -> x * power(x,n-1)     (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `float` because `1.0:float`   (Clause 1, function value.)

Type inference: `power`

```
let rec power (x,n) =
   match (x,n) with
   | (_,0) -> 1.0                  (* 1 *)
   | (x,n) -> x * power(x,n-1)     (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `float` because `1.0:float` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.

Type inference: `power`

```
let rec power (x,n) =
   match (x,n) with
   | (_,0) -> 1.0                    (* 1 *)
   | (x,n) -> x * power(x,n-1)       (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `float` because `1.0:float`   (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):float`, because τ_3 = `float`.

Type inference: power

```
let rec power (x,n) =
   match (x,n) with
   | (_,0) -> 1.0                    (* 1 *)
   | (x,n) -> x * power(x,n-1)      (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `float` because `1.0:float` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):float`, because τ_3 = `float`.
- multiplication can have

  `int*int -> int` or `float*float -> float`

  as types, but no “mixture” of `int` and `float`

Type inference: `power`

```
let rec power (x,n) =
   match (x,n) with
   | (_,0) -> 1.0                    (* 1 *)
   | (x,n) -> x * power(x,n-1)       (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `float` because `1.0:float` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):float`, because τ_3 = `float`.
- multiplication can have

  `int*int -> int` or `float*float -> float`

  as types, but no “mixture” of `int` and `float`
- Therefore `x:float` and τ_1=`float`.

Type inference: power

```
let rec power (x,n) =
   match (x,n) with
   | (_,0) -> 1.0                  (* 1 *)
   | (x,n) -> x * power(x,n-1)     (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `float` because `1.0:float` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):float`, because τ_3 = `float`.
- multiplication can have

  `int*int -> int` or `float*float -> float`

  as types, but no “mixture” of `int` and `float`
- Therefore `x:float` and τ_1=`float`.

The F# system determines the type `float*int -> float`

A higher-order version of the power function

We shall now look at a version of `power` $x\ n = x^n$ with the type

```
power: float -> (int -> float)
```

A higher-order version of the power function

We shall now look at a version of `power` $x\ n = x^n$ with the type

```
power: float -> (int -> float)
```

- the argument of power is the base x

A higher-order version of the power function

We shall now look at a version of `power` $x\ n = x^n$ with the type

```
power: float -> (int -> float)
```

- the argument of power is the base x
- and `power` x is the function that maps exponent n to x^n

A higher-order version of the power function

We shall now look at a version of `power` $x\ n = x^n$ with the type

```
power: float -> (int -> float)
```

- the argument of power is the base x
- and `power` x is the function that maps exponent n to x^n

The function may be evaluated in *stages*:

```
let pow2 = power 2.0;;

pow2 3;;
val it : float = 8.0
pow2 4;;
val it : float = 16.0
```

A higher-order version of the power function

We shall now look at a version of `power` $x\ n = x^n$ with the type

```
power: float -> (int -> float)
```

- the argument of power is the base x
- and `power` x is the function that maps exponent n to x^n

The function may be evaluated in *stages*:

```
let pow2 = power 2.0;;

pow2 3;;
val it : float = 8.0
pow2 4;;
val it : float = 16.0
```

This higher-order version of power is declared by

```
let rec power x n = match n with
    | 0 -> 1.0
    | _ -> x * power x (n-1);;
```

A higher-order version of the power function

We shall now look at a version of `power` $x\ n = x^n$ with the type

```
power: float -> (int -> float)
```

- the argument of power is the base x
- and `power` x is the function that maps exponent n to x^n

The function may be evaluated in *stages*:

```
let pow2 = power 2.0;;

pow2 3;;
val it : float = 8.0
pow2 4;;
val it : float = 16.0
```

This higher-order version of power is declared by

```
let rec power x n = match n with
    | 0 -> 1.0
    | _ -> x * power x (n-1);;
```

The value of the function is a function

A expression for anonymous functions

The function expression

$$
\begin{array}{l}
\texttt{function} \\
\mid \ pat_1 \ \rightarrow \ e_1 \\
\quad \vdots \\
\mid \ pat_n \ \rightarrow \ e_n
\end{array}
$$

allows you to “tabulate” argument-value pairs of a function.

```
function
| 2  -> 28   // February
| 4  -> 30   // April
| 6  -> 30   // June
| 9  -> 30   // September
| 11 -> 30   // November
| _  -> 31;; // All other months
 val it : int -> int = <fun:clo@17-2>
```

A expression for anonymous functions

The function expression

$$\begin{array}{l}\texttt{function}\\ \mid\ pat_1 \ \rightarrow\ e_1\\ \ \vdots\\ \mid\ pat_n \ \rightarrow\ e_n\end{array}$$

allows you to “tabulate” argument-value pairs of a function.

```
function
| 2  -> 28   // February
| 4  -> 30   // April
| 6  -> 30   // June
| 9  -> 30   // September
| 11 -> 30   // November
| _  -> 31;; // All other months
 val it : int -> int = <fun:clo@17-2>

it 2;;
val it : int = 28
```

Another higher-order version of the power function

We now have another look at `power` $x\ n = x^n$ with the type

```
power: float -> (int -> float)
```

Another higher-order version of the power function

We now have another look at `power` $x\ n = x^n$ with the type

```
power: float -> (int -> float)
```

The following declaration explicitly reveals that `power` x is a function:

```
let rec power x =
   function
   | 0 -> 1.0
   | n -> x * power x (n-1);;
```

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
`not`	`bool -> bool`	negation

```
not true = false
not false = true
```

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
`not`	`bool -> bool`	negation

```
not true = false
not false = true
```

Expressions

e_1 `&&` e_2	"conjunction $e_1 \wedge e_2$"
e_1 `||` e_2	"disjunction $e_1 \vee e_2$"

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
`not`	`bool -> bool`	negation

```
not true = false
not false = true
```

Expressions

e_1 `&&` e_2 "conjunction $e_1 \wedge e_2$"

e_1 `||` e_2 "disjunction $e_1 \vee e_2$"

— are lazily evaluated, e.g.

```
1<2 || 5/0 = 1
⇝ true
```

Precedence: `&&` has higher than `||`

Summary

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference
- higher-order functions

Part 2: Lists

- Lists: values and constructors
- Recursions following the structure of lists
- Polymorphism

- The list concept is a natural, built-in ingredient of functional languages

Lists

A list is a finite sequence of elements having the same type:

$[v_1; \ldots; v_n]$ ([] is called the empty list)

Lists

A list is a finite sequence of elements having the same type:

$[v_1; \ldots; v_n]$ ([ ] is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]
```

Lists

A list is a finite sequence of elements having the same type:

$[v_1; \ldots; v_n]$ ([ ] is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]

["a"; "ab"; "abc"; ""];;
val it : string list = ["a"; "ab"; "abc"; ""]
```

Lists

A list is a finite sequence of elements having the same type:

$[v_1; \ldots; v_n]$ ([ ] is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]

["a"; "ab"; "abc"; ""];;
val it : string list = ["a"; "ab"; "abc"; ""]

[sin; cos];;
val it : (float->float) list = [<fun:...>; <fun:...>]
```

Lists

A list is a finite sequence of elements having the same type:

$[v_1; \ldots; v_n]$ ([ ] is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]

["a"; "ab"; "abc"; ""];;
val it : string list = ["a"; "ab"; "abc"; ""]

[sin; cos];;
val it : (float->float) list = [<fun:...>; <fun:...>]

[(1,true); (3,true)];;
val it : (int * bool) list = [(1, true); (3, true)]
```

Lists

A list is a finite sequence of elements having the same type:

$[v_1; \ldots; v_n]$ ([] is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]

["a"; "ab"; "abc"; ""];;
val it : string list = ["a"; "ab"; "abc"; ""]

[sin; cos];;
val it : (float->float) list = [<fun:...>; <fun:...>]

[(1,true); (3,true)];;
val it : (int * bool) list = [(1, true); (3, true)]

[[]; [1]; [1;2]];;
val it : int list list = [[]; [1]; [1; 2]]
```

List constructors

A non-empty list $[x_1; x_2; \ldots; x_n]$, $n \geq 1$, consists of

- a *head* x_1 and
- a *tail* $[x_2; \ldots; x_n]$

List constructors

A non-empty list $[x_1; x_2; \ldots; x_n]$, $n \geq 1$, consists of

- a *head* x_1 and
- a *tail* $[x_2; \ldots; x_n]$

The list type has two constructors:

- The empty list $[]$
- The cons constructor $x_1 :: [x_2; \ldots; x_n] = [x_1; x_2; \ldots; x_n]$

– they are used to construct and to decompose lists

Recursion on lists – a simple example

$$\texttt{suml [}x_1\texttt{;}x_2\texttt{; ... ;}x_n\texttt{]} = \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n = x_1 + \sum_{i=2}^{n} x_i$$

Recursion on lists – a simple example

$$\texttt{suml [}x_1\texttt{;}x_2\texttt{; ... ;}x_n\texttt{]} = \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n = x_1 + \sum_{i=2}^{n} x_i$$

Constructors are used in list patterns

```
let rec suml xs =
   match xs with
   | []       -> 0
   | x::tail -> x + suml tail;;
val suml : int list -> int
```

Recursion on lists – a simple example

$$\texttt{suml}\ [x_1; x_2; \ldots; x_n] = \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n = x_1 + \sum_{i=2}^{n} x_i$$

Constructors are used in list patterns

```
let rec suml xs =
   match xs with
   | []       -> 0
   | x::tail -> x + suml tail;;
val suml : int list -> int
```

```
   suml [1;2]
⇝  1 + suml [2]          (x ↦ 1 and tail ↦ [2])
⇝  1 + (2 + suml [])     (x ↦ 2 and tail ↦ [])
⇝  1 + (2 + 0)           (the pattern [] matches the value [])
⇝  1 + 2
⇝  3
```

Recursion follows the structure of lists

A polymorphic list function (I)

The function remove y xs gives the list obtained from xs by deleting every occurrence of *y*, e.g. `remove` $2\ [1;2;0;2;7] = [1;0;7]$.

Recursion is following the structure of the list:

```
let rec remove y xs =
   match xs with
   | []                -> []
   | x::tail when x=y -> remove y tail
   | x::tail          -> x::remove y tail;;
```

A polymorphic list function (I)

The function remove y xs gives the list obtained from xs by deleting every occurrence of *y*, e.g. `remove` 2 [1; 2; 0; 2; 7] = [1; 0; 7].

Recursion is following the structure of the list:

```
let rec remove y xs =
   match xs with
   | []                  -> []
   | x::tail when x=y -> remove y tail
   | x::tail            -> x::remove y tail;;
```

List elements can be of any type that supports equality

```
remove :  'a -> 'a list -> 'a list when'a : equality
```

A polymorphic list function (I)

The function remove y xs gives the list obtained from xs by deleting every occurrence of *y*, e.g. `remove` 2 [1; 2; 0; 2; 7] = [1; 0; 7].

Recursion is following the structure of the list:

```
let rec remove y xs =
   match xs with
   | []               -> []
   | x::tail when x=y -> remove y tail
   | x::tail          -> x::remove y tail;;
```

List elements can be of any type that supports equality

```
remove :  'a -> 'a list -> 'a list when'a : equality
```

- `'a` is a *type variable*
- 'a : equality is a type constraint

A polymorphic list function (I)

The function remove y xs gives the list obtained from xs by deleting every occurrence of *y*, e.g. `remove` 2 [1; 2; 0; 2; 7] = [1; 0; 7].

Recursion is following the structure of the list:

```
let rec remove y xs =
   match xs with
   | []                -> []
   | x::tail when x=y -> remove y tail
   | x::tail           -> x::remove y tail;;
```

List elements can be of any type that supports equality

```
remove :  'a -> 'a list -> 'a list when'a : equality
```

- `'a` is a *type variable*
- 'a : equality is a type constraint

The F# system infers the most general type for remove

A polymorphic list function (II)

- A type containing type variables is called a polymorphic type
- The remove function is called a polymorphic function.

  `remove : 'a -> 'a list -> 'a list when 'a : equality`

The function has many forms, one for each instantiation of 'a:

A polymorphic list function (II)

- A type containing type variables is called a polymorphic type
- The remove function is called a polymorphic function.

  `remove : 'a -> 'a list -> 'a list when 'a : equality`

The function has many forms, one for each instantiation of 'a:

Instantiating `'a` with `int`:

```
remove 2 [1; 2; 0; 2; 7];;
val it : int list = [1; 0; 7]
```

A polymorphic list function (II)

- A type containing type variables is called a polymorphic type
- The remove function is called a polymorphic function.

  ```
  remove : 'a -> 'a list -> 'a list when 'a : equality
  ```

The function has many forms, one for each instantiation of 'a:

Instantiating `'a` with `int`:

```
remove 2 [1; 2; 0; 2; 7];;
val it : int list = [1; 0; 7]
```

Instantiating `'a` with `int list`:

```
remove [2] [[2;1]; [2]; [0;1]; [2]; [5;6;7]];;
val it : int list list = [[2; 1]; [0; 1]; [5; 6; 7]]
```

A polymorphic list function (II)

- A type containing type variables is called a polymorphic type
- The remove function is called a polymorphic function.

```
remove : 'a -> 'a list -> 'a list when 'a : equality
```

The function has many forms, one for each instantiation of 'a:

Instantiating `'a` with `int`:

```
remove 2 [1; 2; 0; 2; 7];;
val it : int list = [1; 0; 7]
```

Instantiating `'a` with `int list`:

```
remove [2] [[2;1]; [2]; [0;1]; [2]; [5;6;7]];;
val it : int list list = [[2; 1]; [0; 1]; [5; 6; 7]]
```

Notice that `->` associates to the right:

```
'a -> 'a list -> 'a list  means 'a -> ('a list -> 'a list)
```

Exploiting structured patterns: the isPrefix function

The function `isPrefix` *xs* *ys* tests whether the list *xs* is a prefix of the list *ys*, for example:

$$\begin{aligned} \texttt{isPrefix}\ [1;2;3]\ [1;2;3;8;9] &= \text{true} \\ \texttt{isPrefix}\ [1;2;3]\ [1;2;8;3;9] &= \text{false} \end{aligned}$$

Exploiting structured patterns: the isPrefix function

The function `isPrefix` *xs ys* tests whether the list *xs* is a prefix of the list *ys*, for example:

$$\texttt{isPrefix}\ [1;2;3]\ [1;2;3;8;9] = \text{true}$$
$$\texttt{isPrefix}\ [1;2;3]\ [1;2;8;3;9] = \text{false}$$

The function is declared as follows:

```
let rec isPrefix xs ys =
   match (xs,ys) with
   | ([],_)               -> true
   | (_,[])               -> false
   | (x::xtail,y::ytail) -> x=y && isPrefix xtail ytail;;

isPrefix [1;2;3] [1;2];;
val it : bool = false
```

Exploiting structured patterns: the isPrefix function

The function `isPrefix` *xs* *ys* tests whether the list *xs* is a prefix of the list *ys*, for example:

$$\begin{array}{lcl} \texttt{isPrefix}\ [1;2;3]\ [1;2;3;8;9] & = & \text{true} \\ \texttt{isPrefix}\ [1;2;3]\ [1;2;8;3;9] & = & \text{false} \end{array}$$

The function is declared as follows:

```
let rec isPrefix xs ys =
   match (xs,ys) with
   | ([],_)              -> true
   | (_,[])              -> false
   | (x::xtail,y::ytail) -> x=y && isPrefix xtail ytail;;

isPrefix [1;2;3] [1;2];;
val it : bool = false
```

A each clause expresses succinctly a natural property:

Exploiting structured patterns: the isPrefix function

The function `isPrefix` *xs ys* tests whether the list *xs* is a prefix of the list *ys*, for example:

$$\begin{array}{lcl} \texttt{isPrefix}\ [1;2;3]\ [1;2;3;8;9] & = & \text{true} \\ \texttt{isPrefix}\ [1;2;3]\ [1;2;8;3;9] & = & \text{false} \end{array}$$

The function is declared as follows:

```
let rec isPrefix xs ys =
   match (xs,ys) with
   | ([],_)              -> true
   | (_,[])              -> false
   | (x::xtail,y::ytail) -> x=y && isPrefix xtail ytail;;

isPrefix [1;2;3] [1;2];;
val it : bool = false
```

A each clause expresses succinctly a natural property:

- The empty list is a prefix of any list

Exploiting structured patterns: the isPrefix function

The function `isPrefix` *xs* *ys* tests whether the list *xs* is a prefix of the list *ys*, for example:

$$\begin{array}{lcl} \texttt{isPrefix}\ [1;2;3]\ [1;2;3;8;9] & = & \mathrm{true} \\ \texttt{isPrefix}\ [1;2;3]\ [1;2;8;3;9] & = & \mathrm{false} \end{array}$$

The function is declared as follows:

```
let rec isPrefix xs ys =
   match (xs,ys) with
   | ([],_)              -> true
   | (_,[])              -> false
   | (x::xtail,y::ytail) -> x=y && isPrefix xtail ytail;;

isPrefix [1;2;3] [1;2];;
val it : bool = false
```

A each clause expresses succinctly a natural property:

- The empty list is a prefix of any list
- A non-empty list is not a prefix of the empty list

Exploiting structured patterns: the isPrefix function

The function `isPrefix` *xs ys* tests whether the list *xs* is a prefix of the list *ys*, for example:

$$\begin{array}{lcl} \texttt{isPrefix}\ [1;2;3]\ [1;2;3;8;9] & = & \text{true} \\ \texttt{isPrefix}\ [1;2;3]\ [1;2;8;3;9] & = & \text{false} \end{array}$$

The function is declared as follows:

```
let rec isPrefix xs ys =
   match (xs,ys) with
   | ([],_)              -> true
   | (_,[])              -> false
   | (x::xtail,y::ytail) -> x=y && isPrefix xtail ytail;;

isPrefix [1;2;3] [1;2];;
val it : bool = false
```

A each clause expresses succinctly a natural property:

- The empty list is a prefix of any list
- A non-empty list is not a prefix of the empty list
- A non-empty list (...) is a prefix of another non-empty list (...) if ...

Summary

- Lists

Summary

- Lists
- Polymorphism

Summary

- Lists
- Polymorphism
- Constructors (:: and [] for lists)

Summary

- Lists
- Polymorphism
- Constructors (:: and [] for lists)
- Patterns

Summary

- Lists
- Polymorphism
- Constructors (:: and [] for lists)
- Patterns
- Recursion on the structure of lists

Summary

- Lists
- Polymorphism
- Constructors (:: and [] for lists)
- Patterns
- Recursion on the structure of lists
- Constructors used in patterns to decompose structured values

Summary

- Lists
- Polymorphism
- Constructors (:: and [] for lists)
- Patterns
- Recursion on the structure of lists
- Constructors used in patterns to decompose structured values
- Constructors used in expressions to compose structured values

Summary

- Lists
- Polymorphism
- Constructors (:: and [] for lists)
- Patterns
- Recursion on the structure of lists
- Constructors used in patterns to decompose structured values
- Constructors used in expressions to compose structured values

Blackboard exercises

- `memberOf` *x* *ys* is true iff *x* occurs in the list *ys*
- `insert`(*x*, *ys*) is the *ordered list* obtained from the *ordered list* *ys* by insertion of *x*
- `sort`(*xs*) gives a ordered version of *xs*