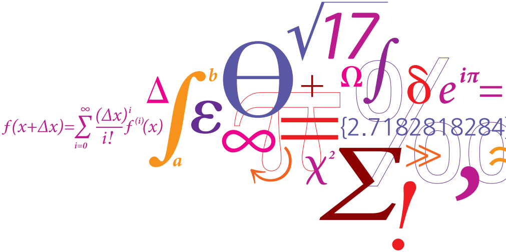
02157 Functional Programming

Lecture 1: Introduction and Getting Started

Michael R. Hansen



DTU Informatics
Department of Informatics and Mathematical Modelling

WELCOME to
02157 Functional Programming

Teacher: Michael R. Hansen
DTU Informatics, mrh@imm.dtu.dk

Teaching assistant: Phan Anh Dung, PhD Student.
Søren Olofsson, master's student
Both at DTU Informatics

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

Today: Friday, September 7.

- Introduction to functional programming and F#
  (341.23 — here)
- about 9:15 – lecture notes can be bought here.
- Make your first programs in the databar
  (341 Rooms: 015 and 019 — E-databar)
- Introduction to lists in F#
  (341.23 — here)
- Computations with polynomials in F#
  (341 Rooms: 015 and 019 — E-databar)

By noon you have solved a non-trivial problem using F#

Today: Friday, September 7.

- Introduction to functional programming and F#
  (341.23 — here)
- about 9:15 – lecture notes can be bought here.
- Make your first programs in the databar
  (341 Rooms: 015 and 019 — E-databar)
- Introduction to lists in F#
  (341.23 — here)
- Computations with polynomials in F#
  (341 Rooms: 015 and 019 — E-databar)

By noon you have solved a non-trivial problem using F#

Practical Matters

- Textbook: Functional Programming using F#, Chapters 1-7, 9.
  by Michael R. Hansen and Hans Rischel.

  Can be bought at the reception of DTU Informatics. Price 100 kr.

  Published by Cambridge University Press the coming winter.

- F# is an open-source functional language integrated in the Visual Studio development platform and with access to all features in the .NET program library. The language is also supported on Linux and MAC systems using the Mono platform.

- We use F# on the Windows platform in the E-databar.

- Look at homepage concerning installations for your own PC (Windows, Linux or Mac).

Practical Matters

- Textbook: Functional Programming using F#, Chapters 1-7, 9.
  by Michael R. Hansen and Hans Rischel.

  Can be bought at the reception of DTU Informatics. Price 100 kr.

  Published by Cambridge University Press the coming winter.

- F# is an open-source functional language integrated in the Visual Studio development platform and with access to all features in the .NET program library. The language is also supported on Linux and MAC systems using the Mono platform.

- We use F# on the Windows platform in the E-databar.

- Look at homepage concerning installations for your own PC (Windows, Linux or Mac).

Practical Matters

- Textbook: Functional Programming using F#, Chapters 1-7, 9.
  by Michael R. Hansen and Hans Rischel.

  Can be bought at the reception of DTU Informatics. Price 100 kr.

  Published by Cambridge University Press the coming winter.

- F# is an open-source functional language integrated in the Visual Studio development platform and with access to all features in the .NET program library. The language is also supported on Linux and MAC systems using the Mono platform.

- We use F# on the Windows platform in the E-databar.

- Look at homepage concerning installations for your own PC (Windows, Linux or Mac).

Practical Matters

- Textbook: Functional Programming using F#, Chapters 1-7, 9.
  by Michael R. Hansen and Hans Rischel.

  Can be bought at the reception of DTU Informatics. Price 100 kr.

  Published by Cambridge University Press the coming winter.

- F# is an open-source functional language integrated in the Visual Studio development platform and with access to all features in the .NET program library. The language is also supported on Linux and MAC systems using the Mono platform.

- We use F# on the Windows platform in the E-databar.

- Look at homepage concerning installations for your own PC (Windows, Linux or Mac).

Imperative models

- Imperative models of computations are expressed in terms of states and sequences of state-changing operations

Example:

```
i := 0;
s := 0;
while i < length(A)
  do s := s+A[i];
     i := i+1
  od
```

An imperative model describes *how* a solution is obtained

Imperative models

- Imperative models of computations are expressed in terms of states and sequences of state-changing operations

Example:

```
i := 0;
s := 0;
while i < length(A)
  do s := s+A[i];
     i := i+1
  od
```

An imperative model describes *how* a solution is obtained

Object-oriented models

- An object is characterized by a state and an interface specifying a collection of state-changing operations.
- Object-oriented models of computations are expressed in terms of a collection of objects which exchange messages by using interface operations.

Object-oriented models add structure to imperative models

An object-oriented model describes *how* a solution is obtained

Object-oriented models

- An object is characterized by a state and an interface specifying a collection of state-changing operations.
- Object-oriented models of computations are expressed in terms of a collection of objects which exchange messages by using interface operations.

Object-oriented models add structure to imperative models

An object-oriented model describes *how* a solution is obtained

Declarative models

In declarative models focus is on *what* a solution is.

- Logical programming (02156 Logical Systems and Logical Programming)
  - Programs are (typically) expressed in a fragment of first-order logic. The formulas have a standard meaning, as well as a procedural interpretation based on logical inferences.

- Functional programming
  - A program is expressed as a mathematical function

    $$f : A \rightarrow B$$

    and function applications guide computations.

Some advantages

- fast prototyping based on abstract concepts
- more advanced applications are within reach
- Supplement modelling and problem solving techniques
- Execute in parallel on multi-core platforms

F# is as efficient as C#

Declarative models

In declarative models focus is on *what* a solution is.

- Logical programming (02156 Logical Systems and Logical Programming)
  - Programs are (typically) expressed in a fragment of first-order logic. The formulas have a standard meaning, as well as a procedural interpretation based on logical inferences.

- Functional programming
  - A program is expressed as a mathematical function

    $$f : A \to B$$

    and function applications guide computations.

Some advantages

- fast prototyping based on abstract concepts
- more advanced applications are within reach
- Supplement modelling and problem solving techniques
- Execute in parallel on multi-core platforms

F# is as efficient as C#

Declarative models

In declarative models focus is on *what* a solution is.

- Logical programming (02156 Logical Systems and Logical Programming)
  - Programs are (typically) expressed in a fragment of first-order logic. The formulas have a standard meaning, as well as a procedural interpretation based on logical inferences.
- Functional programming
  - A program is expressed as a mathematical function

$$f : A \to B$$

    and function applications guide computations.

Some advantages

- fast prototyping based on abstract concepts
- more advanced applications are within reach
- Supplement modelling and problem solving techniques
- Execute in parallel on multi-core platforms

F# is as efficient as C#

Some functional programming background

In functional programming, the model of computation is the application of functions to arguments. no side-effects

- Introduction of λ-calculus 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 $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of the ”variable-free” programming language FP (Backus 1977), by providing a rich collection of functionals (combining forms for functions).
- Introduction of functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) in the 1970s.

Some functional programming background

In functional programming, the model of computation is the application of functions to arguments. no side-effects

- Introduction of λ-calculus 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 $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of the "variable-free" programming language FP (Backus 1977), by providing a rich collection of functionals (combining forms for functions).
- Introduction of functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) in the 1970s.

Some functional programming background

In functional programming, the model of computation is the application of functions to arguments. no side-effects

- Introduction of λ-calculus 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 $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of the "variable-free" programming language FP (Backus 1977), by providing a rich collection of functionals (combining forms for functions).
- Introduction of functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) in the 1970s.

Some functional programming background

In functional programming, the model of computation is the application of functions to arguments.

no side-effects

- Introduction of λ-calculus 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 $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of the ”variable-free” programming language FP (Backus 1977), by providing a rich collection of functionals (combining forms for functions).
- Introduction of functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) in the 1970s.

Some background of the "SML-family"

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML-like systems (SML, OCAML, F#, . . .) have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .

- F# is now integrated in the .net environment

- Declarative aspects are sneaking into more "main stream languages"

- Often used to teach high-level programming concepts

Some background of the "SML-family"

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML-like systems (SML, OCAML, F#, . . .) have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .

- F# is now integrated in the .net environment

- Declarative aspects are sneaking into more "main stream languages"

- Often used to teach high-level programming concepts

Some background of the “SML-family”

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML-like systems (SML, OCAML, F#, . . .) have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .

- F# is now integrated in the .net environment

- Declarative aspects are sneaking into more ”main stream languages”

- Often used to teach high-level programming concepts

Some background of the “SML-family”

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML-like systems (SML, OCAML, F#, . . .) have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .

- F# is now integrated in the .net environment

- Declarative aspects are sneaking into more ”main stream languages”

- Often used to teach high-level programming concepts

Some background of the “SML-family”

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML-like systems (SML, OCAML, F#, . . .) have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .

- F# is now integrated in the .net environment
- Declarative aspects are sneaking into more ”main stream languages”
- Often used to teach high-level programming concepts

Some background of the "SML-family"

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML-like systems (SML, OCAML, F#, . . .) have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .

- F# is now integrated in the .net environment
- Declarative aspects are sneaking into more "main stream languages"
- Often used to teach high-level programming concepts

Overview of the course

- Functional programming concepts and techniques
- A model-based programming approach using a functional language with a strong type system.
- Program correctness, including structural induction and well-founded induction

Fun with a variety of applications, such as

- a library for piecewise linear curves – with applications
- a Sudoko solver
- an interpreter for a simple programming language
- a lambda-calculus interpreter
- a model checker for CTL – a temporal logic
- ...

Homepage for the course: `www.imm.dtu.dk/courses/02157`

Overview of the course

- Functional programming concepts and techniques
- A model-based programming approach using a functional language with a strong type system.
- Program correctness, including structural induction and well-founded induction

Fun with a variety of applications, such as

- a library for piecewise linear curves – with applications
- a Sudoko solver
- an interpreter for a simple programming language
- a lambda-calculus interpreter
- a model checker for CTL – a temporal logic
- ...

Homepage for the course: `www.imm.dtu.dk/courses/02157`

Overview of the course

- Functional programming concepts and techniques
- A model-based programming approach using a functional language with a strong type system.
- Program correctness, including structural induction and well-founded induction

Fun with a variety of applications, such as

- a library for piecewise linear curves – with applications
- a Sudoko solver
- an interpreter for a simple programming language
- a lambda-calculus interpreter
- a model checker for CTL – a temporal logic
- ...

Homepage for the course: `www.imm.dtu.dk/courses/02157`

A major goal

Teach abstraction (not a concrete programming language)

- Modelling
- Design
- Programming

Why?

More complex problems can be solved in an succinct, elegant and understandable manner

How?

Solving a broad class of problems showing the applicability of the theory, concepts, techniques and tools.

Functional programming techniques once mastered are useful for the design of programs in other programming paradigms as well.

A major goal

Teach abstraction (not a concrete programming language)

- Modelling
- Design
- Programming

Why?

More complex problems can be solved in an succinct, elegant and understandable manner

How?

Solving a broad class of problems showing the applicability of the theory, concepts, techniques and tools.

Functional programming techniques once mastered are useful for the design of programs in other programming paradigms as well.

A major goal

Teach abstraction (not a concrete programming language)

- Modelling
- Design
- Programming

Why?

More complex problems can be solved in an succinct, elegant and understandable manner

How?

Solving a broad class of problems showing the applicability of the theory, concepts, techniques and tools.

Functional programming techniques once mastered are useful for the design of programs in other programming paradigms as well.

A major goal

Teach abstraction (not a concrete programming language)

- Modelling
- Design
- Programming

Why?

More complex problems can be solved in an succinct, elegant and understandable manner

How?

Solving a broad class of problems showing the applicability of the theory, concepts, techniques and tools.

Functional programming techniques once mastered are useful for the design of programs in other programming paradigms as well.

F# supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Imperative and object-oriented programming

  assignments, loops, arrays, objects, Input/Output, etc.

Programming as a modelling discipline

- High-level programming, declarative programming, executable declarative specifications B, Z, VDM, RAISE
- Fast time-to-market

F# supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Imperative and object-oriented programming

  assignments, loops, arrays, objects, Input/Output, etc.

Programming as a modelling discipline

- High-level programming, declarative programming, executable declarative specifications B, Z, VDM, RAISE
- Fast time-to-market

F# supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Imperative and object-oriented programming

  assignments, loops, arrays, objects, Input/Output, etc.

Programming as a modelling discipline

- High-level programming, declarative programming, executable declarative specifications B, Z, VDM, RAISE
- Fast time-to-market

F# supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Imperative and object-oriented programming

  assignments, loops, arrays, objects, Input/Output, etc.

Programming as a modelling discipline

- High-level programming, declarative programming, executable declarative specifications B, Z, VDM, RAISE
- Fast time-to-market

F# supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Imperative and object-oriented programming

  assignments, loops, arrays, objects, Input/Output, etc.

Programming as a modelling discipline

- High-level programming, declarative programming, executable declarative specifications B, Z, VDM, RAISE
- Fast time-to-market

Course context

Prerequisites for 02157: Programming in an imperative/object-oriented language, discrete mathematics, algorithms and data structure, as obtained, for example, from the bachelor programme in Software Technology.

Successor course of 02157:

02257 Applied functional programming
3-weeks period January

An extension of 02157 that aims at an effective use of functional programming in connection with courses and projects at the M.Sc. programme in computer science and engineering, and in industrial applications.

- Computer science applications. Interpreter for a programming language, for example.
- "Practical applications". Involving a database, for example.
- Functional pearls. Monadic parsing, for example.

Course context

Prerequisites for 02157: Programming in an imperative/object-oriented language, discrete mathematics, algorithms and data structure, as obtained, for example, from the bachelor programme in Software Technology.

Successor course of 02157:

02257 Applied functional programming
3-weeks period January

An extension of 02157 that aims at an effective use of functional programming in connection with courses and projects at the M.Sc. programme in computer science and engineering, and in industrial applications.

- Computer science applications. Interpreter for a programming language, for example.
- “Practical applications”. Involving a database, for example.
- Functional pearls. Monadic parsing, for example.

Overview of Getting Started

Main functional ingredients of F#:

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

GOAL: By the end of this first part you have constructed succinct, elegant and understandable F# programs, e.g. for

- $\mathrm{sum}(m, n) = \sum_{i=m}^{n} i$
- Fibonacci numbers ($F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$)
- Binomial coefficients $\binom{n}{k}$

Overview of Getting Started

Main functional ingredients of F#:

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

GOAL: By the end of this first part you have constructed succinct, elegant and understandable F# programs, e.g. for

- $\mathrm{sum}(m, n) = \sum_{i=m}^{n} i$
- Fibonacci numbers ($F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$)
- Binomial coefficients $\binom{n}{k}$

The Interactive Environment

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

⇐ Input to the F# system

⇐ 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”

The Interactive Environment

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

⇐ Input to the F# system

⇐ 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"

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”

The Interactive Environment

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

⇐ Input to the F# system

⇐ 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

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

$$\left[\begin{array}{lll} \texttt{price} & \mapsto & 125 \\ \texttt{newPrice} & \mapsto & 250 \\ \texttt{it} & \mapsto & \text{false} \end{array}\right]$$

is called an environment

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

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

$$\left[\begin{array}{lll} \texttt{price} & \mapsto & 125 \\ \texttt{newPrice} & \mapsto & 250 \\ \texttt{it} & \mapsto & \mathrm{false} \end{array}\right]$$

is called an environment

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

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` $f\,x = e$

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` $f\,x = e$

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` $f\,x = e$

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
```

Anonymous functions: by example (1)

An anonymous function computing the number of days in a month:

```
function
| 1 -> 31 // January
| 2 -> 28 // February // not a leap year
| 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
... warning ... Incomplete pattern matches ...
val it : int -> int = <fun:clo@17-2>

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

A function expression with a pattern for every month

Anonymous functions: by example (1)

An anonymous function computing the number of days in a month:

```
function
| 1 -> 31 // January
| 2 -> 28 // February // not a leap year
| 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
... warning ... Incomplete pattern matches ...
val it : int -> int = <fun:clo@17-2>

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

A function expression with a pattern for every month

Anonymous functions: by example (1)

An anonymous function computing the number of days in a month:

```
function
| 1 -> 31 // January
| 2 -> 28 // February // not a leap year
| 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
... warning ... Incomplete pattern matches ...
val it : int -> int = <fun:clo@17-2>
```

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

A function expression with a pattern for every month

Anonymous functions: by example (2)

One *wildcard pattern* _ can cover many similar cases:

```
function
 | 2  -> 28  // February
 | 4  -> 30  // April
 | 6  -> 30  // June
 | 9  -> 30  // September
 | 11 -> 30  // November
 |  _  -> 31;;// All other months
```

An even more succinct definition can be given using an *or*-pattern:

```
function
 | 2          -> 28  // February
 | 4|6|9|11 -> 30  // April, June, September, November
 | _          -> 31  // All other months
;;
```

Anonymous functions: by example (2)

One *wildcard pattern* _ can cover many similar cases:

```
function
 | 2  -> 28  // February
 | 4  -> 30  // April
 | 6  -> 30  // June
 | 9  -> 30  // September
 | 11 -> 30  // November
 |  _  -> 31;;// All other months
```

An even more succinct definition can be given using an *or*-pattern:

```
function
 | 2         -> 28  // February
 | 4|6|9|11 -> 30  // April, June, September, November
 | _         -> 31  // All other months
;;
```

Recursion. Example $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition:

recursion formula

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

Computation:

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

Recursion. Example $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition: recursion formula

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

Computation:

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

Recursive declaration. Example $n!$

Function declaration:

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

Evaluation:

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

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

Recursive declaration. Example $n!$

Function declaration:

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

Evaluation:

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

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

Recursive declaration. Example $n!$

Function declaration:

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

Evaluation:

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

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

Recursive declaration. Example $n!$

Function declaration:

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

Evaluation:

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

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

Recursive declaration. Example $n!$

Function declaration:

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

Evaluation:

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

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

Recursive declaration. Example $n!$

Function declaration:

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

Evaluation:

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

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

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

Mathematical definition: recursion formula

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

Function declaration:

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

Patterns:

(_, 0) matches any pair of the form $(x, 0)$.
The wildcard pattern _ matches any value.

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

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

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

Mathematical definition: recursion formula

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

Function declaration:

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

Patterns:

(_, 0) matches any pair of the form $(x, 0)$.
The wildcard pattern _ matches any value.

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

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

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 = function
  | (_,0) -> 1.0                (* 1 *)
  | (x,n) -> x * power(x,n-1)   (* 2 *)
```

Patterns:

`(_,0)` matches any pair of the form $(x, 0)$.
The wildcard pattern `_` matches any value.

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

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

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

Function declaration:

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

Evaluation:

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

If-then-else expressions

Form:

$$\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

Evaluation rules:

$$\texttt{if}\ \mathrm{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \rightsquigarrow \quad e_1$$
$$\texttt{if}\ \mathrm{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \rightsquigarrow \quad e_2$$

Alternative declarations:

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

let rec power(x,n)   =  if n=0 then 1.0
                        else x * power(x,n-1);
```

Use of patterns usually gives more understandable programs

If-then-else expressions

Form:

$$\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

Evaluation rules:

$$\texttt{if}\ \mathrm{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \rightsquigarrow \quad e_1$$

$$\texttt{if}\ \mathrm{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \rightsquigarrow \quad e_2$$

Alternative declarations:

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

let rec power(x,n)   =  if n=0 then 1.0
                        else x * power(x,n-1);
```

Use of patterns usually gives more understandable programs

If-then-else expressions

Form:

`if` b `then` e_1 `else` e_2

Evaluation rules:

`if` true `then` e_1 `else` e_2 $\rightsquigarrow$ e_1

`if` false `then` e_1 `else` e_2 $\rightsquigarrow$ e_2

Alternative declarations:

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

let rec power(x,n)   =  if n=0 then 1.0
                        else x * power(x,n-1);
```

Use of patterns usually gives more understandable programs

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 ||

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 ||

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 ||

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
`String.length`	`string -> int`	length of string
`+`	`string*string -> string`	concatenation
`= < <= ...`	`string*string -> bool`	comparisons
`string`	`obj -> string`	conversions

Examples

```
- "auto" < "car";
> val it = true : bool

- "abc"+"de";
> val it = "abcde": string
```

```
- String.length("abc"^"def");
> val it = 6 : int

- string(6+18);
> val it = "24": string
```

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
`String.length`	`string -> int`	length of string
`+`	`string*string -> string`	concatenation
`= < <= ...`	`string*string -> bool`	comparisons
`string`	`obj -> string`	conversions

Examples

```
- "auto" < "car";
> val it = true : bool

- "abc"+"de";
> val it = "abcde": string
```

```
- String.length("abc"^"def");
> val it = 6 : int

- string(6+18);
> val it = "24": string
```

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
`String.length`	`string -> int`	length of string
`+`	`string*string -> string`	concatenation
`= < <= ...`	`string*string -> bool`	comparisons
`string`	`obj -> string`	conversions

Examples

```
- "auto" < "car";
> val it = true : bool

- "abc"+"de";
> val it = "abcde": string
```

```
- String.length("abc"^"def");
> val it = 6 : int

- string(6+18);
> val it = "24": string
```

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 ->

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 ->

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 ->

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 = function
  | (_,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`

Type inference: power

```
let rec power = function
  | (_,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`

Type inference: `power`

```
let rec power = function
  | (_,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`

Type inference: power

```
let rec power = function
  | (_,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`

Type inference: power

```
let rec power = function
  | (_,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`

Type inference: power

```
let rec power = function
  | (_,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`

Type inference: power

```
let rec power = function
  | (_,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`

Type inference: power

```
let rec power = function
  | (_,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`

Summary

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

Breath first round through many concepts aiming at program construction from the first day.

We will go deeper into each of the concepts later in the course.

Summary

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

Breath first round through many concepts aiming at program construction from the first day.

We will go deeper into each of the concepts later in the course.

Overview

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

The purpose of this lecture is to give you an (as short as possible) introduction to lists, so that you can solve a problem which can illustrate some of F#'s high-level features.

This part is *not* intended as a comprehensive presentation on lists, and we will return to the topic again later.

Overview

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

The purpose of this lecture is to give you an (as short as possible) introduction to lists, so that you can solve a problem which can illustrate some of F#'s high-level features.

This part is *not* intended as a comprehensive presentation on lists, and we will return to the topic again later.

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]]
```

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]]
```

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]]
```

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]]
```

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]]
```

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]]
```

Trees for lists

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

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

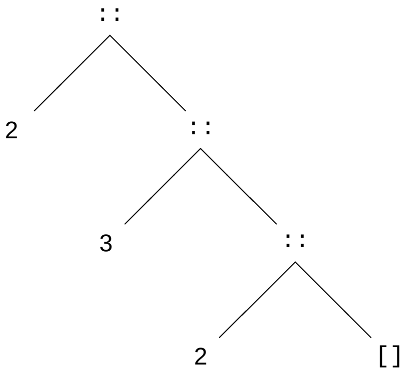
Graph for [2,3,2]

Graph for [2]

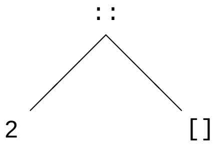
Trees for lists

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

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



Graph for `[2,3,2]`



Graph for `[2]`

List constructors: [ ] and : :

Lists are generated as follows:

- the empty list is a list, designated [ ]
- if x is an element and xs is a list, then so is $x :: xs$ (type consistency)

: : associate to the right, i.e. $x_1::x_2::xs$ means $x_1::(x_2::xs)$

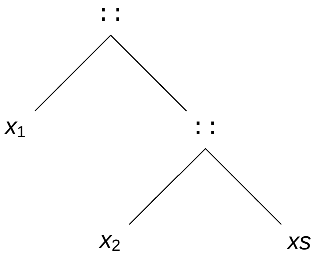
Graph for $x_1::x_2::xs$

List constructors: [ ] and ::

Lists are generated as follows:

- the empty list is a list, designated [ ]
- if x is an element and xs is a list,
  then so is $x :: xs$ (type consistency)

:: associate to the right, i.e. $x_1::x_2::xs$ means $x_1::(x_2::xs)$



Graph for $x_1::x_2::xs$

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 = function
  | []    -> 0
  | x::xs -> x + suml xs;;
> val suml : int list -> int
```

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

Recursion follows the structure of lists

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 = function
  | []    -> 0
  | x::xs -> x + suml xs;;
> val suml : int list -> int
```

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

Recursion follows the structure of lists

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 = function
  | []    -> 0
  | x::xs -> x + suml xs;;
> val suml : int list -> int
```

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

Recursion follows the structure of lists

Infix functions

It is possible to declare infix functions in F#, i.e. the function symbol is between the arguments.

The *prefix function* on lists is declared as follows:

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

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

- The special way of declaring the function `(<=.) xs ys` makes `<=.` an infix operator
- The `match (xs,ys)` construct allows for branching out on patterns for `(xs,ys)`

Suitable use of infix functions can increase readability significantly

Infix functions

It is possible to declare infix functions in F#, i.e. the function symbol is between the arguments.

The *prefix function* on lists is declared as follows:

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

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

- The special way of declaring the function `(<=.) xs ys` makes `<=.` an infix operator
- The `match (xs,ys)` construct allows for branching out on patterns for `(xs,ys)`

Suitable use of infix functions can increase readability significantly

Infix functions

It is possible to declare infix functions in F#, i.e. the function symbol is between the arguments.

The *prefix function* on lists is declared as follows:

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

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

- The special way of declaring the function `(<=.) xs ys` makes `<=.` an infix operator
- The `match (xs,ys)` construct allows for branching out on patterns for `(xs,ys)`

Suitable use of infix functions can increase readability significantly

Infix functions

It is possible to declare infix functions in F#, i.e. the function symbol is between the arguments.

The *prefix function* on lists is declared as follows:

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

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

- The special way of declaring the function `(<=.) xs ys` makes `<=.` an infix operator
- The `match (xs,ys)` construct allows for branching out on patterns for `(xs,ys)`

Suitable use of infix functions can increase readability significantly

Exercises

- `length` *xs* : the length of the list *xs* (is a predefined function).
- `remove`(*x*, *ys*) : removes all occurrences of *x* in the list *ys*

Have fun with your first non-trivial functional program:
polynomials represented as lists

Exercises

- `length` *xs* : the length of the list *xs* (is a predefined function).
- `remove`(*x*, *ys*) : removes all occurrences of *x* in the list *ys*

Have fun with your first non-trivial functional program:
polynomials represented as lists