Introduction

Lars-Henrik Eriksson

Functional Programming 1

Presentation originally by Tjark Weber

based on notes by Sven-Olof Nyström

Welcome!

Lars-Henrik Eriksson	Sven-Olof Nyström	Albert Mingkun Yang
Lecturer	Assisting teacher	Lab assistant

Today

- Practical matters
- General information
- Introduction to Functional Programming
- Introduction to Standard ML

Course Registration

To get credit for the course, you must be admitted and **registered**!

If you are admitted already, please **web-register** yourself for this course on the Student Portal as soon as possible.

If you are admitted but cannot web-register for any reason, contact the Student Office (`it-kansli@it.uu.se`).

Late Admission

If you are not **admitted** yet:

- Master students should contact their programme counsellor.
- Exchange students should contact Ulrika Jaresund (`Ulrika.Jaresund@it.uu.se`), the exchange student coordinator at the IT Department.
- Students with an older registration, who want to re-register, should contact the Student Office (`it-kansli@it.uu.se`).
- Students on the reserve list will be contacted by the Student Office if they are admitted.
- Everyone else should apply at `http://www.antagning.se`, via Late application ("Sen anmälan").

Dropping Courses

We hope that you will enjoy this course!

However, if you decide to **drop** it, you must inform the Student Office (`it-kansli@it.uu.se`).

- If less than 3 weeks have passed since the course started, your course registration will simply be removed.
- After 3 weeks a “course intermission” will be reported to UPPDOK instead.

Teachers cannot help with admission/registration. Please contact the Student Office (IT-kansliet) if you have questions about these formalities.

Special Needs

If you have special needs (for instance, if you need more time on exams), please contact the responsible coordinator: see

`http://teknat.uu.se/student/stod-och-service/sarskilt-stod/`

Also consider informing your teachers.

The Course

5 ECTS credits (hp) $\approx$ 135 hours of work:

- 12 lectures during August/September/October
- 7 labs
- 4 assignments (3 group, 1 individual)
- Programming exercise (take-home exam)

Web page:	`https://studentportalen.uu.se/`
Language:	English

The Student Portal

General course and lab information, ethics rules and a coding convention are all available through the Student Portal.

Lecture slides, example programs and lab assignments will be made available on the Student Portal.

Solutions to assignments must be submitted via the Student Portal.

Of course, you can also contact your teachers by email and in person.
(But submissions must be done using the Student Portal.)

Your Background and Expectations

- Which programming language(s) are you most familiar with?
- What do you expect (or would you like) to learn in this course?

Please get out your laptop or smartphone, go to

`www.menti.com`

and use the code

40 41 14

Course Contents

Fundamental concepts: functions, relations, recursion, tail-recursion, type systems, polymorphism, datatypes, recursive datatypes, introduction to higher-order functions, data abstraction.

Programming in a functional programming language, such as Standard ML.

Similarities and differences with imperative and object-oriented programming.

Learning Outcomes

In order to pass, the student must be able to . . .

- list and define the fundamental concepts of functional programming.
- manually execute a given (simple) functional program.
- manually infer the type of a given (simple) functional program.
- implement (simple) algorithms and data structures as functional programs.
- design (large) functional programs that are modular and have reusable components.
- explain on a simple problem how functional programming differs from imperative and object-oriented programming.

Reading List

At least one of the following books:

- Hansen, Michael R.; Rischel, Hans:
  Introduction to Programming using SML
  Addison-Wesley, 1999 – 355 p.
- Paulson, Lawrence C.:
  ML for the Working Programmer
  2nd ed., Cambridge Univ. Press, 1996 – 476 p.
- Ullman, Jeffrey D.:
  Elements of ML Programming
  ML97 ed., Prentice Hall, 1998 – 383 p.

Buy one today!

Assessment

Assignments are mandatory. You must pass all four assignments to pass the course. (Attending the lectures and labs is optional, but highly recommended!)

The take-home exam is voluntary, and only required for grades 4 and 5. You can only take the exam once.

Cheating

Plagiarism and cheating are serious academic offenses and can lead to suspension from the university for six months.

Cheating includes:

- Knowingly using some fellow student's solution to an exercise while solving it.
- Knowingly submitting a (changed) copy of some fellow student's solution.
- Knowingly submitting a solution based on a hardcopy or Internet publication without citing it.
- Knowingly helping some student to do any of the three actions above.

Golden rule: **Give credit when you use someone else's ideas.**

Introduction to Functional Programming

Programming Paradigms

imperative	(object-oriented)
logic-based	$\boxed{\text{functional}}$

Why Do We Care?

Klarna ERICSSON facebook

Google IBM Microsoft

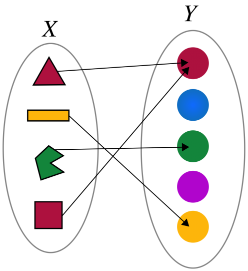
Functional Programming Languages

- Lisp (J. McCarthy, 1962)
- Scheme (Sussmann & Steele 1975)
- FP (J. Backus, 1977)
- LCF, ML (Meta Language) (Milner et al., 1977)
- CAML (Huet et al., 1990)
- SML (Standard ML) (Milner et al., 1990)
- Haskell (P. Hudak, 1990)
- Erlang (Armstrong et al., 1993)
- Scala (M. Odersky, 2003)
- F# (D. Syme, 2005)
- . . .

Functions

A (total) **function** is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to *exactly one* output.

`http://en.wikipedia.org/wiki/Function_%28mathematics%29`

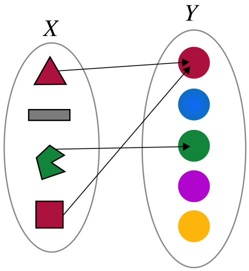


We write $f\colon X \to Y$ for a function f from X to Y.

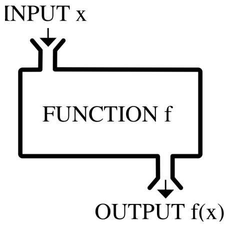
Partial Functions

A **partial function** is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to *at most one* output.

Thus, a partial function may be undefined for some (or even all) inputs.



Specifying Functions



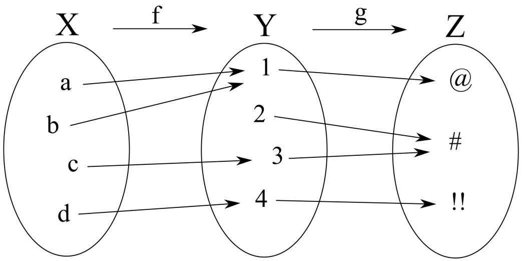
A function may be given by an **expression** or algorithm that determines the output value. For example,

```
fun double x = 2 * x
```

Function Composition

Function composition is the application of one function to the results of another.

`http://en.wikipedia.org/wiki/Function_composition`



Principles of Functional Programming

Functional programming treats computation as the **evaluation of mathematical functions**, while avoiding state and mutable data.

`http://en.wikipedia.org/wiki/Functional_programming`

Fundamental principles:

- Execution by evaluation of expressions
- Declaration of functions
- Application of functions
- Recursion
- Inductive data types

Introduction to SML

Standard ML

Standard ML (SML) is a general-purpose, modular, functional programming language. It is garbage collected, has compile-time type inference and type checking, and is type safe.

SML is a relatively clear and simple language—virtually perfect for an introduction to functional programming. At the same time, it is being used for large software projects and cutting-edge research. Also, it has a formal semantics.

The community is small, and the infrastructure (IDEs, debuggers, libraries, ...) and documentation may not be as well-developed as for more popular languages.

SML Expressions

```
32+15

3.12+4.1

10 - 100

not true

"fun" ^ "ctional"

size("hello")

size "hello"

if 2+2=5 then "hello" else "goodbye"
```

SML Expressions (cont.)

SML expressions can be evaluated **interactively**. Note that the system also determines the type of each result.

```
> 32+15;
val it = 47: int
> 3.12+4.1;
val it = 7.22: real
> 10 - 100;
val it = ~90: int
> not true;
val it = false: bool
> "fun" ^ "ctional";
val it = "functional": string
> size("hello");
val it = 5: int
> size "hello";
val it = 5: int
> if 2+2=5 then "hello" else "goodbye";
val it = "goodbye": string
```

Evaluation of Expressions

Reduce to normal form:

$$
\begin{array}{l}
\texttt{3 + 4 * 2 < 5 * 2} \\
\quad\longrightarrow \texttt{3 + 8 < 5 * 2} \\
\quad\longrightarrow \texttt{11 < 5 * 2} \\
\quad\longrightarrow \texttt{11 < 10} \\
\quad\longrightarrow \texttt{false}
\end{array}
$$

Of course, you know how to evaluate arithmetic expressions. We will extend these principles to cover other aspects of the language.

Evaluation of Conditional Expressions

```
if 2+2=5 then "hel" ^ "lo" else "good" ^ "bye"
 ⟶ if 4=5 then "hel" ^ lo" else "good" ^ "bye"
 ⟶ if false then "hel" ^ "lo" else "good" ^ "bye"
 ⟶ "good" ^ "bye"
 ⟶ "goodbye"
```

Note that `if ... then ... else ...` is an **expression** that evaluates to a value (and not a command that modifies some state).

Value Declarations

Value declarations associate a value to an identifier.

Like other SML code, they may be given interactively or as part of an SML program.

```
val a = 1;
val pi = 3.14159;
val two_pi = 2.0 * pi;
val &@!+< = 42;

two_pi;
a + a;
&@!+<  div 7;
```

Bindings and Environments

An **environment** determines the types and values of identifiers.

```
val sum = 24;
val sum = 3.51;
```

These value declarations give rise to two different environments.

In any given environment, the value of an identifier can never change.
Identifiers are *not* variables!

Function Declarations

Like other SML code, function declarations may be given interactively or as part of an SML program.

```
fun sqr x = x*x;
fun abs x = if x >= 0 then x else ~x;
```

SML uses **eager evaluation**: arguments are evaluated before functions are applied.

```
sqr (sqr 2)
 ⟶ sqr (2*2)
 ⟶ sqr 4
 ⟶ 4*4
 ⟶ 16
```

Type Inference

The SML system **infers** the types of identifiers.

Examples:

```
> fun abs x = if x >= 0 then x else ~x;
val abs = fn: int -> int

> fun abs x = if x >= 0.0 then x else ~x;
val abs = fn: real -> real

> fun test x = if x<0 then "negative" else "non-negative";
val test = fn: int -> string
```

Tuples

Two or more values can form a **tuple**.

Examples:

```
(1, 2, 3)
(1.1, 1.2)
("foo", "bar")
(3, "three", 3.0)
```

Functions with Multiple Arguments

The naive approach: use tuples.

```
fun average (a,b) = (a+b)/2.0
fun max (a,b) = if a>b then a else b
```

The elegant approach: use **currying**.

```
fun average a b = (a+b)/2.0
fun max a b = if a>b then a else b
```

Recursion

Functions may call themselves:

```
fun triangle n =
  if n=0 then 0 else n + triangle (n-1)
```

Recursion

Functions may call themselves:

```
fun triangle n =
  if n=0 then 0 else n + triangle (n-1)
```

triangle n evaluates to the sum $n + \cdots + 2 + 1 + 0$.

Recursion: Evaluation

triangle 2
$\longrightarrow$ **if** 2=0 **then** 0 **else** 2 + triangle (2−1)
$\longrightarrow$ **if** false **then** 0 **else** 2 + triangle (2−1)
$\longrightarrow$ 2 + triangle (2−1)
$\longrightarrow$ 2 + triangle 1
$\longrightarrow$ 2 + (**if** 1=0 **then** 0 **else** 1 + triangle (1−1))
$\longrightarrow$ 2 + (**if** false **then** 0 **else** 1 + triangle (1−1))
$\longrightarrow$ 2 + (1 + triangle (1−1))
$\longrightarrow$ 2 + (1 + triangle 0)
$\longrightarrow$ 2 + (1 + (**if** 0=0 **then** 0 **else** 0 + triangle (0−1)))
$\longrightarrow$ 2 + (1 + (**if** true **then** 0 **else** 0 + triangle (0−1)))
$\longrightarrow$ 2 + (1 + 0)
$\longrightarrow$ 2 + 1
$\longrightarrow$ 3

Recursion: Comment

Why isn't the definiton of `triangle` circular?

What will this function compute?

```
fun silly x = silly x
```

Lists

Lists are defined by these two rules:

- [] is a list, the *empty list*.
- If x is an element (the *head*) and xs is a list (the *tail*) whose elements are of the same type as x, then x :: xs is a list.

Examples:	More succinct notation:
[ ]	[ ]
3 :: [ ]	[3]
2 :: 3 :: [ ]	[2 , 3]
1 :: 2 :: 3 :: [ ]	[1 , 2 , 3]

Recursion Over Lists

Example:

```
fun sum [] = 0
  | sum (x::xs) = x + sum xs
```

Let's test the function:

```
sum []
sum [1,2,3]
```