

# Programming Languages, Interpreters, and Compilers

IFC

Programming Languages, F2002

Page 1-1

- The course, teachers, lecture, exercises, literature
- Course plan and goals
- The Standard ML programming language
- The Moscow ML system
- Abstract syntax versus concrete syntax
- Representing simple arithmetic expressions
- Direct evaluation of expressions
- Evaluation environments that map variables to values

IFC

Programming Languages, F2002

Page 1-2

## The course, teachers, lectures, exercises

- Teachers: Peter Sestoft (lectures) and Andrzej Wasowski (exercises)
- The course home page is <http://www.it-c.dk/courses/F00/F2002/>
- The home page presents necessary resources, latest news, ...
- Course plan
- The lecture slides are available from the course plan (on the Web)
- Please do ask questions during lectures
- Get Hansen and Rischel's *Introduction to Programming Using Standard ML*  
It would be good to read Chapters 1, 2, 3, 5, 7-1-7-5, 8, 1, 8, 3, and 8.4 soon, and chapter 9 in a few weeks.
- We shall use parts of Mogensen: *Basics of Compiler Design*
- Otherwise, lecture slides, lecture notes, and example code
- Exercise classes: Wednesday 9h-12h in room 3.19
- Weekly exercises are given, some of which must be handed in to Andrzej; see separate exercise sheets
- Solve at least some exercises at home, and discuss and finish them at Wednesday morning

IFC

Programming Languages, F2002

Page 1-3

## Programming languages everywhere

### General-purpose programming languages

Fortran, COBOL, Algol, Pascal, C, Simula, Smalltalk, C++, Java, C#, JavaScript, Python, Prolog, ...

### Special-purpose programming languages

Postscript — for controlling printers and photocopiers:

```
/TeXDict 300 dict def TeXDict begin/N(def)def/B(bind def)N/S{exch}N/X(S
N)B/A{dup}B/TR{translate}N/IsIs false N/vsize N/hsize 11 72 mul N/hsiz 8.5 72
mul N/landplus90{false}def/@origin{IsIs[{0 landplus90[1 -1]{-1 1}ifelse 0 ...
```

Quake C — for writing computer games:

```
void(entity targ, attacker) ClientObituary = {
  if (targ.classname == "player" && attacker.classname == "monster_army") {
    bprint (targ.netname);
  } else
    bprint (" has stupidly been shot by a dummy grunt\n");
  inherited ();
};
```

IFC

Programming Languages, F2002

Page 1-4

## Course contents and goals

- The Standard ML programming language:

Goals: Use as tool (meta-language) to present other languages. Also, to learn a language different from Java.

- Syntax description and tools:

Goal: Provide notations and practical tools to solve tricky and frequently occurring tasks.

Regular expressions, scanner generators (mosmllex).

Context-free grammars, parser generators (mosmlyac).

- Programming language concepts:

Goal: Provide concepts and vocabulary that enable you to understand new languages that you must use.

Syntax (form) versus semantics (meaning); interpretation versus compilation; abstract machines.

Abstract syntax versus concrete syntax; and static semantics versus dynamic semantics.

- Programming language paradigms:

Goal: Use programming languages more competently and computers more efficiently.

Functional: expressions and functions; environment and closures; no assignment.

Imperative: expressions, statements, and procedures; environment and store.

Object-oriented: expressions, statements, and methods; environment, store, objects, and heap.

IT-C

## The Standard ML (SML) programming language: a crash course

- Expressions and types
- Variables and variable bindings
- Scope, let, local
- Function declarations, and recursive functions
- Pattern matching and case
- Pairs and tuples
- Lists, cons (::), and append (@)
- Exceptions
- Datatypes
- Using datatypes to represent trees and expressions

IT-C

## Running SML programs: the Moscow ML system

Moscow ML is an *implementation* of the SML *programming language* — there are several others.

It is installed at IT-C and you can download and install it at home: see the course home page.

Moscow ML permits interactive execution of SML program phrases:

```
[sestoft@jones sestoft]$ mosml -P full
Moscow ML version 2.00 (June 2000)
Enter 'quit()' to quit.
- 3+4;
> val it = 7 : int
- val res = 3+4;
> val res = 7 : int
- res;
> val it = 7 : int
```

You may run Moscow ML inside the Emacs editor: Type Alt+X shell then mosml -P full.

There's much documentation on the Moscow ML home page:

- Moscow ML Owner's Manual: general use, mosmllex, mosmlyac
- Moscow ML Library Documentation: all built-in libraries; see especially the List and String libraries
- Moscow ML Language Reference: precise syntax of the language, and most important libraries

IT-C

## SML arithmetic expressions, the int type

```
3+4;
```

## SML variables and declarations

```
val res = 3+4; (* variable binding, not assignment *)
res;
```

## SML arithmetic expressions, the real type

```
Math.sqrt 2.0;
```

## SML logical expressions, the bool type

```
3 < 4;
val res = 3 < 4;
```

## SML conditional expressions

```
if 3 < 4 then 117 else 118;
```

## SML string operations

```
val title = "Professor";
val name = "Tausen";
val junkmail = "Dear " ^ title ^ " " ^ name ^ ", You have won $$$!";
val n = size junkmail;
val junkmail2 =
String.concat["Dear ", title, " ", name, ", You have won $$$!"];
```

IT-C

### SML: limiting the scope of a binding

The scope of a variable is that part of the program in which it may be used.

Use `let` to limit the scope of a variable in an expression:

```
val x = 5; (* declare x *)

let val x = 3 < 4 (* a new x is declared *)
in
  if x then 117 else 118
end;

x; (* outer x still has the value 5 *)

Use local to limit the scope of a variable in a declaration:

local val x = 3 < 4 (* yet another x is declared *)
in
  val z = if x then 117 else 118
end;

x; (* outer x still has the value 5 *)
```

SML (and Java) have *nested scopes*: an inner declaration can make a hole in the scope of an outer variable.

FIG

Programming Languages, F2002

Page 1-9

### SML pattern matching

```
fun fac 0 = 1
  | fac n = n * fac(n-1);
```

fac 7;

### SML case expressions — same function as above

```
fun fac n =
  case n of
    0 => 1
  | _ => n * fac(n-1); (* the _ matches anything *)
```

### SML pairs and tuples, product types

```
val w = (2, true, 3.4, "blah");

fun add (x, y) = x + y;
add (2, 3);

val noon = (12, 0);
val talk = (15, 15);

fun earlier ((h1, m1), (h2, m2)) =
  h1 < h2 orelse (h1 = h2 andalso m1 < m2)
```

FIG

Programming Languages, F2002

Page 1-11

### SML function declarations

```
fun circleArea r = Math.pi * r * r;

val a = circleArea 10.0;

fun double x = 2.0 * x;

double 3.5;

fun junkmail title name =
  "Dear " ^ title ^ " " ^ name ^ ", You have won $$$!";

SML recursive function declarations

fun fac n = if n=0 then 1 else n * fac(n-1);

fac 7;
```

### SML lists of integers

```
val x1 = 7 :: 9 :: 13 :: [];
val x2 = [7, 9, 13];
val equal = (x1 = x2);

fun sum [] = 0
  | sum (x::xr) = x + sum xr;

fun prod [] = 1
  | prod (x::xr) = x * prod xr;

fun len [] = 0
  | len (x::xr) = 1 + len xr;

val x2sum = sum x2;
val x2prod = prod x2;
val x2len = len x2;

val x3 = [47, 11];

val x1x3 = x1 @ x3; (* append lists x1 and x3 *)
```

FIG

Programming Languages, F2002

Page 1-10

FIG

Programming Languages, F2002

Page 1-12

### Summary of Standard ML concepts

- **Types:** int, real, bool, string, int list, int\*int
- **Expressions:** 117, 3.14, true, "A38", [2,3,5,7], (3, 14)  
x + y  
if x <> 0.0 then 1000/x else 1.0  
case xs of [] => true | \_ => false  
let val x = 0.1  
in  
if x <> 0.0 then 1000.0/x else 1.0  
end
- **Declarations:**  
val x = 0.1  
fun isEmpty (xs : int list) = case xs of [] => true | \_ => false  
local val MAX = 1023  
in  
fun lim x = if x > MAX then MAX else x  
end

FLC

### Efficiency of list operations

```
fun rev [] = []
  | rev (x::xr) = rev xr @ [x];
  (* inefficient *)

val x1r = rev x1;

local
  fun irev [] = res
    | irev (x::xr) = irev xr (x :: res)
  in
    fun rev xs = irev xs []
  end;
  (* efficient *)

rev x1;

SML exceptions

exception IllegalHour

fun mins h =
  if h < 0 orelse h > 23 then raise IllegalHour
  else h * 60;
```

FLC

### SML datatypes

```
datatype person =
  Student of string
  | Teacher of string * int
  (* name and phone no *)

val people = [Student "Niels Jørgen", Teacher("Peter", 831)];

fun getphone (Teacher(name, phone)) = phone
  | getphone (Student name) = raise Fail "no phone";

The option datatype

datatype intopt =
  SOME of int
  | NONE

fun getphone (Teacher(name, phone)) = SOME phone
  | getphone (Student name) = NONE;
```

Instead of intopt we could use the built-in type int option — more about that next week.

FLC

### Comparing SML datatypes and Java class hierarchies

The person datatype with constructors Student and Teacher and function getphone in Java:

```
abstract class Person {
  String name;
  abstract public Integer getPhone();
}

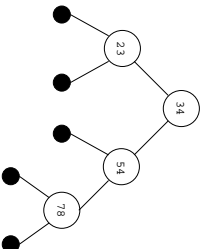
class Student extends Person {
  public Student(String name) {
    this.name = name;
  }
  public Integer getPhone() { return null; } // null instead of NONE
}

class Teacher extends Person {
  int phone;
  public Teacher(String name, int phone) {
    this.name = name; this.phone = phone;
  }
  public Integer getPhone() { return new Integer(phone); }
}

...
LinkedList people = new LinkedList();
people.add(new Student("Niels Jørgen"));
people.add(new Teacher("Peter", 831));
```

FLC

## Binary trees



### Representing binary tree using recursive datatypes

```
datatype inttree =
  Lf
  | Br of int * inttree * inttree;

val t1 = Br(34, Br(23, Lf, Lf), Br(54, Lf, Br(78, Lf, Lf)));

fun sumtree Lf
  | sumtree (Br(v, t1, t2)) = v + sumtree t1 + sumtree t2;

val t1sum = sumtree t1;
```

FC

Programming Languages, F2002

Page 1-17

### Let's do some exercises ...

- Write an SML function `sqr` : `int -> int` so that `sqr x` returns  $x^2$ .
- Write an SML function `pow` : `int -> int -> int` so that `pow x n` returns  $x^n$ .
- Write an SML function `dup` : `string -> string` that concatenates a string with itself.
- Write an SML function `dupn` : `string -> int -> string` so that `dupn s n` creates the concatenation of `n` copies of `s`.
- Assume the time of day is represented as a pair `(hh, mm)` : `int * int`. Write an SML function `timediff` : `int * int -> int * int -> int` so that `timediff t1 t2` computes the difference in minutes between `t1` and `t2`.
- Write an SML function `minutes` : `int * int -> int` to compute the number of minutes since midnight.
- Write an SML function `downto` : `int -> int list` so that `downto n` returns the `n`-element list `[n, n-1, ..., 1]`. Use `if-then-else` expressions to define the function.
- Define the `downto` function again, now using pattern matching.

FC

Programming Languages, F2002

Page 1-18

### Representing programming language concepts in Standard ML

- Representing expressions without variables:
- Evaluating expressions
- Representing expressions with variables
- Evaluation environments represented as association lists

FC

Programming Languages, F2002

Page 1-19

### Using SML as meta-language: modelling other languages

A language of very simple expressions: integer constants and primitive operators:

`17`

`3 + 4`

`(7 * 9) + 10`

Such *object language* expressions can be represented using recursive datatypes:

```
datatype expr =
  CstI of int
  | Prim of string * expr list;

val e1 = CstI 17;
val e2 = Prim("+", [CstI 3, CstI 4]);
val e3 = Prim("*", [Prim("+", [CstI 7, CstI 9]), CstI 10]);
```

How represent `3 - (5 + 8)`?

How represent `177 - 177`?

FC

Programming Languages, F2002

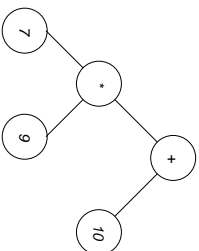
Page 1-20

### Concrete syntax

$(7 * 9) + 10$

$7 * 9 + 10$

### Abstract syntax tree



### Abstract syntax term

```
Prim("+", [Prim("**", [CSTI 7, CSTI 9]), CSTI 10])
```

For now we work on abstract syntax. In two weeks we shall take a closer look at concrete syntax.

IFC

Programming Languages, F2002

Page 1-21

### Changing the meaning of subtraction

The `eval` function defines the meaning or *semantics* of our small expression language.

We can decide ourselves what an object language expression means.

For instance, we may decide that subtraction `e1-e2` cannot produce negative results:

```
fun eval (e : expr) : int =
  case e of
  CSTI i => i
  | Prim("+", [e1, e2]) => eval e1 + eval e2
  | Prim("**", [e1, e2]) => eval e1 * eval e2
  | Prim("-", [e1, e2]) =>
    let val res = eval e1 - eval e2
    in if res < 0 then 0 else res end
  | Prim _
  => raise Fail "unknown primitive";

val e4v = eval (Prim("-", [CSTI 10, CSTI 27]));
```

IFC

Programming Languages, F2002

Page 1-23

### Evaluating expressions using recursive functions

```
fun eval (e : expr) : int =
  case e of
  CSTI i => i
  | Prim("+", [e1, e2]) => eval e1 + eval e2
  | Prim("**", [e1, e2]) => eval e1 * eval e2
  | Prim("-", [e1, e2]) => eval e1 - eval e2
  | Prim _
  => raise Fail "unknown primitive";

val e1v = eval e1;
val e2v = eval e2;
val e3v = eval e3;
```

IFC

Programming Languages, F2002

Page 1-22

### Association lists

An association list is a list of pairs (name, value) of a name and a value.

An association list has type `(string * int) list`.

```
val env = [("a", 3), ("c", 78), ("Daf", 666), ("D", 111)];
val emptyenv = []: (* the empty environment *)

fun lookup []
  |> lookup ((Y, v)::r) x = if x=Y then v else lookup r x;

lookup env "c";
```

An association list may be used as an *evaluation environment*

The purpose of an evaluation environment is to bind variables to their values.

IFC

Programming Languages, F2002

Page 1-24

## Object language expressions with variables

17

3 + a

(b \* 9) + a

### Abstract syntax

```
datatype expr =
  | CstI of int
  | Var of string
  | Prim of string * expr list

val e1 = CstI 17;
val e2 = Prim("+", [CstI 3, Var "a"]);
val e3 = Prim("++", [Prim("**", [Var "b", CstI 9]), Var "a"]);
```

IFC

## Object language expressions with variable bindings and nested scope

```
datatype expr =
  | CstI of int
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr list

val e1 = Let("z", CstI 17, Prim("+", [Var "z", Var "z"]));
val e2 = Let("z", CstI 17,
  Prim("+", [Let("z", CstI 22, Prim("**", [CstI 100, Var "z"])),
    Var "z"]));
```

IFC

## Evaluation within an environment

```
fun eval e (env : (string * int) list) : int =
  case e of
    CstI i => i
  | Var x => lookup env x
  | Prim("+", [e1, e2]) => eval e1 env + eval e2 env
  | Prim("*", [e1, e2]) => eval e1 env * eval e2 env
  | Prim("-", [e1, e2]) => eval e1 env - eval e2 env
  | Prim _ => raise Fail "unknown primitive"

val e1v = eval e1 env;
val e2v1 = eval e2 env;
val e2v2 = eval e2 [{"a", 314}];
val e3v = eval e3 env;
```

IFC

## Evaluation with variables and nested scope

```
fun eval e (env : (string * int) list) : int =
  case e of
    CstI i => i
  | Var x => lookup env x
  | Let(x, ehs, ebody) =>
    let val xval = eval ehs env
        val env1 = (x, xval) :: env
    in eval ebody env1 end
  | Prim("+", [e1, e2]) => eval e1 env + eval e2 env
  | Prim("*", [e1, e2]) => eval e1 env * eval e2 env
  | Prim("-", [e1, e2]) => eval e1 env - eval e2 env
  | Prim _ => raise Fail "unknown primitive"

val e1v = eval e1 emptyenv;
val e2v = eval e2 emptyenv;
```

IFC