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 photo-setters:

```
/TexDict 300 dict def TeXDict begin/N{def}def/B{bind def}N/S{exch}N/B/A{dup}B/TR{translate}N/isls false N/vsize 11 72 mul N/hsize 8.5 mul N/landplus90{false}def/@rigin{isls}{[0 landplus90{ 1 -1}{-1 1}ifelse
```

Quake C — for writing computer games:

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


FORTRAN ----> FORTRAN77

ALGOL ----> ALGOL68

LISP

PROLOG

BETA

JAVA

C#

C++

SASM

HASKELL

SML

SCHEME

SMALLTALK

GJ

Java 5.0

C# 2.0

VISUAL BASIC

ADA

FORTRAN90

BASIC

PASCAL

COBOL

FORTRAN
This part of the course

- The Standard ML programming language:
  Goals: Use as tool (meta-language) to present other languages. Also, to learn a yet another language different from Java.

- 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.
Plan & Material

- 3 lectures on SML
- Readings: Harper draft book on SML (see course home page).
- The slides, based on slides of Peter Sestoft.
- 1 lecture on expression problem (to compare with OO)
- Reading: article by Mads Tørgersen (see course home page).
ML at ITU

- Definition of Standard ML (Tofte)
- The ML Kit compiler (Tofte, Birkedal, Elsman)
- Region-based Memory Management (Tofte, Birkedal, Elsman)
- Moscow ML compiler (Sestoft)
- Standard Basis Library (Sestoft)
- SML Server (Elsman)
- SMLtoJs (Elsman)
- Theory associated with ML-like languages (PLS group)
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
Running SML programs: the Moscow ML system

Moscow ML is an *implementation* of the SML *programming language* — there are several other implementations. It is installed at ITU 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 2.01 (January 2004)
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 \texttt{Alt+X shell then mosml -P full}.

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

- Moscow ML Owner’s Manual: general use, \texttt{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
SML basics

SML arithmetic expressions, the \texttt{int} type

\begin{verbatim}
3+4;
\end{verbatim}

SML variables and declarations

\begin{verbatim}
val res = 3+4;  (* variable binding, not assignment *)
res;
\end{verbatim}

SML arithmetic expressions, the \texttt{real} type

\begin{verbatim}
Math.sqrt 2.0;
\end{verbatim}

SML logical expressions, the \texttt{bool} type

\begin{verbatim}
3 < 4;
val res = 3 < 4;
\end{verbatim}
SML basics, II

SML conditional expressions

if 3 < 4 then 117 else 118;

SML string operations

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

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:

```sml
code
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:

```sml
code
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.
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 $$$!";

fun fac n = if n=0 then 1 else n * fac(n-1);
fac 7;
SML pattern matching

```sml
fun fac 0 = 1
    | fac n = n * fac(n-1);
fac 7;
```

SML case expressions — same function as above

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

SML pairs and tuples, product types

```sml
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)
```
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 *)
Summary of Standard ML concepts so far

- **Types**: int, real, bool, string, int list, int*int, real->real

- **Expressions**: 117, 3.14, true, "A38", [2,3,5,7], (3, 14), Math.sqrt

x + y
if x <> 0.0 then 1000.0/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
SML type check and type errors

The compiler complains and refuses to compile programs that contain type errors:

```ml
- sum [7.0, 9.0, 13.0];
  ! Toplevel input:
  ! sum [7.0, 9.0, 13.0];
  !       ^^^
  ! Type clash: expression of type 
  !    real
  ! cannot have type
  !    int
```

Sometimes syntactic mistakes cause type errors — what’s wrong here?

```ml
fun fac n = if n=0 then 1 else n * fac;
  ! Toplevel input:
  ! fun fac n = if n=0 then 1 else n * fac;
  !       ^^^
  ! Type clash: expression of type 
  !    int -> int
  ! cannot have type
  !    int
```
More SML concepts: Exceptions

Use to signal problems at run-time.
Very much like Java/C++ exceptions — actually, those were inspired by SML's exceptions.

```ml
exception IllegalHour

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

```sml
datatype person =
    Student of string (* name *)
  | 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";
```

SML strings, lists, datatype values and tuples are stored on a heap, like objects in Java and C#.
An SML variable contains only a reference to its value; the value is not copying when assigned or passed.
A datatype value or tuple can be shared freely because it is immutable: its fields cannot be updated. Why is this important?
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.
Comparing SML datatypes and Java class hierarchies

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

```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 0
}
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));
```
Representing binary tree using recursive datatypes

```ml
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 = 0 |
  sumtree (Br(v, t1, t2)) = v + sumtree t1 + sumtree t2;

val t1sum = sumtree t1;
```
Let’s do some exercises . . .

- Write an SML function \( \text{sqr} : \text{int} \rightarrow \text{int} \) so that \( \text{sqr} \ x \) returns \( x^2 \).
- Write an SML function \( \text{pow} : \text{int} \rightarrow \text{int} \rightarrow \text{int} \) so that \( \text{pow} \ x \ n \) returns \( x^n \).
- Write an SML function \( \text{dup} : \text{string} \rightarrow \text{string} \) that concatenates a string with itself.
- Write an SML function \( \text{dupn} : \text{string} \rightarrow \text{int} \rightarrow \text{string} \) so that \( \text{dupn} \ s \ n \) creates the concatenation of \( n \) copies of \( s \).
- Assume the time of day is represented as a pair \( (\text{hh}, \text{mm}) : \text{int} \times \text{int} \).
  Write an SML function \( \text{timediff} : \text{int} \times \text{int} \rightarrow \text{int} \times \text{int} \rightarrow \text{int} \) so that \( \text{timediff} \ t1 \ t2 \) computes the difference in minutes between \( t1 \) and \( t2 \).
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.
What is the difference between the types `int -> int -> int` and `int * int -> int`?

A function of type `int * int -> int` takes one argument which is a pair:

```ml
fun addp (x, y) = x + y;
val res1 = addp(17, 25);
```

A function of type `int -> int -> int` takes two separate arguments:

```ml
fun addc x y = x + y;
val res2 = addc 17 25;
```

Alternatively, it can be applied to a single argument to produce a function of type `int -> int`:

```ml
val addSeventeen = addc 17;
> val addSeventeen = fn : int -> int
```
This function may be applied as many times as desired:

```ml
val res3 = addSeventeen 25;
val res4 = addSeventeen 100;
```

Function `addc` is called a *Curried* version of function `addp`; named after the logician Haskell B. Curry. Curried functions can be implemented using inner classes in Java.
Lars Birkedal (PLS, ITU)  
Introduction to ML  
March, 2008 27 / 34
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:

```ml
datatype expr =
  CstI of int
| Prim of string * expr * expr
```

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

How to represent 3 - (5 + 8) ?
How to represent 177 - 177 ?
Concrete syntax

\[(7 \times 9) + 10\]
\[7 \times 9 + 10\]

Abstract syntax tree (AST)

Abstract syntax term

\[\text{Prim}("+", \text{Prim}("\times", \text{CstI} 7, \text{CstI} 9), \text{CstI} 10)\]

We’ll work on abstract syntax.
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;
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 $e_1 - e_2$ cannot produce negative results:

```ml
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));
```
Association lists

An association list is a list of pairs \((\text{name}, \text{value})\) of a name and a value.

An association list has type \((\text{string} \times \text{int})\ \text{list}\).

```ml
val env = ["a", 3], ["c", 78], ["baf", 666], ["b", 111];
val emptyenv = []; (* the empty environment *)

fun lookup [] x = raise Fail (x ^ " not found")
    | 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.
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 * expr

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