UNIVERSITETET
I OSLO

The Algol family and ML

Volker Stolz
stolz@ifi.uio.no

Department of Informatics – University of Oslo

Initially by Gerardo Schneider.
Based on John C. Mitchell's slides (Stanford U.)

INF 3110 - 2016

ML lectures

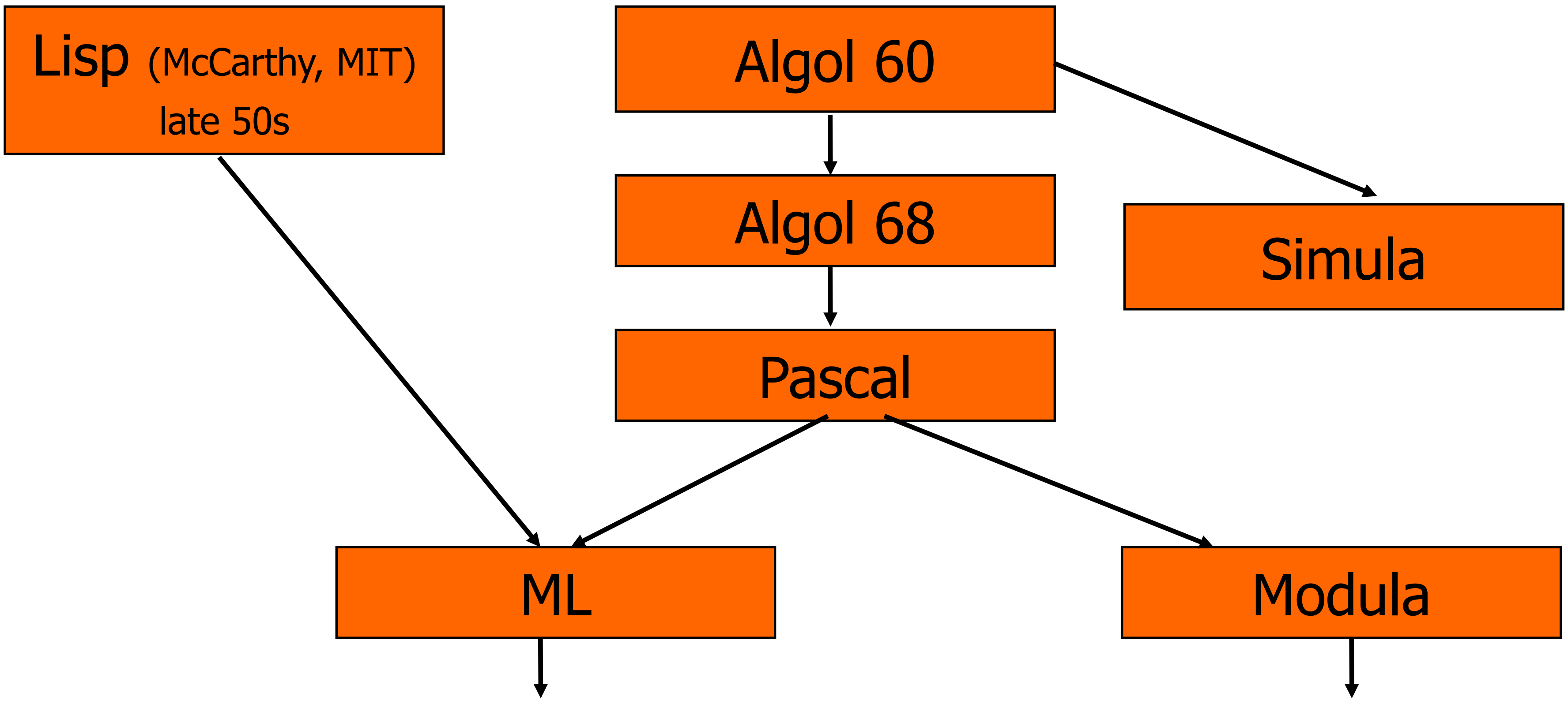
- **16.09: The Algol Family and ML (Mitchell's chap. 5)**
- 23.09: More on ML & Types (chap. 5 and 6)
- 21.10: More on Types, Type Inference and Polymorphism (chap. 6)
- 28.10: Control in sequential languages, Exceptions and Continuations (chap. 8)
- Prolog I / Prolog ||

INF 3110 - 2016

Outline

- Brief overview of Algol-like programming languages (Mitchell, Chapter 5)
  - Algol 60
  - Algol 68
  - Pascal
  - Modula
  - C
- Basic ML (Mitchell's Chapter 5 + more)

A (partial) Language Sequence



Many other languages in the "family":

Algol 58, Algol W, Euclid, Ada, Simula 67, BCPL, Modula-2, Oberon, Modula-3 (DEC), Delphi, …

Algol 60

- Designed: 1958-1963 (J. Backus, J. McCarthy, A. Perlis,…)
- General purpose language. Features:
  - Simple imperative language + functions
  - Successful syntax, used by many successors
    - Statement oriented
    - begin … end blocks (like C { … }) (local variables)
    - if … then … else
  - BNF (Backus Normal Form)
    - Became the standard for describing syntax
  - ALGOL became a standard language to describe algorithms.
  - Recursive functions and stack storage allocation
  - Fewer ad hoc restrictions than Fortran
    - General array references: A[x + B[3]*y]
    - Parameters in procedure calls
  - Primitive static type system

Algol 60 Sample

```
real procedure average(A,n);
   real array A; integer n;          ← no array bounds
   begin
      real sum; sum := 0;
      for i = 1 step 1 until n do
            sum := sum + A[i];
      average := sum/n               ← no ";" here
   end;
```

set procedure return value by assignment

Some trouble spots in Algol 60

- Shortcoming of its type discipline
  - Type "array" as a procedure parameter
    - no array bounds
  - "procedure" can be a parameter type
    - no argument or return types for procedure parameter
- Parameter passing methods
  - *Pass-by-name* had various anomalies (side effects)
  - *Pass-by-value* expensive for arrays
- Some awkward control issues
  - *goto* out of a block requires memory management

Algol 60 Pass-by-name

- Substitute text of actual parameter *(copy rule)*
  - Unpredictable with side effects!
- Example

```
procedure inc2(i, j);
   integer i, j;
   begin
      i := i+1;
      j := j+1
   end;
inc2 (k, A[k]);
```

→

```
begin
      k := k+1;
      A[k] := A[k] +1
end;
```

Is this what you expected?

Algol 68

- Intended to improve Algol 60
  - Systematic, regular type system
- Parameter passing
  - Eliminated pass-by-name (introduced *pass-by-reference*)
  - Pass-by-value and pass-by-reference using pointers
- Storage management
  - Local storage on stack
  - Heap storage, explicit *alloc* and garbage collection
- Considered difficult to understand
  - New terminology
    - types were called "modes"
    - arrays were called "multiple values"
  - Elaborate type system (e.g. array of pointers to procedures)
  - Complicated type conversions

Pascal

- Designed by N. Wirth (70s)
- Evolved from Algol W
- Revised type system of Algol
  - Good data-structuring concepts (based on C.A.R. Hoare's ideas)
    - records, variants (union type), subranges (e.g. [1…10])
  - More restrictive than Algol 60/68
    - Procedure parameters cannot have procedure parameters
- Popular teaching language (over 20 years! Till the 90s)
- Simple one-pass compiler

Procedure parameters in Pascal

- Allowed

```
procedure Proc1(i,j: integer);

procedure Proc2(procedure P(i:integer); i,j: integer);
```

- Not allowed

```
procedure NotA(procedure Proc3(procedure P(i:integer)));
```

Limitations of Pascal

- Array bounds part of type

  procedure p(a : array [1..10] of integer)

  procedure p(n: integer, a : array [1..n] of integer)

  illegal

  - Practical drawbacks:
    - Types cannot contain variables
    - How to write a generic *sort* procedure?
      - Only for arrays of some fixed length

  How could this have happened? Emphasis on teaching

- Not successful for "industrial-strength" projects

INF 3110 - 2016

Modula

- Designed by N. Wirth (late 70s)
- Descendent of Pascal
- Main innovation over Pascal: Module system
  - Modules allow certain declarations to be grouped together
    - *Definition module*: interface
    - *Implementation module*: implementation
- Modules in Modula provides minimal information hiding

C Programming Language

- Designed for writing Unix by Dennis Ritchie † 2011
- Evolved from B, which was based on BCPL
  - B was an untyped language; C adds some checking
- Relation between arrays and pointers
  - An array is treated as a pointer to first element
  - E1[E2] is equivalent to **pointer dereference** *((E1)+(E2))
  - Pointer arithmetic is *not* common in other languages
- Popular language
  - Memory model close to the underlying hardware
  - Many programmers like C flexibility (?!)
  - However weak type checking can just as well be seen as a disadvantage.

ML

- A *function-oriented imperative language* (or a mostly functional language with imperative features)
- Typed programming language. Clean and expressive type system.
- Sound type system (type checking), but not unpleasantly restrictive.
- Intended for interactive use ... (but not only...)
- Combination of Lisp and Algol-like features
  - Expression-oriented, Higher-order functions, Garbage collection, Abstract data types, Module system, Exceptions
- General purpose non-C-like, not OO language

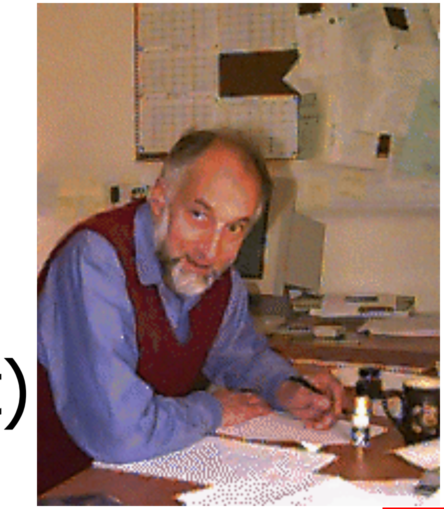
Why study ML ?

- Learn to think and solve problems in new ways
- All programming languages have a functional "part" - useful to know
- Verifiable programming: Easier to reason about a functional language, and to prove properties of programs
- More compact (simple?) code
- Higher order functions
- Certain aspects are easier to understand and program (e.g. recursion)

Why study ML ?

- Learn a different PL
- Discuss general PL issues
  - Types and type checking (Mitchell's chapter 6)
    - General issues in static/dynamic typing
    - Type inference
    - Polymorphism and Generic Programming
  - Memory management (Mitchell's chapter 7)
    - Static scope and block structure
    - Function activation records, higher-order functions
  - Control (Mitchell's chapter 8)
    - Exceptions
    - Tail recursion and continuations
    - Force and delay

Origin of ML



- Designed by R. Milner † 2010 (70s and 80s)
- Logic for Computable Functions (LCF project)
  - Based on Dana Scott's ideas (1969)
    - Formulate logic to prove properties of typed func. prog.
    - Simply typed (polymorphic) lambda calculus.
  - Milner's goals
    - Project to automate logic
    - Notation for programs
    - Notation for assertions and proofs
    - Write programs that find proofs
      - Too much work to construct full formal proof by hand
    - Make sure proofs are correct
  - ***M***eta-***L***anguage of the LCF system

LCF proof search

- *Proof tactic*: function that tries to find a proof

tactic(formula) = { succeed and return proof; search forever; fail }

- Express tactics in the Meta-Language (ML)
- Use a *type system* to distinguish successful from unsuccessful proofs and to facilitate correctness

Tactics in ML type system

- Tactic has a functional type

  tactic : formula → proof

- What if the formula is not correct and there is no proof?

  Type system must allow "failure"

tactic(formula) = { succeed and return proof / search forever / fail and *raise exception* }

- First type-safe exception mechanism!

Function types in ML

$f : A \rightarrow B$ means

for every $x \in A$,

$$f(x) = \begin{cases} \text{some element } y=f(x) \in B \\ \text{run forever} \\ \text{terminate by raising an exception} \end{cases}$$

Later development of ML

- Developed into different dialects
- Standard ML 1983, SML 1997
- CML: Concurrent ML (USA)
- Caml: Concurrent ML (INRIA, France)
- *OCAML* (Objective Caml -INRIA): ML extended with OO and a module system
  - First language that combines full power of OOP with ML-style static typing and type inference
  - Advanced OO programming idioms: type-parametric classes, binary methods, mytype specialization) in a statically type-safe way (see http://caml.inria.fr/about/history.en.html)

SML

- http://www.smlnj.org
- In the practical part of the course we will use **Standard ML of New Jersey** (SML/NJ, v110.67)
  - From the prompt: sml

```
stolz ~ $ sml
Standard ML of New Jersey v110.76 [built: Tue Oct 29 11:16:33 2013]
-
```

  - See Pucella 1.6. ”Getting started”
  - Note: to read in a file with sml code
    - use ”filename.sml”;

Core ML

- Basic Types
  - Unit (unit)
  - Booleans (bool)
  - Integers (int)
  - Strings (string)
  - Characters (char)
  - Reals (real)
  - Tuples
  - Lists
  - Records
- Patterns
- Declarations
- Functions
- Type declarations
- Reference Cells
- Polymorphism
- Overloading
- Exceptions

Basic Overview of ML

- SML has an Interactive compiler: *read-eval-print*
  - Expressions are type checked, compiled and executed
  - Compiler infers type before compiling or executing
- Examples

```
- (5+3)-2;
> val it = 6 : int        "it" is an id bound to the value of last exp
- if 5>3 then "Big" else "Small";
> val it = "Big" : string
- val greeting = "Hello";
> val greeting = "Hello" : string
```

INF 3110 - 2016

Overview by Type

- Booleans
  - true, false : bool
  - if … then … else … types must match; “else” is mandatory
- Integers
  - 0, 1, 2, … -1, -2, … : int .
  - +, -, * , div … : int * int → int .
  - =,<,<=,>,<= : int * int -> bool .
  - (op >) turns the infix operator > into a function: 1 < 5 but (op <)(1,5)
- Strings
  - “Universitetet i Oslo” : string
  - “Universitetet” ^ “ i ” ^ “Oslo”
- Char
  - #”a”
- Reals
  - 1.0, 2.2, 3.14159, … decimal point used to disambiguate
  - No ‘=‘ operator for reals 1.0 = 1.0 → Error
  - Cannot combine reals and ints, no coercion. 1.0 + 2 → Error

Compound Types

- Unit
  - () : unit similar to void in C
- Tuples
  - (1 , 2) : int * int ;
  - (4, 5, "ha det!") : int * int * string;
  - #3(4, 5, "ha det!")

    > val it = "ha det" : string
- Records
  - Are tuples with labeled fields:
  - {name="Jones", age=34}: {name: string, age: int};
  - #name({name="Jones", age=34}); > val it = "Jones" : string
  - **Order does not matter:**

    {name="Jones", age=34} = {age=34, name="Jones"}; → true

    ("Jones",34) = (34,"Jones") → Error.
- Lists
  - nil;
  - 1 :: nil ;
  - 1::(2::(3::(4::nil)))
  - 1 :: [2, 3, 4]; infix cons notation

    > val it = [1,2,3,4] : int list
  - [1,2] @ [3,4] append

    > val it = [1,2,3,4] : int list

INF 3110 - 2016

Value declarations and patterns

- val keyword, type annotations
  - val mypi = 3.1415; > val mypi = 3.1415 : real
  - val name : string = "Gerardo"; > val name = "Gerardo" : string
- Patterns can be used in place of identifiers (more later)

  <pat> ::= <id> | <tuple> | <cons> | <record>| <constr>
- Value declarations
  - General form : val <pat> = <exp>
  - Examples:

    - val myTuple = ("Carlos", "Johan");

    - val (x,y) = myTuple;

    - val myList = [1, 2, 3, 4];

    - val x::rest = myList;
  - Local declarations

    let val x = 2+3 in x*4 end;

    > val it = 20 : int

INF 3110 - 2016

Functions and Pattern Matching

- Function declaration
  - Functions are as other values:
    - (5*6) ;

      > val it = 30 : int
    - fn x => x * 2 ; "anonymous function", in lambda notation λx . (x * 2)

      > val it = fn : int -> int
    - val dbl = fn x => x * 2 ; > val dbl = fn : int -> int
  - But we have a special syntax for defining functions:
    - fun dbl x = x * 2 ; > val dbl = fn : int -> int
- Function declaration, general form
  - fun f (<pattern>) = <expr>
    - fun f (x,y) = x+y; Actual par. must match pattern (x,y)
  - fn <pattern> => <expr>
    - fn (x,y) => x+y; Anonymous function
- Multiple-clause definition
  - fun <name> <pat$_1$> = <exp$_1$> | …

    | <name> <pat$_n$> = <exp$_n$>
  - fun length (nil) = 0

    | length (x::s) = 1 + length(s);

  > val length = fn ′a list -> int
  - length ["J", "o", "n"] > val it = 3 : int

Some functions on lists

- Insert an element in an ordered list

```
fun insert (e, nil)      = [e]
 |   insert (e, x::xs) = if e>x then x :: insert(e,xs)
                                  else e::(x::xs);
- insert (3,[1,2,5]) ;
> val it = [1,2,3,5] : int list
```

- Append lists

```
fun append(nil, ys) = ys
|    append(x::xs, ys) = x :: append(xs, ys);
- append ([3,4],[1,2]) ;
>val it = [3,4,1,2] : int list
```

Data-type declarations

- Enumeration types
  - datatype color = Red | Yellow | Blue;
    - elements are: Red, Yellow, Blue   <- Constructors!
- Tagged union types
  - datatype value = I of int | R of real | S of string;
    - elements are: I(9) , R(8.3) , S("hello") ...
  - datatype keyval = StrVal of string * string | IntVal of string * int ;
    - elements are: StrVal("foo","bar") , IntVal("foo",55) ...
  - datatype mylist = Nil | Cons of value * mylist
    - elements are: Nil , Cons (I(8) ,Nil) , Cons (R(1.0), Cons (I (8), Nil))
- General form

  datatype <name> = <clause> | … | <clause>

  <clause> ::= <constructor> | <constructor> of <type>

INF 3110 - 2016

Type abbreviations

- We use *datatype* to define new types
- The keyword *type* can be used to define a type *abbreviation:*

```
- type int_pair = int * int ;
```

- The type inference will not report types as the defined abbrev.:

```
- val a = (3,5);
> val a = (3,5) : int * int
```

- We can force the use of type abbreviation:

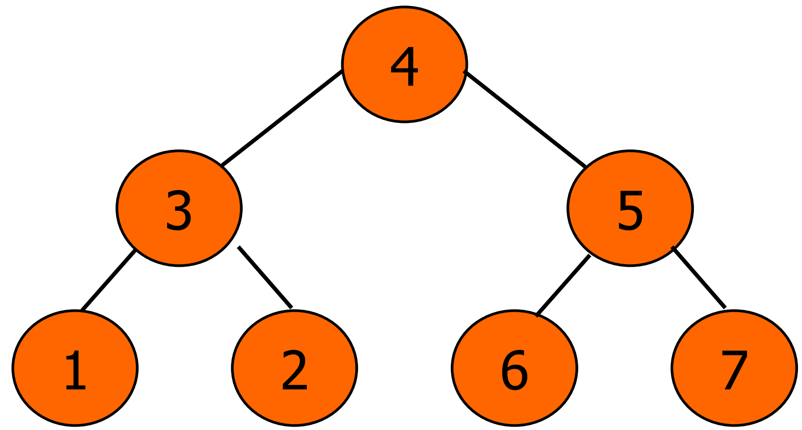
```
- val a : int_pair = (3,5);
> val a = (3,5) : int_pair
```

INF 3110 - 2016

Datatype and pattern matching

- Recursively defined data structure
  - datatype tree = Leaf of int | Node of int*tree*tree;

```
Node(4, Node(3,Leaf(1), Leaf(2)),
        Node(5,Leaf(6), Leaf(7))
    )
```



- Recursive function (sum)
  - fun sum (Leaf n) = n
    | sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2);

Case expression

- Datatype
  - datatype exp = Num of int | Var of var | Plus of exp*exp;
- Case expression

```
case e of Num(i)        =>  …  |
          Var(v)        =>  ….  |
          Plus(e1,e2) => …
- fun eval(e) = case e of  Num(i) => i
                         | Var(v) => lookUp(v)
                         | Plus(e1,e2) => eval(e1) + eval(e2)
```

- Case matching is done in order
- Use _ to catch all missing

```
- fun bintoString(i) = case x of 0 => "zero"
                            |    1 = > "one"
                            |    _ => "illegal value";
> val bintoString = fn : int -> string
```

- Can also use _ in declarations if we don't care about the value being matched
  - fun hd(x::xs) = x ;
  - fun hd(x::_) = x ;

INF 3110 - 2016

insert: Three ”different” declarations

1. ```
   fun insert (e, ls) =
       case ls of nil  => [e]
                | x::xs => if e>x then x::insert(e, xs) else e::ls ;
   ```

2. ```
   fun insert (e, nil)     = [e]
     | insert (e, x::xs) = if e>x then x::insert(e, xs)
                                  else e::(x::xs) ;
   ```

3. ```
   fun insert (e: int, ls: int list) : int list =
       case ls of nil   => [e]
                 | x::xs => if e>x then x::insert(e, xs) else e::ls ;
   ```

INF 3110 - 2016
   ```

ML imperative constructs

- None of the constructs seen so far have side effects
  - An expression has a value, but evaluating it does not change the value of any other expression
- Assignment
  - Different from other programming languages:

    To separate side effects from pure expressions as much as possible

  - Restricted to *reference cells*

INF 3110 - 2016

Variables and assignment

- General terminology: L-values and R-values
  - Assignment (pseudocode, not ML!) `y := x+3;`
    - Identifier on left refers to a *memory location*, called L-value
    - Identifier on right refers to *contents*, called R-value
- Variables
  - Most languages
    - A variable names a storage location
    - Contents of location can be read, can be changed
  - ML reference cell (L-value)
    - A reference cell has a different type than a value
    - Explicit operations to read contents or change contents
    - Separates naming (declaration of identifiers) from "variables"

ML reference cells

- Different types for location and contents

  x : int non-assignable integer value

  y : int ref location whose contents must be integer

- Operations

  ref x expression creating new cell containing value x

  !y returns the contents (value) of location y

  y := x places value x in reference cell y

- Examples

  - val x = ref 0 ; create cell x with initial value 0

  > val x = ref 0 : int ref

  - x := x+3; place value of x+3 in cell x; requires x:int

  > val it = () : unit (type is "unit" since it is an expression with side effects)

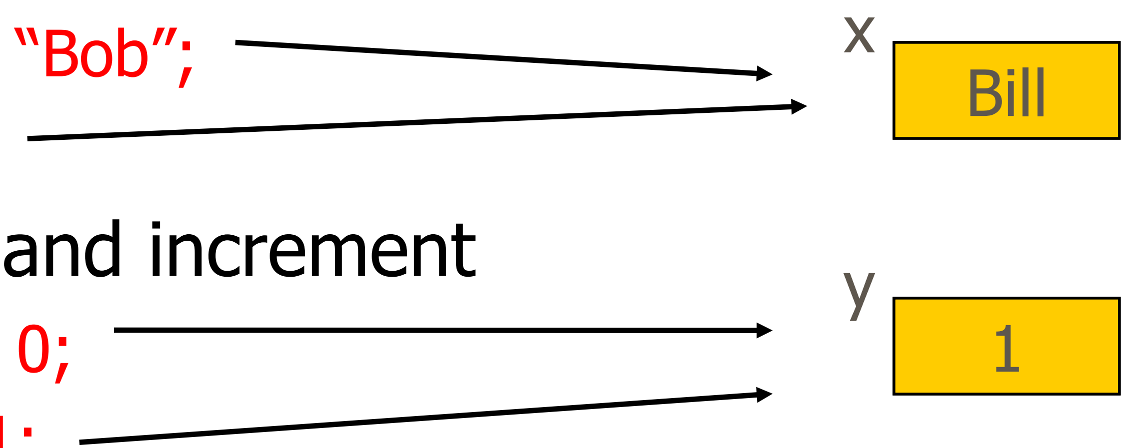
  - x := !x + 3; add 3 to contents of x and store result in location x

  > val it = () : unit

  - !x; > val it = 6 : int

ML examples

- Create cell and change contents
  - val x = ref "Bob";
  - x := "Bill";



- Create cell and increment
  - val y = ref 0;
  - y := !y + 1;
  - y := y + 1 Error!
- In summary:
  - x : int not assignable (like constant in other PL)
  - y : int ref assignable reference cell

Further reading

- Extra material on ML.
- See links on the course page:" Syllabus/achievement requirements "
  - Riccardo Pucella: *Notes on programming SML/NJ (Pensum/Syllabus :Secs. 1.1-1.3, 1.6, and sec. 2.)*
  - In Norwegian: Bjørn Kristoffersen: *Funksjonell programmering i standard ML; kompendium 61*, 1995.
  - **SML/NJ** http://www.smlnj.org/
  - Functions and types available at the top-level: http://www.smlnj.org/doc/basis/pages/top-level-chapter.html
- L.C. Paulson: *ML for the working programmer*

INF 3110 - 2016

ML lectures

- **15.09: The Algol Family and ML (Mitchell's chap. 5 + more)**
- 22.09: More on ML & Types (chap. 5 and 6)
- 13.10: More on Types, Type Inference and Polymorphism (chap. 6)
- ??.??: Control in sequential languages, Exceptions and Continuations (chap. 8)
- ??.??: Prolog I
- ??.??: Prolog ||