The Algol family and ML

Arild B. Torjusen
aribraat@ifi.uio.no

Department of Informatics – University of Oslo

**Based on John C. Mitchell's slides (Stanford U.) ,
adapted by Gerardo Schneider, UiO.**

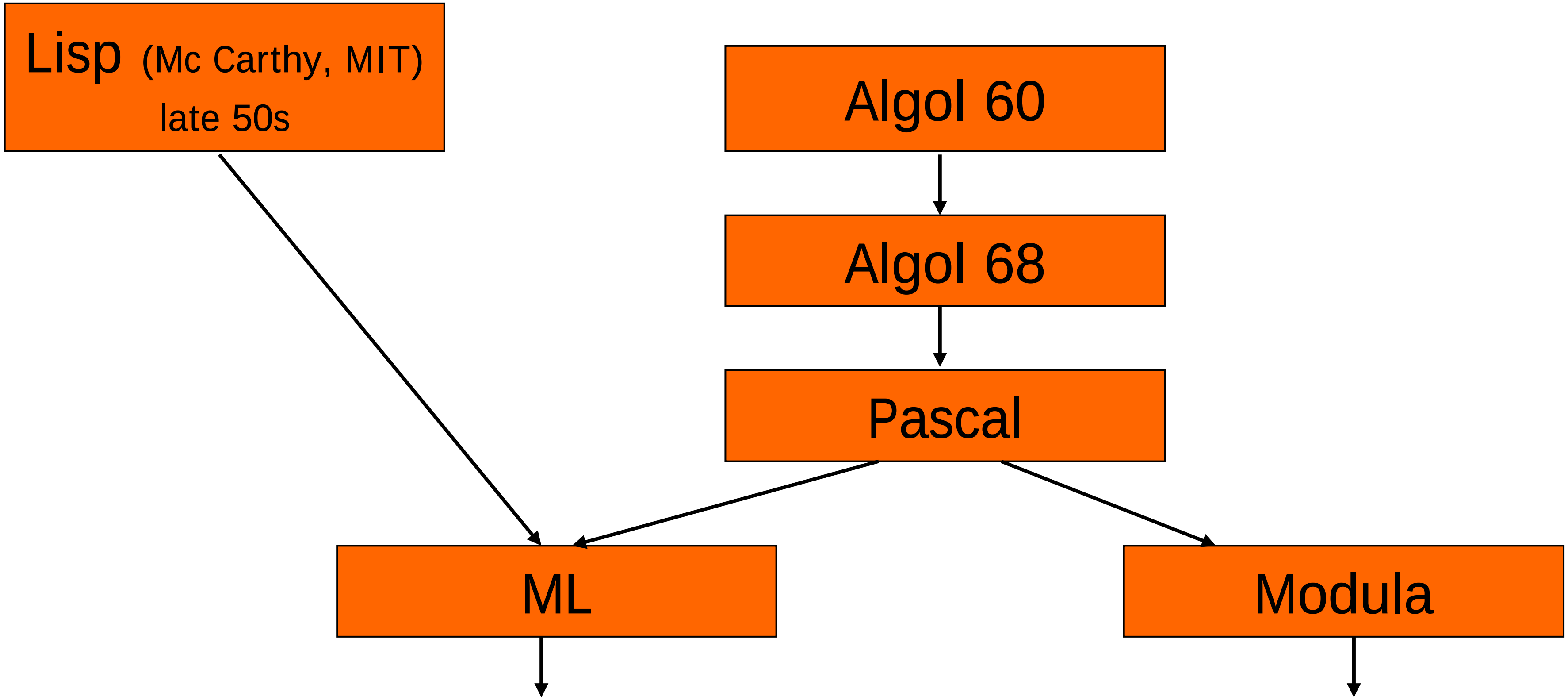
ML lectures

1. **04.09: The Algol Family and ML (Mitchell's chap. 5 + more)**
2. 11.09: More on ML & types (chap. 5 and 6)
3. 18.09: More on Types, Type Inference and Polymorphism (chap. 6)
4. 02.10: Control in sequential languages, Exceptions and Continuations (chap. 8)

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 – see at the end)

A (partial) Language Sequence



Lisp (Mc Carthy, MIT)
late 50s

Algol 60

Algol 68

Pascal

ML

Modula

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. Mc Carthy, A. Perlis,...)
- General purpose language. Features:
  - Simple imperative language + functions
  - Successful syntax, BNF -- used by many successors
    - statement oriented
    - Begin ... End blocks (like C { ... }) (local variables)
    - if ... then ... else
  - 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;
   begin
        real sum; sum := 0;
        for i = 1 step 1 until n do
                sum := sum + A[i];
        average := sum/n
   end;
```

no array bounds

no “;” here

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
- 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
- Fixed some problems of Algol 60
  - Eliminated pass-by-name (introduced pass-by-reference)
- Storage management
  - Local storage on stack
  - Heap storage, explicit alloc and garbage collection

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 (till the 90s)
- Simple one-pass compiler

Limitations of Pascal

◆ Array bounds part of type

illegal

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

- 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

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 (1969 - 1973)
- 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 ptr 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 (?!)

ML

- A *function-oriented imperative language*
- Typed programming language (sound)
- Intended for interactive use
- 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
- *OCAML*: ML extended with OO and a sophisticated module system

Why study ML ?

- Learn an important language that’s different
- Discuss general programming languages 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

Why study ML ?

- Learn to think about, and solve problems in new ways
- All programming languages has a functional “part”. Useful to know.
- Verifiable programming: Easier to reason about a functional language.
- More compact (simple?) code. Higher order functions.
- Certain aspects are easier to understand and program. E.g. recursion.

History of ML

- Designed by Robin Milner – part of the LCF project
- Logic for Computable Functions (LCF project)
  - Based on Dana Scott’s ideas (1969)
    - Formulate logic for proving properties of typed functional programs
  - Milner
    - Project to automate logic
    - Notation for programs
    - Notation for assertions and proofs
    - Need to 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

$$\text{tactic(formula)} = \begin{cases} \text{succeed and return proof} \\ \text{search forever} \\ \text{fail} \end{cases}$$

- 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”

$$\text{tactic(formula)} = \begin{cases} \text{succeed and return proof} \\ \text{search forever} \\ \text{fail and } \textit{raise exception} \end{cases}$$

- First type-safe exception mechanism!

Function types in ML

$f : A \to B$ means

for every $x \in A$,

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

SML

- In the practical part of the course we will use **Standard ML** of New Jersey (SML/NJ, v110.49)
  - From the prompt: sml
- Assistants:
  - John Olav Lund (Gr 3)
  - Marius Einan Storeide (Gr 1 & 2)
- Mandatory exercise (”oblig”) – next monday on the course homepage

Core ML

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

Basic Overview of ML

- 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
```

Overview by Type

- Booleans
  - true, false : bool
  - if ... then ... else ... types must match; "else" is mandatory
- Integers
  - 0, 1, 2, ... -1, -2, ... : int
  - +, * , ... : int * int → int
- Strings
  - "Universitetet i Oslo" : string
  - "Universitetet" ^ " i " ^ "Oslo"
- Reals
  - 1.0, 2.2, 3.14159, ... decimal point used to disambiguate

Compound Types

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

    > val it = "ha det" : string
- Records
  - {name=“Anibal”, hungry=true}: {name: string, hungry: bool};
  - #name({name = “Anibal”, hungry=true});

    > val it = “Anibal” : string
- 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

Patterns and Declarations

- Patterns can be used in place of identifiers

  `<pat> ::= <id> | <tuple> | <cons> | <record> ...`

- 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;
    ```

Functions and Pattern Matching

◆ Function declaration

- fun f(<pattern>) = <expr>
  - fun f (x,y) = x+y;   actual par. must match pattern (x,y)
  - fun f x y = x+y;
- fn <pattern> => <expr>
  - fn x => x+1; anonymous function (like Lisp “lambda”)
  - val addone = fn x => x+1 ;

◆ Multiple-clause definition

- fun <name> <pat$_1$> = <exp$_1$> | …

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

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

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);
```

- Append lists

```
fun append(nil, ys) = ys
|    append(x::xs, ys) = x :: append(xs, ys);
```

Map function on lists

- Apply a function to every element of list

```
fun map (f, nil) = nil
|     map (f, x::xs) = f(x) :: map (f,xs);

 fun incr x = x+1 ;

 map (incr, [1,2,3]);             ➡ [2,3,4]

 map (fn x => x*x, [1,2,3]);           ➡ [1,4,9]
```

Map is a ***high-order*** function (or a ***functional***)

Data-type Declarations

- General form

  datatype <name> = <clause> | ... | <clause>

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

- Examples
  - datatype color = red | yellow | blue;
    - elements are: *red*, *yellow*, *blue*
  - datatype atom = atm of string | nmbr of int;
    - elements are: *atm("A")*, *atm("B")*, ..., *nmbr(0)*, *nmbr(1)*, ...
  - datatype list = nil | cons of atom*list;
    - elements are: *nil*, *cons(atm("A"), nil)*, ...

      cons(nmbr(2), cons(atm("ugh"), nil)), ...

Type Abbreviations

- The keyword *type* can be used to define a type *abbreviation:*
  - type int_pair = int * int ;
  - (1,2) : int_pair ;
  - type person = {name : string, age : int }

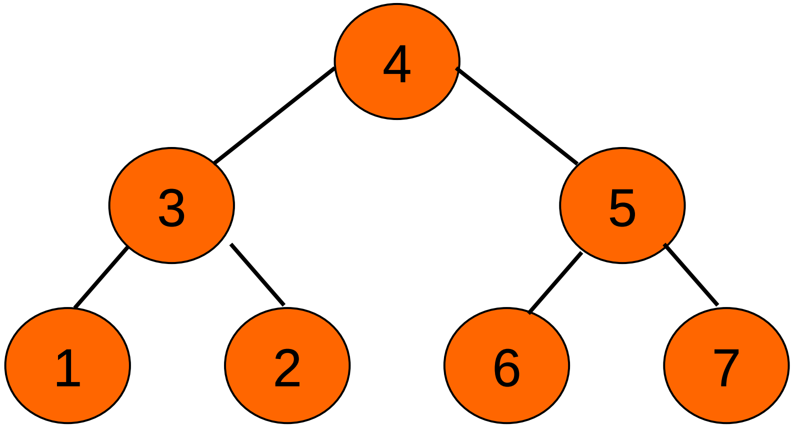
```
- fun getName(x) = #name(x)
fun getName (x) = #name(x);
stdIn:1.1-38.6 Error: unresolved flex record
   (can't tell what fields there are besides #name)
fun getName(x : person) = #name(x)
fun getName(x) = #name(x)
```

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

```
fun sum (Leaf n) = n
  | sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2);
```

Case expression

- Datatype

```
datatype exp = Var of int | Const of int | Plus of exp*exp;
```

- Case expression

```
case e of  Var(n)       =>  ...   |
           Const(n)     => ....  |
           Plus(e1,e2) => ...
fun eval(e) =
    case e of  Var(n)    => n
            | Const(n)  => n
            | Plus(e1,e2) => eval(e1) + eval(e2) ;
```

insert: Three ”different” declarations

1.
```
fun insert (e,nil)     = [e]
   | insert (e, x::xs) = if e>x then x::insert(e,xs)
                                  else e::(x::xs);
```
2.
```
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;
```
3.
```
fun insert (e,ls) =
    case ls of nil  => [e]
               | x::xs => if e>x then x::insert(e,xs) else e::ls;
```

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*

Variables and assignment

- General terminology: L-values and R-values
  - Assignment `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 mutable cell is another type of 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 initialized to x
!y	the contents of location y
y := x	places value x in reference cell y

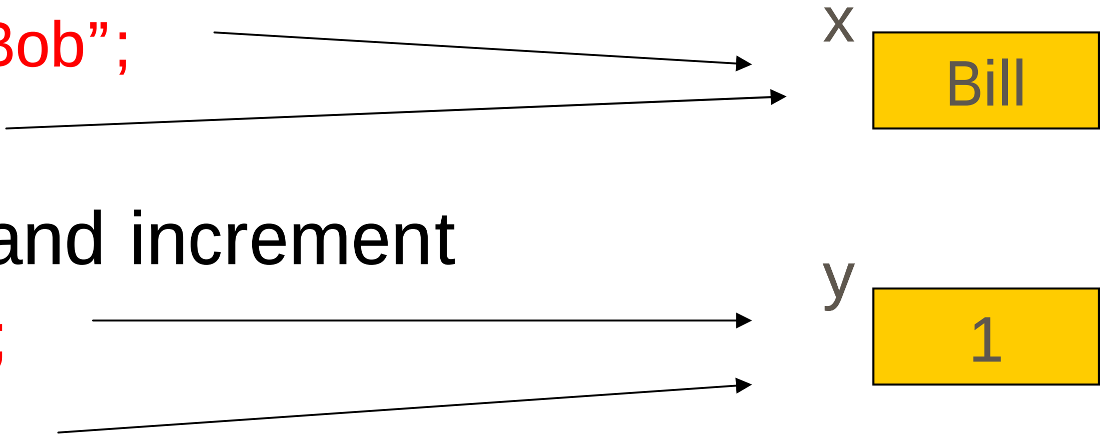
- Examples

`val y = ref 0 ;`	create cell y with initial value 0
`y := x+3;`	place value of x+3 in cell y; requires x:int
`y := !y + 3;`	add 3 to contents of y and store in location y

ML examples

- Create cell and change contents

```
val x = ref "Bob";
x := "Bill";
```



- Create cell and increment

```
val y = ref 0;
y := !y + 1;
```

- While loop

```
val i = ref 0;
while !i < 10 do i := !i +1;
!i;
```

Further reading

- Extra material on ML.
- See links on the course page:”Pensum/læringskrav”
  - Bjørn Kristoffersen: *Funksjonell programmering i standard ML; kompendium 61*, 1995. Pensum!
  - Riccardo Pucella: *Notes on programming SML/NJ*
- L.C. Paulson: *ML for the working programmer*

ML lectures

1. 04.09: The Algol Family and ML (Mitchell's chap. 5 + more)
2. **11.09: More on ML & types (chap. 5 and 6)**
3. 18.09: More on Types, Type Inference and Polymorphism (chap. 6)
4. 02.10: Control in sequential languages, Exceptions and Continuations (chap. 8)