ML

CSCI 334
Stephen Freund

Algol 60
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;

Language History

ML

- Combination of Lisp and Algol-like features
- Expression-oriented
- Higher-order functions
- Garbage collection
- Static types
- Abstract data types
- Module system
- Exceptions
Goals in study of ML

- Types, type checking, polymorphism
- Memory management
- Control Structures

Robin Milner and ML's Origins

- Dana Scott, 1969
  - LCF
  - Logic for stating theorems about programs
- Robin Milner
  - Automated theorem proving for LCF
  - Hard search problem
  - Incomplete: may not find proof
  - ML: meta-language for writing programs (tactics) to find proofs

Tactics

- Tactics guide search in theorem prover
  - "Try induction to prove T"
  - "Assume X and derive contradiction"
  - "Try A and then B"

- Tactic is partial function from formula -> proof
  - finds proof
  - never terminates
  - reports an error

Language Ideas to Support Tactics

- Type system
  - guarantees correctness of generated proof
- Exception handling
  - deals with tactics that fail (Turing Award)
- Higher-order functions
  - composition of tactics
  - fun compose(t1, t2) = 
    \( \lambda \text{formula. if t1(formula) then ... else if t2(formula) ...} \)
Running ML
- Type `sml` on Unix machines
- System will give you prompt
- Enter expression or declarations to evaluate:
  - `3 + 5;`
  - `val it = 8 : int`
  - `it * 2;`
  - `val it = 16 : int`
  - `val six = 3 + 3;`
  - `val six = 6 : int`
- Or "sml < file.ml"

Defining Functions
- Example
  - `fun succ x = x + 1;`
  - `val succ = fn : int -> int`
  - `succ 12;`
  - `val it = 13 : int`
  - `17 * (succ 3);`
  - `val it = 68 : int;`
- Or:
  - `val succ = fn x => x + 1;`
  - `val succ = fn : int -> int`
Recursion

• All functions written using recursion and if.. then.. else (and patterns):
  - fun fact n =
    if n = 0 then 1 else n * fact (n-1);

• if..then..else is an expression:
  - if 3<4 then "moo" else "cow";
  val it = "moo" : string
  - types of branches must match

Local Declarations

- fun cylinderVolume diameter height =
  let val radius = diameter / 2.0;
  fun square y = y * y
  in
    3.14 * square(radius) * height
  end;

val cylinderVolume = fn : real -> real -> real
- cylinderVolume 6.0 6.0;
val it = 169.56 : real

Built-in Data Types

• unit
  - only value is ()

• bool
  - true, false
  - operators not, andalso, orelse

• int
  - ..., -2, -1, 0, 1, 2, ...
  - +,-,*,div,mod,abs
  - =,<,<=,etc.

• real
  - 3.17, 2.2, ...
  - +, -, *, /
  - <, <=, etc.
  - no conversions from int to real: 2 + 3.3 is bad
  - no equality (test that -0.001 < x-y < 0.001, etc.)

• strings
  - "moo"
  - "moo" ^ "cow"
**Overloaded Operators**

- +,-, etc. defined on both int and real
- Which one to use depends on operands:
  - fun succ x = x + 1
    val succ = fn : int -> int
  - fun double x = x * 2.0
    val double = fn : real -> real
  - fun double x = x + x
    val double = fn : int -> int

**Type Declarations**

- Can add types when type inference does not work
  - fun double (x:real) = x + x;
    val double = fn : real -> real
  - fun double (x:real) : real = x + x;
    val double = fn : real -> real

**Compound Types**

- Tuples, Records, Lists
- Tuples
  (14, "moo", true): int * string * bool
- Functions can take tuple argument
  - fun power (exp,base) =
    if exp = 0 then 1
    else base * power(exp-1,base);
    val power = fn : int * int -> int
  - power(3,2);

**Curried Functions (named after Curry)**

- Previous power
  - fun power (exp,base) =
    if exp = 0 then 1
    else base * power(exp-1,base);
    val power = fn : int * int -> int
- Curried power function
  - fun cpower exp =
    fn base =>
    if exp = 0 then 1
    else base * cpower (exp-1) base;
    val cpower = fn : int -> (int -> int)
Curried Functions (named after Curry)

- Previous power
  - fun power (exp, base) =
    if exp = 0 then 1
    else base * power(exp-1, base);
  val power = fn : int * int -> int

- Curried power function
  - fun cpower exp base =
    if exp = 0 then 1
    else base * cpower (exp-1) base;
  val cpower = fn : int -> (int -> int)

Curried Functions

- Why is this useful?
  - fun cpower exp base =
    if exp = 0 then 1
    else base * cpower (exp-1) base;
  val cpower = fn : int -> (int -> int)

- Can define
  - val square = cpower 2
  val square = fn : int -> int
  - square 3;
  val it = 9 : int

Records

- Like tuple, but with labeled elements:
  { name="Gus", salary=3.33, id=11 }:
    { name:string, salary:real, id:int };

- Selector operator:
  - val x =
    { name="Gus", salary=3.33, id=11 };
  - #salary(x);
  val it = 3.33 : real
  - #name(x);
  val it = "Gus" : string
Lists

• Examples
  - [1, 2, 3, 4], ["wombat", "numbat"]
  - nil is empty list (sometimes written [])
  - all elements must be same type

• Operations
  - length  length [1,2,3] ⇒ 3
  - @ - append  [1,2]@[3,4] ⇒ [1, 2, 3, 4]
  - :: - prefix  1::[2,3] ⇒ [1, 2, 3]
  - map  map succ [1,2,3] ⇒ [2,3,4]

Functions on Lists

• fun product (nums) =
  if (nums = nil)
  then 1
  else (hd nums) * product(tl nums);

val product = fn : int list -> int

- product([5, 2, 3]);
  val it = 30 : int;

Pattern Matching

• List is one of two things:
  - nil
  - "first elem" :: "rest of elems"
  - [1, 2, 3] = 1::[2,3] = 1::2::[3] = 1::2::3::nil

• Can define function by cases

  fun product (nil) = 1
  | product (x::xs) = x * product (xs);

Patterns on Integers

• Patterns on integers
  fun listInts 0 = [0]
  | listInts n = n::listInts(n-1);

  listInts 3 ⇒ [3, 2, 1, 0];

• More on patterns for other data types next time
Many Types Of Lists

- `1::2::nil : int list`
- "wombat"::"numbat"::nil : string list
- What type of list is `nil`?
  - nil;
  - val it = [] : 'a list
- Polymorphic type
  - 'a is a type variable that represents any type
  - `1::nil : int list`
  - "a"::nil : string list

The Length Function

- Another Example

  ```ml
  fun length (nil) = 0
  | length (x::xs) = 1 + length (xs);
  ```

  - What is the type of `length`?
  - How about this one:

    ```ml
    fun id x = x;
    ```

Polymorphism

```ml
fun length (nil) = 0
| length (x::xs) = 1 + length (xs);
val it = fun 'a list -> int

fun id x = x;
val it = fun 'a -> 'a
```

Patterns and Other Declarations

- Patterns can be used in place of variables
- Most basic pattern form
  - val <pattern> = <exp>;
- Examples
  - val x = 3;
  - val tuple = ("moo", "cow");
  - val (x,y) = tuple;
  - val myList = [1, 2, 3];
  - val w::rest = myList;
  - val v::_ = myList;

Type variable represents any type
Datatype
public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int EAST = 3;
public static final int WEST = 4;

public move(int x, int y, int dir) {
    switch (dir) {
        case NORTH: ...
        case ...
    }
}

Datatype
datatype Direction =
    NORTH | SOUTH | EAST | WEST;

fun move((x,y),NORTH) = (x,y-1)
    | move((x,y),SOUTH) = (x,y+1)
    ...
;


History of ML

• Read
  - Sections 5.1-5.2
  - Read Kernighan, "Why Pascal is Not My Favorite Programming Language" (on web)

• General themes
  - Languages get better over time
  - Bizarre things have been tried (call by name)
  - What was significant about Algol 60?
  - Why did Algol 68 fail?
  - What is Pascal good for? bad for? Why?
**More Pattern Matching**

- Patterns on integers
  
  ```haskell
  fun listInts 0 = [0]
  | listInts n = n::listInts(n-1);
  
  listInts 3 ⇒ [3, 2, 1, 0];
  ```

- Map
  
  ```haskell
  fun mapl (f,nil) = nil
  | mapl (f,x::xs) = f(x)::mapl(f,xs);
  
  mapl (fn x => x + 1, [3,2,1]) ⇒ [4,3,2]
  ```

**Pattern Details...**

- Patterns must cover all cases
  
  ```haskell
  - fun bad (x::xs) = "moo";
  Warning: match nonexhaustive... 
  ```

- Name can only appear once in a pattern
  
  ```haskell
  - fun bad (x,y,x) = ...;
  Error: duplicate variable in pattern
  ```

**More Patterns**

- fun coords(shape): (int * int) =...

- val (x,y) = coords(rectangle1);

val x = 100;
val y = 200;

- val head::tail = [1,2,3];

val head = 1;
val tail = [2, 3];
Equality Types and Polymorphism

fun search item nil = false
  | search item (x::xs) =
      if item = x then true
      else search item xs;

• Type is `'a -> (''a list -> bool)
• Element x is compared to item with =
  - equality not defined on real or function types
  - search 3.4 [2.2, 3.2] is bad
  - search 1[1, 2, 3, 4] is ok

Patterns and Other Declarations

• Patterns can be used in place of variables
• Most basic pattern form
  - val <pattern> = <exp>;
• Examples
  - val x = 3;
  - val tuple = ("moo", "cow");
  - val (x,y) = tuple;
  - val myList = [1, 2, 3];
  - val w::rest = myList;
  - val v::_ = myList;

Datatype

public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int EAST = 3;
public static final int WEST = 4;

public move(int x, int y, int dir) {
    switch (dir) {
        case NORTH: ...
        case ...
    }
}

Datatype

datatype Direction =
    NORTH | SOUTH | EAST | WEST;

fun move((x,y),NORTH) = (x,y-1)
  | move((x,y),SOUTH) = (x,y+1)
  ...
  ;
Datatype Declarations

• Example
  - datatype Color = Red | Blue | Green;

• General Form
  - datatype <typename> =
    <constructor> | ... | <constructor>;

• Constructors show how to generate elements of the new type
• Pattern matching used to deconstruct values

Colors

• Example
  - datatype Color = Red | Blue | Green;
  - fun toString(Red) = "Red"
    | toString(Blue) = "Blue"
    | toString(Green) = "Green";

  val toString = fn : Color -> string;

Tagged Union

- datatype Student =
  UGrad of string
  | Grad of string * School;

- fun getName(UGrad(n)) = n
  | getName(Grad(n,s)) = n;
val getName = fn : Student -> string

- val t = Grad("Bob", Williams);
val it = "Bob";

Recursive Datatypes

- datatype Tree = LEAF of int
  | NODE of Tree * Tree;

NODE(LEAF(3), NODE(LEAF(11), LEAF(5)))

- fun inTree (n, LEAF(x)) = (x = n)
  | inTree (n, NODE(l, r)) =
    inTree(n, l) inTree find(n, r);

3 11 5
Expression Trees

datatype NumExpr =
  Val of int
  | Plus of NumExpr * NumExpr
  | Mult of NumExpr * NumExpr;

fun eval (Val(x)) = x
  | eval(Plus(left,right)) =
    eval(left) + eval(right)
  | ...;

Polymorphic Recursive Datatypes

- datatype 'a Tree = LEAF of 'a
  | NODE of 'a Tree * 'a Tree;

NODE(LEAF(3), NODE(LEAF(11), LEAF(5)))
NODE(LEAF("A"), NODE(LEAF("B"), LEAF("C")))

- fun inTree (n, LEAF(x)) = (x = n)
  | inTree (n, NODE(l, r)) =
    inTree(n, l) or else inTree(n, r);

Summary So Far

• Types
  - unit, bool, int, real, string
  - tuples, records, lists

• Features
  - val/fun declarations
  - overloading
  - polymorphism
  - patterns
  - data types

Imperative Features

• Example
  int x,y;
  x = y + 3;

• Value of y is added to 3 and stored in Location for x

• Location of variable is its L-value
• Value of variable is its R-value
Reference Cells

- In ML, L-values and R-values have different types
  - l-values are reference cells
  - special operations to access/update cells
- Operations on reference cells
  - \( \text{ref v} \) create a new cell containing \( v \)
  - \( !r \) the value stored in \( r \)
  - \( r := v \) store \( v \) in cell \( r \)

Examples

- \( \text{val x = ref 1;} \)
  \( \text{val x = ref 1 : int ref;} \)
- \( x := 3 * (!x) + 2; \)
  \( \text{val it = (): unit} \)
- \( !x; \)
  \( \text{val it = 8;} \)

Loops

\( \text{val i = ref 0;} \)
\( \text{val x = ref 0;} \)
\( \text{while (!i < 10) do} \)
  \( \text{(i := !i + 1; x := !x + !i);} \)
  \( !x; \)

EXTRA SLIDES
Contains

fun contains(x, nil) = nil
  | contains(x, y::rest) =
    if (x = y) then true
    else contains(x,rest);

Datatype

public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int EAST = 3;
public static final int WEST = 4;

public move(int x, int y, int dir) {
  switch (dir) {
    case NORTH: ...
    case ...
  }
}

Datatype

datatype Direction =
  NORTH | SOUTH | EAST | WEST;

fun move((x,y),NORTH) = (x,y-1)
  | move((x,y),SOUTH) = (x,y+1)
  ...

Constructors

datatype Person =
  Student of string * int
  | Prof of string list * string;

val steve = Prof(["134","334"],"Steve");

fun isProf(Prof(_,_)) = true
  | isProf(_) = false;

fun getName(Student(n,_)) = n
  | getName(Prof(_,n)) = n;
Expression Trees

datatype NumExpr =
  Val of int
  | Plus of NumExpr * NumExpr
  | Mult of NumExpr * NumExpr;

fun eval(Val(x)) = x
  | eval(Plus(left,right)) =
    eval(left) + eval(right)
  | ...;

Algol 60

• Basic Language of 1960
  - Simple imperative language + functions
  - Successful syntax, BNF -- used by many successors
    • statement oriented
    • Begin ... End blocks (like C { ... })
    • if ... then ... else
  - Recursive functions and stack storage allocation
  - Fewer ad hoc restrictions than Fortran
    • General array references: A[x + 8[3]*y]
  - Type discipline was improved by later languages
  - Very influential but not widely used in US

Algol Joke

• Question
  - Is x := x equivalent to doing nothing?
• Interesting answer in Algol
  
    integer procedure p;
    begin
      ...
      p := p
      ...
    end;

• Assignment here is actually a recursive call
Some trouble spots in Algol 60

- Type discipline improved by later languages
  - Limited array bounds checks
  - higher-order functions (see hw)

- Parameter passing methods

- (awkward control issues...)

Algol 60 Pass-by-name

- Substitute text of actual parameter
  - Unpredictable with side effects!

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

  Is this what you expected?

Pascal

- Revised type system of Algol
  - Good data-structuring concepts
    - records, variants, subranges (ie, [1..10])
  - Popular teaching language

- Array bounds part of type

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

illegal

Algol 60

- Basic Language of 1960
  - Simple imperative language + functions
    - formal grammars
    - recursion
    - statement oriented
    - Begin ... End blocks (like C {...})
    - if ... then ... else
  - Fewer ad hoc restrictions than Fortran
    - General array references: A[x + B[3]*y]
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;

Some trouble spots in Algol 60

- Type checking problems
  - bounds checking for arrays
- problems with functions passed to other functions
- Parameter passing methods
  - Pass-by-value (like Java/C)
  - Pass-by-name (in book)

Algol 60 Pass-by-name

- Substitute text of actual parameter
  - 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

- Considered difficult to understand
  - Idiosyncratic terminology
  - Complicated grammar
  - Overly complicated type system
- Fixed some problems of Algol 60
  - Eliminated pass-by-name
- Hard to compile efficiently

Pascal

- Revised type system of Algol
  - Good data-structuring concepts
    - records, variants, subranges (seen in ML)
  - More restrictive than Algol 60/68
- Popular teaching language
- Simple one-pass compiler

Limitations of Pascal

- Array bounds part of type
  
  procedure p(x: array [1..10] of integer);
  procedure p(x: integer, a: array [1..n] of integer)

  illegal

- Not successful for “industrial-strength” projects
  - Kernighan -- Why Pascal is not my favorite language
  - Left niche for C; niche has expanded!!

Pattern Matching

- All lists are built with :: and nil
  - [1, 2, 3] = 1::[2,3] = 1::2::[3] = 1::2::3::nil

- Can define function by cases
  
  fun length nil = 0
  | length (x::xs) = 1 + length xs;

  fun product nil = 1
  | product (x::xs) = x * product xs;
Many Types Of Lists

- \(1::2::\text{nil} : \text{int list}\)
- "wombat"::"numbat"::\text{nil} : \text{string list}\)
- What type of list is \text{nil}?
  - \text{nil};
  - \text{val it} = [] : 'a list
- Polymorphic type
  - 'a is a type variable that represents any type
  - \(1::\text{nil} : \text{int list}\)
  - "a"::\text{nil} : \text{string list}
  - Type of length: 'a list -> \text{int}

Other Examples of Polymorphism

fun \text{id}(x) = x: 'a -> 'a

map:('a -> 'b) -> ('a list -> 'b list)

- \text{map} \text{succ} \{1, 2\};
- \{2, 3\} : \text{int list};

- \text{map} \text{size} \{"moo", "wombat"\};
- \{3, 6\} : \text{int list};

Pattern Matching

- Can define function by cases
  - \text{fun} \text{length} x =
    - \text{if} \ (x = \text{nil}) \ \text{then} \ 0
    - \text{else} \ 1 + \text{length} \ (\text{tl} \ x)

- List is one of two things:
  - \text{nil}
  - \text{x :: xs}, where \text{xs} is the "rest of the list"

  - [1, 2, 3] = 1::[2,3] = 1::2::[3] = 1::2::3::nil

Pattern Matching

- Can define function by cases

  - \text{fun} \text{length} (\text{nil}) = 0
    | \text{length} \ (x::xs) = 1 + \text{length} \ (xs);

  - \text{fun} \text{product} (\text{nil}) = 1
    | \text{product} \ (x::xs) = x * \text{product} \ (xs);
Pattern Matching
- All lists are built with :: and nil
  - \([1, 2, 3] = 1::[2,3] = 1::2::[3] = 1::2::3::\text{nil}\)
- Can define function by cases
  - fun length \text{nil} = 0
  - | length \((x::xs)\) = 1 + length \(xs\);
  - fun product \text{nil} = 1
  - | product \((x::xs)\) = \(x \times \text{product} \(xs\);

More Pattern Matching
- Patterns on integers
  - fun listInts 0 = [0]
  - | listInts \(n\) = \(n::\text{listInts}(n-1)\);
  - listInts 3 \(\Rightarrow [3, 2, 1, 0]\);
- Map
  - fun mapl \((f,\text{nil})\) = \text{nil}
  - | mapl \((f,x::xs)\) = \(f(x)::\text{mapl}(f,\text{xs})\);
  - mapl \((\text{fn } x \Rightarrow x + 1, [3,2,1])\) \(\Rightarrow [4,3,2]\)

Logic for Computable Functions (LCF)
- Dana Scott, 1969
  - logical system for stating theorems about programs
- Robin Milner
  - automated theorem proving for LCF
  - program to find proof of theorems
  - theorem proving is a hard search problem

A Recent Implementation...