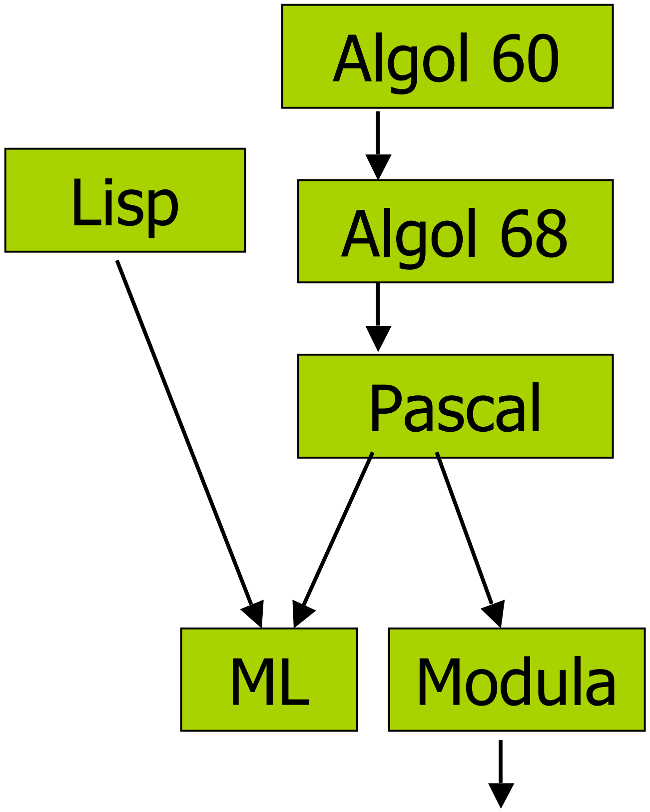
The ML Language

Typed Functional Programming
with Assignments

The Algol Family---Imperative Programming

- Modify variables through statements
  - Block of statements separated by ";"
    Begin … End (Algol, Pascal), { … } in C
  - Conditionals and loops
- Rich and structured type system
  - Basic types: int, char, string, complex, …
  - Compound types: record, struct, union/variant, range, array, pointer,…
- Example languages: Algol, Pascal, C
- ML: typed functional programming
  - Developed by Robin Milner et al.
  - Meta-language for **Logic for Computable Functions.** Compiled and then interpreted
  - Every expression has a single type; expression types checked at compile time



ML: Typed Functional Programming Language

- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
- Sound and expressive type system
  - If a function f has type **A→B**, then for every **x** in **A**,
    - If **f(x)** terminates without raising exceptions, then it has type **B**.
  - Allows parametric types for functions and compound data structures
  - Support union of different types
  - Compiler automatically infers variable types
    - **Type system does not allow casts or other loopholes**

ML Atomic Values(Basic Types)

- Basic types
  - () : unit
  - true/false : bool
  - 3: int
  - “ab” : string
  - 3.0 : real
- Special operations (infix notation)
  - For bool: andalso orelse not
  - For int: + - * div
  - For string: ^ (concatenation)
  - For real: + - * /
- Explicit type conversion
  - real(3) ➔ 3.0 : real

ML Compound Types

- Type parameters: **‘a, ‘b, ‘x, ‘y, ……**
- List: ‘**t1 list**, where ‘t1 is a type
  - Values: **nil : ‘a list, [] : ‘a list, [“a”, “b”] : string list, [7] : int list**
  - Operators: **null** (null?), **hd** (car), **tl** (cdr), **::** (cons)
- Tuple: ‘**t1*‘t2*…** , where ‘t1,‘t2,… are type parameters
  - **(3, 4, “abc”)** : int * int * string
  - Operators: **#2(3, 4, “abc”)** ==> 4 : int
- Record: **{ID1:‘t1,ID2:‘t2,…}**, where ID1,ID2,… are names
  - **{First = 3, Second = “my”}** : {First:int, Second: string}
  - Operators: **#First{First = 3, Second = “my”}** ==> 3 : int
- Reference cell (assignable variable): ‘**t1 ref**
  - **ref 3** : int ref; Operators: **!(ref 3)** ==> 3 : int
- Function abstraction: ‘**t1 -> ‘t2**
  - **fn x => x + 5** : int→int; **fun add5(x) => x + 5** : int→int

ML Union of Different Types

- The datatype declaration (equivalent to union in C)

  datatype <name> = <clause> | … | <clause>
  - Each <clause> is either ID or ID of <type_expression>
  - Can be accessed via pattern matching
- Examples
  - **datatype color = Red | Blue | Green**
    - Elements are **Red, Blue and Green**
  - **datatype tree = LEAF of int | NODE of tree*tree**
    - Values are **LEAF(5), Node(Node(LEAF(2),LEAF(3)),LEAF(5))**
  - **datatype atom = atm of string | nmbr of int**
    - Values are **atm("A"), atm("B"), …, nmbr(0), nmbr(1), …**
  - **datatype list = nil | cons of atom*list**
    - Values are **nil, cons(atm("A"), nil), cons(nmbr(2), cons(atm("ugh"), nil)), …**

ML Patterns

<pattern> ::= <value>
| <var>
| <var> as <pattern>
| (<pattern>,…,<pattern>)
| <pattern>::<pattern>
| {<name>=<pattern>,…,<name>=<pattern>}
| <name>(<pattern>,…,<pattern>)

- Examples of patterns
  - nil, x, (x1,x2,x3), x1::x2,
  - {field1=x1,field2=x2}
  - LEAF(x)
- Used to check structure of compound values
  - Variables are assigned with proper values if matching is successful
  - No variable can occur twice in any pattern

ML Functional Programming Via Patterns

- The Case expression

```
case <exp> of
    <pattern1> => <exp1>
  | <pattern2> => <exp2>
    ......
  | <patternn> => <expn>
```

  - Compare to the cond operator in Scheme
- Variable declaration: `val <pattern> = <exp>;`
- Function Declarations

```
fun <name> <pattern1>  = <exp> ......
  | <name> <pattern> = <expn>;
```

Example --- Appending A List

- In Scheme

```
(define Append (lambda (xs ys)
     (cond ((null? xs) ys)
           ((cons? Xs) (cons (car xs) (Append (cdr xs) ys))))))
```

- In ML

```
fun Append(xs,ys) =
     case (xs) of nil=>ys
                | x1::x2 => x1::Append(x2,ys);
```

or

```
fun Append(xs,ys) =
     if null(xs) then ys else hd(xs)::Append(tl(xs),ys);
```

Or

```
fun Append(nil, ys) = ys
|    Append(x1::x2, ys) = x1 :: Append(x2, ys);
```

- NOTE: all elements in the ML list must have the same type

Example---Tree Search

```
(define Find (lambda (x y)
        (if  (cons? y)
             (or (Find x (car y)) (Find x (cdr y)))
             (eq? x y))))
```

- What types are expected for each variable?
  - x: an atomic type (number, symbol, boolean)
  - y: an atomic type or a possibly nested list of atomic values
- Programming in ML
  - Need to define the types for x and y explicitly

Solution---
Translating Scheme To ML

- Define datatype of expressions

```
datatype ‘label tree =
  Empty | Atom of ‘label
          | Node of ‘label tree * ‘label tree;
```

- Pattern-based evaluation

```
fun Find (x, Empty) = false
 | Find (x, Atom(y)) = x = y
 | Find (x, Node(y1,y2)) =
        Find(x, y1) orelse Find(x, y2);
```

Example---Higher Order Functions

```
(define  maplist  (f  x)
     (cond  ((null? x)   nil)
                  (else  (cons (f (car x))
                                  (maplist f (cdr x))))))
```

- What types are expected for each variable?
  - f: a function mapping atomic values
  - x: a possibly nested list of atomic values

Solution---
Translating Scheme To ML

- Define datatype of expressions

```
datatype 'a tree = Empty | Node of 'a tree * 'a
  tree
```

- Pattern-based evaluation

```
fun maplist (f, Empty) = Empty
   | maplist(f, Node(x1,x2)) =
              Node(maplist(f, x1), maplist(f, x2));
```

ML Nested Blocks

- Syntax: let <varDecls> in <exp> end
- Examples

```
let val x = 3; val y = 4 in x + y end;
let fun foo(x) = x + 1 in foo(4) end;
let val x = 3; val y = 4
in let fun foo(x) = x + 1 in foo(x + y) end
end;
```

- Each let … in …end introduces a number of local variables (or functions)
  - These variables can be used only within the local expression
  - NOTE: function definitions are not evaluated until they are called (invoked) with arguments

ML Assignments and Side-effects

- Creating a reference cell: ref <value>
  - Each reference cell is the address to a box (memory storage)
  - Only reference cells can be modified in ML
- Assignment: <ref cell> := <exp>
  - Assignment has unit type (equivalent to the void type in C)
- Dereference: !<ref cell>
  - Return the value contained in the reference cell
- Examples
  - val x = ref 0; ➔ val x = ref 0 : int ref
  - x := 3 * (!x) + 5; ➔ val it = () : unit
  - !x; ➔ val it = 5: int
  - val y = ref "apple"; ➔ val y = ref "apple" : string ref
  - y := "Green tomatoes"; ➔ val it = () : unit
  - !y; ➔ val it = "Green tomatoes" : string

ML loops

- Syntax:

  <loop> ::= while <exp> do <exp>;

  - Loops do not return values (has unit type)
- Loops must operate through assignments
  - Within each function definition, first use nested blocks to create local reference cells
  - Repetitively modify the cells to accumulate results
  - Return the accumulated results after the loop terminates

Example: Recursion vs. Loops

- Append lists

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

- Using loop and modification

```
fun append(xs,ys) =
    let val rxs = ref (reverse(xs)); val res = ref ys;
    in while not (null(!rxs)) do
       (res := hd(!rxs)::(!res); rxs := tl(!rxs) );
        !res
    end;
```