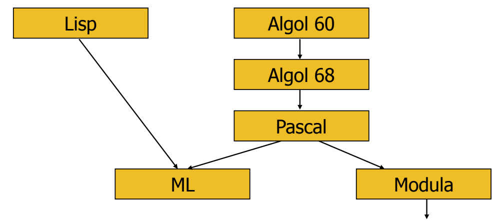
ML

CSCI 334
Stephen Freund

Language Sequence



Many other languages:
Algol 58, Algol W, Euclid, EL1, Mesa (PARC), …
Modula-2, Oberon, Modula-3 (DEC)

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;
```


ML

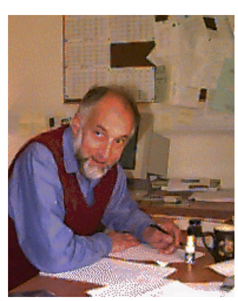
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection

  - **Static types**
  - **Abstract data types**
  - **Module system**
  - **Exceptions**
- General purpose non-C-like, non-OO language

Goals in study of ML

- Types and type checking
  - Static vs. dynamic typing
  - Type inference
  - Polymorphism and Generic Programming
- Memory management
  - Static scope and block structure
  - parameter passing
  - Function activation records, higher-order functions
- Control
  - Statements, { blocks }, ...
  - Exceptions
  - Tail recursion

Robin Milner and ML's Origins



- Dana Scott, 1969
  - LCF
  - logic for stating theorems about programs
- Robin Milner
  - automated theorem proving for LCF
  - theorem proving is a hard search problem
  - ML: meta-language for writing programs (tactics) to find proofs

Tactics

- Tactics guide search in theorem prover
- Tactic is *partial* function from formula -> proof
  - finds proof
  - never terminates
  - reports an error

7

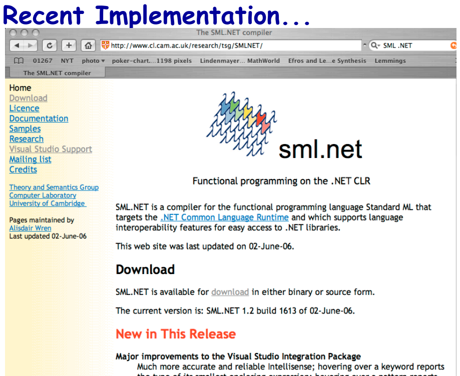
Language Ideas to Support Tactics

- 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) =

    λformula. if t1(formula) then …
    else if t2(formula) …

8

A Recent Implementation...



9

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`"

10

Defining Functions

No type info given- compiler infers it

- 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
```

11

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

12

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
```

13

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.

14

Built-in Data Types

- 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"

15

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
```

16

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
```

17

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);
```

18

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]`

Lists

- 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

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

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

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


Polymorphism

```
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
```

Type variable represents any type