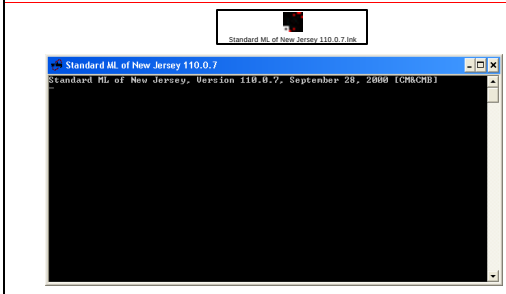
An Introduction (I) to SML, Java & Prolog

SML

Function-Oriented Style

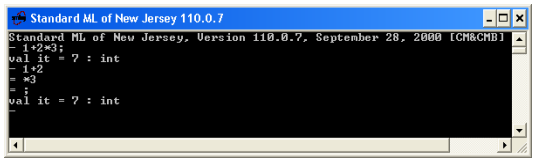
- Functions
- Function applications
- Expressions
- Solving problems using functions!
- There are a number of dialects
  - Standard ML-NJ

SML-NJ Environment



- Prompt

SML-NJ Environment



- Prompt

expression

= Next prompt

;

ML replies with value and type

it = a special variable that receives the value of the expression you type

Primitive Data Types: Integer, Real, Boolean

```
-1234;
val it = 1234 : int
- 123.4;
val it = 123.4 : real
- true;
val it = true : bool
- false;
val it = false : bool
```

Primitive Data Types: String, Character

```
- "fred";
val it = "fred" : string
- "H";
val it = "H" : string
- #"H";
val it = #"H" : char
```


Operators

```
- ~ 1 + 2 - 3 * 4 div 5 mod 6;
val it = ~1 : int
- ~ 1.0 + 2.0 - 3.0 * 4.0 / 5.0;
val it = ~1.4 : real
```

Left associative, precedence is {**+**,**-**} < {*****,**/**,**div**,**mod**} < {**~**}.

Operators

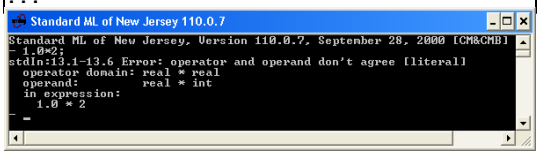
```
- "bibity" ^ "bobity" ^ "boo";
val it = "bibitybobityboo" : string
- 2 < 3;
val it = true : bool
- 1.0 <= 1.0;
val it = true : bool
- 1 = 2;
val it = false : bool
- true <> false;
val it = true : bool
```


Conditional Expression

```
- if 1 < 2 then #"x" else #"y";
val it = #"x" : char
- if 1 > 2 then 34 else 56;
val it = 56 : int
- (if 1 < 2 then 34 else 56) + 1;
val it = 35 : int
```


Types

```
- 1 * 2;
val it = 2 : int
- 1.0 * 2.0;
val it = 2.0 : real
- 1.0 * 2;
???
```



ML does not perform implicit type conversion

Type Conversion Functions

```
- real(123);
val it = 123.0 : real
- floor(3.6);
val it = 3 : int
- floor 3.6;
val it = 3 : int
- str #"a";
val it = "a" : string
```

Variables

```
- val x = 1+2*3;
val x = 7 : int
- x;
val it = 7 : int
- val y = if x = 7 then 1.0 else 2.0;
val y = 1.0 : real
```


Local Variables

```
-let val x = 1 in x+2*3 end;
val it = 7 : int
- x;
???
- let val x=1
=val y=2
=in
=x+y
=end;
val it = 3 : int
```


Functions

```
-fun square x = x*x;
val square = fn : int -> int
```

```
<fun-def> ::=
        fun <function-name>
            <parameter> =
                <expression> ;
```


Function Application

```
- square 2;
val it = 4 : int
- square 2+1;
val it = 5 : int
- square (2+1);
val it = 9 : int
```

Function application has higher precedence than any operator.

Functions

```
-fun square x = x*x;
val square = fn : int -> int
-square 2;
val it = 4 : int
-val s = fn x => x*x;
val square = fn : int -> int
-s 2;
val it = 4 : int
```


Functions

```
-fun quot(a,b) = a div b;
val quot = fn : int * int -> int
- quot (6,2);
val it = 3 : int
- val pair = (6,2);
val pair = (6,2) : int * int
- quot pair;
val it = 3 : int
```

All ML functions take exactly one parameter.

To pass more than one thing, you can pass a tuple.

Example – The Factorial Function

```
-fun fact n =
=   if n = 0 then 1
=   else n * fact(n-1);
val fact = fn : int -> int
- fact 5;
val it = 120 : int


- fun fact (n: int): int =
=   if n = 0 then 1
=   else n * fact(n-1);
val fact = fn : int -> int
```


Tuples

```
- (1, 2);
val it = (1,2) : int * int
- #1 (1, 2);
val it = 1 : int
- val point1 = ("red", (300,200));
val point1 = ("red",(300,200)) : string *
(int * int)
- #1 (#2 point1);
val it = 300 : int
```

A tuple is like a record with no field names.

Lists

```
- [1,2,3];
val it = [1,2,3] : int list
- [1.0,2.0];
val it = [1.0,2.0] : real list
- [true];
val it = [true] : bool list
- [(1,2),(1,3)];
val it = [(1,2),(1,3)] : (int * int) list
- [[1,2,3],[1,2]];
val it = [[1,2,3],[1,2]] : int list list
```

Unlike tuples, all elements of a list must be the same type.

Lists

```
-[];
val it = [] : 'a list
- nil;
val it = [] : 'a list

-null [];
val it = true : bool
- null [1,2,3];
val it = false : bool

-nil = [];
val it = true : bool
```

Empty list is **`[]`** or **`nil`**.

List Operations - ::

```
- 1 :: [2,3];
val it = [1,2,3] : int list
- op ::;
val it = fn: 'a * 'a list -> 'a list
-val z = 1::2::3::[];
val z = [1,2,3] : int list
- [1,2] :: [3,4];
-???
```

List-builder (*cons*) operator is **`::`**

The **`::`** operator is right-associative.

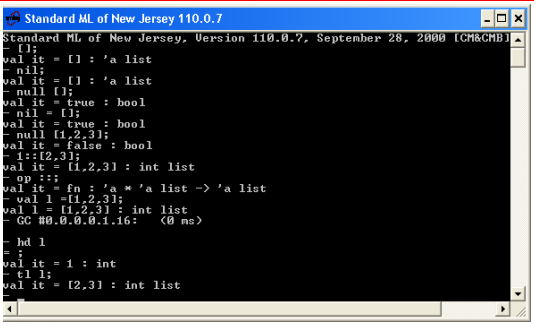
List Operations

```
- hd z;
val it = 1 : int
- tl z;
val it = [2,3] : int list
- tl(tl z);
val it = [3] : int list
- tl(tl(tl z));
val it = [] : int list
```

The **`hd`** function gets the head of a list: the first element.

The **`tl`** function gets the tail of a list: the whole list after the first element.

List Operations



List Operations

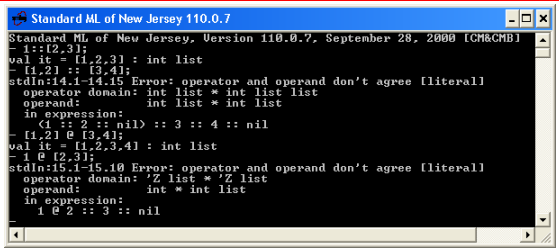
```
-[1,2,3]@[4,5,6];
val it = [1,2,3,4,5,6] : int list
- op @;
val it = fn: 'a list * 'a list -> 'a list
- 1 @ [2,3];
???
```

The **@** operator concatenates lists.

Operands are two lists of the same type.

Note: **1@[2,3,4]** is wrong: either use **[1]@[2,3,4]** or **1::[2,3,4]**

List Operations



Note: **1@[2,3,4]** is wrong: either use **[1]@[2,3,4]** or **1::[2,3,4]**

List Functions

```
-fun length x =
=   if null x then 0
=   else 1 + length (tl x);
val length = fn : 'a list -> int

-length [true,false,true];
val it = 3 : int

- length [4.0,3.0,2.0,1.0];
val it = 4 : int
```


List Functions

```
-fun listsum x =
=   if null x then 0
=   else hd x + listsum(tl x);
val listsum = fn : int list -> int

- listsum [1,2,3,4,5];
val it = 15 : int
```


List Functions

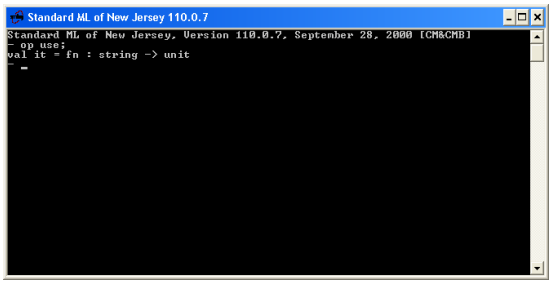
```
-fun reverse L =
=   if null L then nil
=   else reverse(tl L) @ [hd L];
val reverse = fn : 'a list -> 'a list

- reverse [1,2,3];
val it = [3,2,1] : int list
```

Garbage Collection

- GC #0.0.0.0.1.3: (0 ms)

Use



Use

```
-op use;
val it = fn: string -> unit

-use ("filename.sml");
```

factorial.sml:

```
(* COMMENTS factorial function *)
fun fact n =  if n = 0 then 1 else n * fact(n-1);
```


Use