SML Syntax

String

`#"str"`	character
`String.sub : string * int -> char`	n-th character
`chr`	ascii to character
`ord : char -> int`	character to ascii
`^`	concatenate
`String.tokens : (char -> bool) -> string -> string list`	tokenize a string
`String.explode : string -> char list`	also implode

List

`@ : 'a list @ 'a list`	concatenation
`List.partition : ('a -> bool) -> 'a list -> 'a list * 'a list`	quicksort
`List.rev : 'a list -> 'a list`	reverse
`List.exists : ('a -> bool) -> 'a list -> bool`	true for any
`List.all : ('a -> bool) -> 'a list -> bool`	true for all
`String.concatWith : string -> string list -> string`	

Referential Transparency

Replace any expression with another expression of "equal" value does not affect the value of the expression

Equivalence

Two programs are equivalent iff
1. They both evaluate to the same value, or
2. They both raise the same exception, or
3. They both enter an infinite loop
Properties
1. Equivalence is an equivalence relation
2. Equivalence is a congruence (one program can be substituted for another)
3. If e |-> e' then e is equivalent to e'

Valuable & Total

Expression **e** is valuable iff there is some value **v** s.t. **e** == **v**
- If **e** = (**e1**,**e2**)
- If **e** = **e1**+**e2**
- If **e** = **e1** :: **e2**
then e is valuable iff e1 is valuable and e2 is valuable
A function **f** : A -> B is total iff for all values**v** : A, **f(v)** is valuable

Currying

```
=== Non-curried ===
fun pow (x, y) : int * int -> int =
    case y of`
        0 => 1`
    |   _ => x * pow(x, y-1)
=== Curried ===
fun pow x : int -> int -> int =
    fn (y) => case y of
                  0 => 1
              |   _ => x * pow(x, y-1)
fun pow x y =
    case y of
        0 => 1
    |   _ => x * pow(x, y-1)
=== Currying and Uncurrying ===
curry : (('a * 'b) -> 'c) -> ('a -> 'b -> 'c))
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
uncurry : ('a -> 'b -> 'c) -> (('a 'b) -> 'c)
```

Composition

```
fun compose (f, g) = fn x => f(g x)
```
Using infix operator
```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```
Pipelining and infix pipeline operator
`fun pipeline (f, g) = g f
```
infix !>
fun x !> f = f x
fun sqrt_of_abs i =i !> Real.fromInt !> Math.sqrt
datatype 'a list = Nil | :: of 'a * 'a list
```

Mergesort

```
fun split (lst : int list) : int list * int list =
    case lst of
        [] => ([], [])
    |   [x] => ([x], [])
    |   x::y::xs => let val (pile1, pile2) = split xs
                    in (x::pile1, y::pile2)
                    end
fun merge(lst1 : int list, lst2 : int list) : int list
=
    case (lst1, lst2) of
        ([], lst2) => lst2
    |   (lst1, []) => lst1
    |   (x::xs, y::ys) =>
            (case x < y of
                true => x::merge(xs,lst2)
            |   false => y::merge(lst1, ys)
fun mergesort (lst : int list) : int list =
    case lst of
        [] => []
    |   [x] => [x]
    | _ => let val (pile1, pile2) = split lst
           in merge(mergesort pile1, mergesort pile2)
           end
```

Generalized math functions

```
fun sum (f, a, b, inc) :
    if (a > b) then 0
    else (f a) + sum(f, inc(a), b, inc)
fun piOver8 = sum(fn x => 1.0 / (x*(x+2.0)), a, b, fn x
=> x + 4.0)
fun integral (f, a, b, dx) =
    dx * sum(f, a+dx/2.0, b, fn x => x+dx)
fun series (operator, f, lo, hi, inc, identity) =
    if (lo > hi) then identity
    else operator((f lo), series (operator, f,
inc(lo), hi, inc, identity))
fun sumSeries (f, a, b, inc) = series (op +, f, a, b,
inc, 0)
fun prodSeries(f, a, b, inc) = series(op *, f, a, b,
inc, 1)
```

Data types

User defined types

```
datatype tree = Empty | Node of tree * int * tree
datatype 'a option = NONE | SOME of 'a
```

Type synonym

```
type intPairList = (int * int) list
```

Map

```
map : ('a -> 'b) * 'a list -> b' list
fun map (f lst) =
    case lst of
        [] => []
    |   h::tail => (f h)::(map f tail)
```

Fold

```
fun foldl(f, acc, lst) =
    case lst of
        [] => acc
    |   h::t => foldl(f, f(h, acc), t) (*tail
recursive*)
fun foldr(f, acc, lst) =
    case lst of
        [] => acc
    |   h::t => f(h, foldr(f, acc, t)) (*not tail
recursive*)
```

Associativity

```
'a -> 'b -> 'c = 'a -> ('b -> 'c)
f a1 a2 = (f a1) a2
```

Filter

```
filter : ('a -> bool) * 'a list -> 'a list
fun filter (p : 'a -> bool, lst : 'a list) =
    case lst of
        [] => []
    |   x::xs => if p x then x::(filter p xs)
                 else filter p xs
```