Joseph Maxwell

Additional notes and functions are listed on the bottom.

1. *(\* Alternate input from 2 lists into a single list*

*Ex: alternate([1,3,5],[2,4,6]) -> [1,2,3,4,5,6]\*)*

fun alternate(*[]*,*[]*) *=* *[]*

*|* alternate(*x*::*xs*, *y*::*ys*) *=* x::y::alternate(xs, ys);

1. *(\* Minus input integer lists*

*Ex: input([1,1,1,2,2],[1,1,2,3])*

*minus([1,1,1,2,2],[1,1,2,3]) = [1,2] integers in common*

*minus([1,1,1,2,2],[1,1,2,3]) = [3] after removing common integers\*\*)*

fun delete(*x*, *[]*) *=* *[](\*removes an element from a list\*)*

*|* delete(*x*,*y*::*l*) *=* if x=y then delete(x,l) else y::delete(x,l);

fun removeDuplicate *[]* *=* *[](\*removes all duplicate elements\*)*

*|* removeDuplicate (*x*::*l*) *=* x::removeDuplicate(delete(x,l));

fun remove (*\_*, *[]*) *=* *[](\*Assumes all elements are unique\*)*

*|* remove (*x*, *y*::*ys*) *=* if x = y then ys

        else y :: remove (x,ys);

fun filter (*[]*,    *ys*) *=* ys

*|* filter (*x*::*xs*, *ys*) *=*

*let*

        val ysWithoutX *=* remove (x, ys)

*in*  *(\*filter([1,2,3],[1,4,2,6,3,7]) -> [4,6,7]  must have unique elements\*)*

        filter(xs, ysWithoutX)

*end*;

fun minus(*x*,*y*) *=* filter(removeDuplicate(x), removeDuplicate(y));

1. *(\* Union input two lists and output union of the two without duplication\*)*

    Ex: Union([1,1,1,2,2],[1,1,2,3]) = [1,2,3]

    Ex2:Union([h,e,l,l,o],[w,o,r,l,d]) = [h,e,l,o,w,r,d]

fun delete(*x*, *[]*) *=* *[](\*removes an element from a list\*)*

*|* delete(*x*,*y*::*l*) *=* if x=y then delete(x,l) else y::delete(x,l);

fun removeDuplicate *[]* *=* *[](\*removes all duplicate elements\*)*

*|* removeDuplicate (*x*::*l*) *=* x::removeDuplicate(delete(x,l));

fun flatten *[]* *=* *[]*

*|* flatten (*x*::*l*) *=* x @ flatten l;

fun union(*x*,*y*) *=* removeDuplicate(flatten*[*x,y*]*);

1. *(\* Intersection takes in multiple sets and creates a list of matching elements in all sets*

*Ex: Intersect([[1,1,1,2,2],[1,1,2,3],[2,3,5,5],[3,5,7,4]]) = [] there is no overlaping element in all sets \*)*

fun member(*x*,*[]*) *=* *false(\*if at end of list, return false base case\*)*

*|* member(*x*,*b*::*y*) *=(\*checks element x against element b from list y, similar to for loop check\*)*

        if x=b then *true(\*positive match\*)*

        else member(x,y);

fun aux(*[]*,*x*) *=* *[]*

*|* aux(*x*::*xs*,*ys*) *=*

        if member(x,ys)then x::aux(xs,ys)

        else aux(xs,ys);

fun multiSetIntersection(*[]*) *=* *[]*

*|* multiSetIntersection(*[xs]*) *=* xs

*|* multiSetIntersection (*xs*::*xss*) *=* aux(xs, multiSetIntersection(xss));

1. *(\* Cartesian Product Function*

*Ex: S1 = {a,b,c}*

*S2 = {1,2}*

*S1xS2 = {(a,1),(a,2),(b,1),(b,2),(c,1),(c,2)}\*)*

*(\*hint: \*)*

fun prodBlock (*[]*,*\_*) *=* *[](\*takes 2 sets and multiplies eachother, returns new set\*)*

*|* prodBlock ((*x*::*xs*), *ys*) *=* map (fn *y* *=>* (x,y)) ys @ prodBlock (xs, ys)

fun Cartesian *zs* *=* foldl (fn (*xs*, *ys*) *=>* map op:: (prodBlock (xs, ys))) *[[]]* (rev zs);

*(\*Fold video: Notable slides SML 190 - fold.pptx\*)*

1. *(\* Powerlist set*

*Ex: S = {1,2}*

*PowS = {{1,2},{1},{2},{}} \*)*

*(\*I got help on this function, time was running short\*)*

fun powerset *[]* *=* *[](\*base case if empty\*)*

*|* powerset *[x]* *=* *[[]*,*[*x*]](\*\*)*

*|* powerset (*x*::*xs*) *=*

*let*

        val power\_subset *=* powerset xs*(\*creates val which is the next recursive step\*)*

*in*

        (List.map (fn *L* *=>* x::*L*) power\_subset) @ power\_subset*(\*appends subset to current set\*)*

*end*;

fun union (*e*, *[]* *:* ''*a* list) *:* ''*a* list *=* *[*e*]*

*|* union (*e*, *x*::*xs*) *=*

    if e = x then x::xs

    else x::union(e, xs)

fun insert (*e* *:* ''*a*, [] : ''*a* list list) *:* ''*a* list list *=* *[]*

*|* insert (*e*, *s*::*ss*) *=* union(e, s)::insert(e, ss)

fun Powerlist *[]* *=* *[](\*core function base case\*)*

*|* Powerlist *[x]* *=* *[[]*,*[*x*]]*

*|* Powerlist (*x*::*xs*) *=*

*let*

        val power\_subset *=* powerset xs

*in*

        power\_subset @ insert(x, power\_subset)

*end*;

1. *(\* finiteListRepresentation takes in a function and number X*

*Returns the function output for first X times*

*Ex: FLR( posIntSqr, 5 ) = [(1,1),(2,4),(3,9),(4,16),(5,25)]*

*Simpple generator??!?\*)*

fun reverse *[]* *=* *[]*

*|* reverse (*x*::*xs*) *=* reverse xs @ *[*x*]*;

fun generate 0 *=* *[](\*easy to understand recursive loop\*)*

*|* generate *n* *=* *[[*n,n\*n*]]*@generate (n-1);

fun FLR(*x*) *=* reverse(generate x);

1. *(\* Update SML function: Updates a finite list with new values*

*Ex: Let FLR = [(1,1),(2,4),(3,9),(4,16),(5,25)]*

*update(FLR, (2,3)) = [(1,1),(2,3),(3,9),(4,16),(5,25)]*

NOTES:

*(\* Our first SML program \*)*

print "\n\nhello world\n\n";

fun isEven  *n* *=* n mod 2 = 0;

fun succ    *n* *=* if isEven n then n div 2 else 3 \* n + 1;

fun maxOf (*v*, *w*) *=* if v < w then w else v;

fun threeN *n* *=*

*let*

        val trackFn *=* maxOf

        fun aux (1,*max*) *=* trackFn (1,max)

*|*   aux (*n*,*max*) *=* aux( succ n, trackFn(n,max) )

*in*

        aux( n, 0 )

*end*;

threeN 7;

fun length *[]* *=* 0*(\*Recursive program with a 0 base case\*)*

| length (x::xs) = 1 + length xs;*(\*if exists, then 1 + next level\*)*

*(\* to return "length [1,2,3]" is equivalent to an int value\*)*

*(\*[1,2,3] can be substituted with any list\*)*

*(\* visual output (1+(1+(1+0))) = 3\*)*

fun sumList *[]* *=* 0*(\*Similar to length, except adds value in list\*)*

| sumList (x::xs) = x + sumList xs;*(\*Only works with int lists\*)*

fun isFactorOf (*k*,*n*) *=* n mod k = 0;

fun isPrime *n* *=*

    if n < 2 then *false*

    else *(\*Could be prime\*)*

*let*

            fun aux 1 *=* *true*

*|* aux *k* *=* if isFactorOf(k,n) then *false*

                else aux (k-1)

*in*

            aux (n-1)

*end*;

fun validateIsPrime *[]*          *=* *true(\*checks all values in list\*)*

*|* validateIsPrime (*x*::*xs*)   *=* isPrime x

                                  andalso*(\*If ALL is prime, return true\*)*

                                  validateIsPrime xs;

fun validateNonPrime *[]*          *=* *true(\*checks all values in list\*)*

*|* validateNonPrime (*x*::*xs*)   *=* not (isPrime x)

                                  andalso*(\*If ALL is non-prime, return true\*)*

                                  validateNonPrime xs;

fun maxOf *[x]* *=* x

*|* maxOf (*x*::*xs*) *=*

*let*

        val max *=* maxOf xs

*in*

        if x < max then max else x

*end*;

fun generate 0 *=* *[](\*generates a list from 1-n\*)*

*|* generate *n* *=* n :: generate (n-1);*(\*n :: generate appends generate after n in list\*)*

*(\*generate 10 creates [10,9,8,7,6,5,4,3,2,1]\*)*

fun remove (*\_*, *[]*) *=* *[](\*Assumes all elements are unique\*)*

*|* remove (*x*, *y*::*ys*) *=* if x = y then ys

        else y :: remove (x,ys);

fun filter (*[]*,    *ys*) *=* ys

*|* filter (*x*::*xs*, *ys*) *=*

*let*

        val ysWithoutX *=* remove (x, ys)

*in*  *(\*filter([1,2,3],[1,4,2,6,3,7]) -> [4,6,7]  must have unique elements\*)*

        filter(xs, ysWithoutX)

*end*;

fun test *primes* *=*

*let*

        val maxPrime *=* maxOf primes;

        val integerList *=* generate maxPrime;

        val nonPrimeList *=* filter (primes, integerList);

*in*

        validateNonPrime nonPrimeList

*end*;

fun delete(*x*, *[]*) *=* *[](\*removes an element from a list\*)*

*|* delete(*x*,*y*::*l*) *=* if x=y then delete(x,l) else y::delete(x,l);

fun removeDuplicate *[]* *=* *[](\*removes all duplicate elements\*)*

*|* removeDuplicate (*x*::*l*) *=* x::removeDuplicate(delete(x,l));

fun simpleMerge *[]* *=* *[]*

*|* simpleMerge (*x*::*l*) *=* x @ simpleMerge l;

fun sum\_pair\_list (*xs* *:* (int \* int) list) *=*

  if null xs

  then 0

  else #1 (hd xs) + #2 (hd xs) + sum\_pair\_list(tl xs)

*(\*sum\_pair\_list [(3,4),(5,6)] -> val it = 18 or (3+5)+(4+6)\*)*

fun map *f* *=*

*let*

    fun m *nil* *=* nil

*|* m (*x*::*xs*) *=* f x :: m xs

*in*

    m

*end*;

fun sq *x* *=* x\*x;

val sqList *=* map sq;

sqList *[*1,2,3,4*]*;

map sqList *[[*1,2*]*,*[*3,4*]*,*[*5,6*]]*;

*(\*  List is a homogeneous aggregation*

*- aggregarion of values of the same type.*

*- Can change sizes*

*Tuples are of different types*

*- Many types*

*- Fixed sizes\*)*

*(\*  cons: element \* element list -> element list*

*type constraint*

*nil: type of 'a list, 'a means un defined*

*cons(1, nil)                    term of int list*

*cons(true, nil)                 term of bool list*

*cons(nil, 1)                    ERROR does not work, bad order*

*cons(1,cons(2 nil))             a list of 2 ints*

*cons((1,2),cons((3,4),nil))     a list containg 2 tuples\*)*

*(\*cons(x,nil)<<cons(1,nil)\*)*

*(\*  When variable 'x' is used to denote a list element(single)*

*xs denotes a list of x elements(list)*

*[1,2,3]                                     int list*

*[(1,true),(2,false),(3,true)] (int\*bool)    list or a tuple list*

*[[1,2],[3],[4,5,6]]                         int list list*

*[]                                          empty list*

*Evaluation is left to right*

*-Operand :: = 'a \*'a list -> 'a list         takes and element on the left and a list on the right then adds the element to the front of the list*

*-Operand @ = 'a list \* 'a list -> 'a list    takes two lists and concatonates the second onto the end of the first*

*Ex: 1::[]       -> [1]*

*Ex: 2::[1]      -> [2,1]*

*Ex: 1::2::[]    -> [1,2]*

*Ex: 1::[2]::3   -> fails, bad order*

*Ex: [1]@[2]     -> [1,2]*

*Ex: []@[1]      -> [1]*

*Ex: special []::[]->[[]]\*)*

*(\*  hd [x] is the first element of the list*

*tl [x] is the list after the "head"\*)*