(\*

datatype card = Jack of suit |

Queen of suit |

King of suit |

Ace of suit |

Num of suit \* int

\*)

datatype suit = Club | Diamond | Heart | Spade

datatype card\_value = Jack | Queen | King

| Ace | Num of int

datatype card = Card of suit \* card\_value

val hands = [Card(Club, Jack), Card(Club, Num(10)), Card(Club, Ace)]

val hands2 = [Card(Club, Jack), Card(Diamond, Num(10)), Card(Club, Ace)]

val hands3 = [Card(Diamond, Num(10)), Card(Club, Ace), Card(Club, Jack)]

val hands4 = [Card(Diamond, Num(10)), Card(Club, Ace), Card(Club, Jack), Card(Spade, Ace)]

(\* assume hand is not empty \*)

fun is\_flush (hand: card list) =

case hand of

Card(\_, \_)::[] => true

| Card(shape1, \_)::Card(shape2, \_)::\_ => shape1 = shape2 andalso

is\_flush(tl hand)

| \_ => false

fun simpleSum(hand: card list) =

case hand of

Card (\_, Ace)::rest => 11 + simpleSum(rest)

| Card (\_, Num(i))::rest => i + simpleSum(rest)

| Card (\_, \_)::rest => 10 + simpleSum(rest)

| \_ => 0

fun hasAce (hand: card list) =

case hand of

Card(\_, Ace)::rest => true

| Card(\_, \_)::rest => hasAce(rest)

| \_ => false

(\* card list -> card list \*)

fun removeAce (hand: card list) =

case hand of

Card(\_, Ace)::rest => rest

| Card(\_, \_)::rest => (hd hand)::removeAce(rest)

| \_ => hand

(\* Big-O complexity of blackjack? \*)

fun blackjack(hand: card list) =

let val sum = simpleSum(hand)

in

if sum <= 21

then sum

else

if hasAce(hand)

then 1 + blackjack(removeAce hand)

else sum

end

(\* blackjack, more efficient version \*)

fun blackjack (score: int, hand: card list) =

case hand of

Card (\_, Ace)::rest => let val try1 = blackjack(score+11, rest)

in

if try1 <= 21

then try1

else blackjack(score+1, rest)

end

| Card (\_, Num(i))::rest => blackjack(score+i, rest)

| Card (\_, \_)::rest => blackjack(score+10, rest)

| \_ => score

datatype exp = Constant of int

| Negate of exp

| Add of exp \* exp

| Multiply of exp \* exp

| If of bool \* exp \* exp

(\* creating exp tree \*)

val ifExpr = If(true, Add(Constant 10, Constant 11),

Multiply(Constant 1, Constant 42))

(\* evaluation of the exp tree \*)

fun eval(e) =

case e of

Constant i => i

| Negate(e2) => ~ (eval (e2))

| Add(e1, e2) => eval(e1) + eval(e2)

| Multiply(e1, e2) => eval(e1) \* eval(e2)

| If(true, e1, e2) => eval(e1)

| If(false, e1, e2) => eval(e2)

fun max\_of\_two (i1, i2) =

if i1 > i2

then i1

else i2

(\* max\_constant: exp -> int \*)

fun max\_constant (e: exp) =

case e of

Constant i => i

| Negate(e2) => max\_constant(e2)

| Add(e1, e2) => max\_of\_two(max\_constant(e1), max\_constant(e2))

| Multiply(e1, e2) => max\_of\_two(max\_constant(e1), max\_constant(e2))

| If(\_, e1, e2) => max\_of\_two(max\_constant(e1), max\_constant(e2))

(\* abstract data type using closures \*)

(\* int set

\* with 3 methods: insert, member, size

\* notice insert returns a set (a new set)

\*)

datatype set = S of { insert : int -> set,

member : int -> bool,

size : unit -> int,

members: int list,

delete : int -> set

}

(\* implementation of sets: this is the fancy stuff, but clients using

this abstraction do not need to understand it \*)

val empty\_set =

let

fun make\_set xs =

let fun contains i = List.exists (fn j => i=j) xs

in

S { insert = fn i => if contains i

then make\_set xs

else make\_set (i::xs),

member = contains,

size = fn () => length xs,

members = xs,

delete = fn i => make\_set (List.filter (fn j => i<>j) xs)

}

end

in

make\_set []

end

(\* example client \*)

val S s1 = empty\_set;

val S s2 = (#insert s1) 34;

val S s3 = (#insert s2) 34;

val S s4 = #insert s3 19;

fun map (f, xs) =

case xs of

[] => []

| x::xs' => f(x)::map(f, xs')

fun filter(f, xs) =

case xs of

[] => []

| x::xs' => if f x then x::filter(f, xs')

else filter(f, xs')

fun fold(f, acc, xs) =

case xs of

[] => acc

| x::xs' => fold(f, (f(acc,x)), xs')

val nums\_list = [[9, 40, 75, 7],

[64, 34, 88, 96],

[91, 92, 53, 31],

[50, 84, 73, 65],

[54, 44, 75, 11],

[91, 71, 48, 46],

[70, 72, 5, 42],

[25, 77, 49, 56],

[89, 4, 73, 52],

[36, 56, 61, 1]]

(\* fun fold(f, acc, xs) \*)

val local\_max = map(fn nums => fold(Int.max, hd(nums), nums),

nums\_list)

(\* [75, 96, ... ] \*)

val global\_max = fold(Int.max, hd(local\_max), local\_max)

(\* given x, count the multiples of x in each list

\* x=11, [[1, 2, 11], [2, 3, 22, 33], [4, 5]]

\* ==> [1, 2, 0]

\*)

(\* 1. apply modular (filter) to \*each\* list

\* ==> [[11], [22, 33], []]

\* 2. count \*each\* list

\* ==> [1, 2, 0]

\*)

(\* filter(f, xs) \*)

fun count\_multiples (x, nums\_list) =

map(length,

map(fn nums => filter(fn y => (y mod x)=0 , nums),

nums\_list))

(\* similar to above, given x, count the multiples of x in each list

\* and returns the index of the list having the maximum count.

\* x=11, [[1, 2, 11], [11, 22, 33], [4, 5]]

\* ==> 1

\*)

(\* 1. call count\_multiples above

\* 2. apply fold. acc = (max\_index, curr\_index, max\_value)

\*)

fun index\_of\_max\_multiple\_count (x, nums\_list) =

let val counts = count\_multiples(x, nums\_list)

(\* [1, 3, 3] \*)

in

fold(fn (acc, y) => if (#3 acc) > y then (#1 acc, 1+(#2 acc), #3 acc)

else (#2 acc, 1+(#2 acc), y),

(0, 0, hd(counts)),

counts)

end

exception InvalidArgument

fun good\_max (xs : int list) =

if null xs

then 0 (\* horrible style; fix later \*)

else if null (tl xs)

then hd xs

else

let val max\_rest = good\_max(tl xs)

in

if hd xs > max\_rest

then hd xs

else max\_rest

end

fun good\_max (xs, exc) =

case xs of

[] => raise exc

| x::[] => x

| x::xs' => Int.max(x, good\_max(xs'))

fun fibo (0) = 1

| fibo (1) = 1

| fibo (n) = fibo(n-1) + fibo(n-2)

fun fibo\_series(0) = [fibo(0)]

| fibo\_series(n) = fibo\_series(n-1) @ [fibo(n)]

(\* fibo3 returns a tuple (f\_n, f\_(n-1))

e.g. fibo3 0 ==> (1, 1)

fibo3 1 ==> (2, 1)

fibo3 2 ==> (3, 2)

\*)

fun fibo3 (0) = (1, 1)

| fibo3 (n) =

let val (p, pp) = fibo3(n-1) (\* n=1, fibo3(0) => (1,1) ==> (2, 1) \*)

in

(p+pp, p)

end

(\* nice example of nested pattern. \*)

(\* nondecreasing: int list -> bool

\*

\* e.g. [1,1,2, 3, 4, 5, 5] => true

\* [1, 2, 1, 3, 5] => false

\*)

fun nondecreasing xs =

case xs of

[] => true

| x::[] => true

| head::(neck::rest) => (head <= neck andalso nondecreasing (neck::rest))

datatype sgn = P | N | Z

(\* multsign: int \* int -> sgn

\*

\* e.g. 1, 1 => P

\* ~1, 10 => N

\* ~10, ~42 => P

\*)

fun multsign (x1,x2) =

let fun sign x = if x=0 then Z else if x>0 then P else N

in

case (sign x1,sign x2) of

(Z,\_) => Z

| (\_,Z) => Z

| (P,P) => P

| (N,N) => P

| \_ => N (\* many say bad style; I am okay with it \*)

end