Cheat Sheets for CSc 372 Exams

Christian Collberg

Useful Functions from the Haskell Standard Prelude

```
fst              :: (a,b) -> a
fst (x,_)         = x

snd              :: (a,b) -> b
snd (_,y)         = y

id               :: a -> a
id     x          = x

const            :: a -> b -> a
const k _         = k

(.)              :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x         = f (g x)

head               :: [a] -> a
head (x:_)          = x

last               :: [a] -> a
last [x]            = x
last (_:xs)         = last xs

tail               :: [a] -> [a]
tail (_:xs)         = xs

init               :: [a] -> [a]
init [x]            = []
init (x:xs)         = x : init xs

null               :: [a] -> Bool
null []             = True
null (_:_)          = False

(++)               :: [a] -> [a] -> [a]
[]      ++ ys       = ys
(x:xs) ++ ys        = x : (xs ++ ys)

map                :: (a -> b) -> [a] -> [b]
map f [ ]           = [ ]
map f (x:xs)        = f x : map f xs

filter             :: (a -> Bool) -> [a] -> [a]
filter _ []         = []
filter p (x:xs)
       | p x          = x : filter p xs
       | otherwise    = filter p xs

concat            :: [[a]] -> [a]
concat             = foldr (++) []

length            :: [a] -> Int
length             = foldl (\x _->x+1) 0

(!!)              :: [a] -> Int -> a
(x:_)  !! 0        = x
(_:xs) !! n | n>0 = xs !! (n-1)
(_:_)  !! _        = error "Prelude.!!: negative index"
[]     !! _        = error "Prelude.!!: index too large"

foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs

foldr             :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x:xs)   = f x (foldr f z xs)

iterate           :: (a -> a) -> a -> [a]
iterate f x        = x : iterate f (f x)

take                 :: Int -> [a] -> [a]
take n _   | n <= 0  = []
take _ []            = []
take n (x:xs)        = x : take (n-1) xs

drop                 :: Int -> [a] -> [a]
drop n xs | n <= 0   = xs
drop _ []            = []
drop n (_:xs)        = drop (n-1) xs

zip               :: [a] -> [b] -> [(a,b)]
zip                = zipWith  (\a b -> (a,b))

zipWith                       :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)    = z a b : zipWith z as bs
zipWith _ _      _          = []

takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile p [ ] = [ ]
takeWhile p (x:xs)
      | p x         = x : takeWhile p xs
      | otherwise   = [ ]

dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile p [ ] = [ ]
dropWhile p (x:xs)
      | p x         = dropWhile p xs
      | otherwise   = x:xs
```

Useful Prolog Predicates

```
append([], L, L)
append([X|L1], L2, [X|L3]) :-
      append(L1, L2, L3).

member(X, [X|_]).
member(X, [_|Y]) :- member(X, Y).

delete_one(X,[X|Z],Z).
delete_one(X,[V|Z],[V|Y]) :-
   X \== V, delete_one(X,Z,Y).

permutation(X,[Z|V]) :-
   delete_one(Z,X,Y),
   permutation(Y,V).
permutation([],[]).
```

Useful Icon Builtin Procedures

Numeric Operations

- `I1`, `I2`, ... are integers.
- `N1`, `N2`, ... are arbitrary numeric types.

`abs(N)`	absolute value
`integer(x)`	convert to integer
`iand(I1,I2)`	bitwise and of two integers
`icom(I1,I2)`	bitwise complement of two integers
`ior(I1,I2)`	bitwise inclusive or of two integers
`ishift(I1,I2)`	shift `I1 by I2` positions
`ixor(I1,I2)`	bitwise inclusive or of two integers
`-N`	unary negation
`?N`	random number between 1 and `N`
`N1 + N2`	addition
`N1 - N2`	subtraction
`N1 * N2`	multiplication
`N1 / N2`	quotient
`N1 % N2`	remainder
`N1 ^ N2`	`N1` to the power of `N2`
`N1 > N2`	if `N1 > N2` then `N2` else fail
`N1 >= N2`	if `N1` $\geq$ `N2` then `N2` else fail
`N1 <= N2`	if `N1` $\leq$ `N2` then `N2` else fail
`N1 < N2`	if `N1 < N2` then `N2` else fail
`N1 = N2`	if `N1 = N2` then `N2` else fail
`N1 ~= N2`	if `N1` $\neq$ `N2` then `N2` else fail
`N1` *op*`:= N2`	`N1 := N1` *op* `N2`, where op is any one of the binary operators. Examples: `X +:= Y` $\equiv$ `X := X + Y, X ||:= Y` $\equiv$ `X := X || Y.`
`seq(I1,I2)`	generate the integers `I1`, `I1+I2`, `I1+2*I2`, `I1+3*I2`, ...
`I1 to I2 by I3`	generate the integers between `I1` and `I2` in increments of `I3`
`&time`	elapsed time

String Operations

`char(i)`	ASCII character number `i`
`find(s, p, f, t)`	positions in `p[f:t]` where `s` occurs.
`map(s1, s2, s3)`	map characters in `s1` that occur in `s2` into the corresponding character in `s3`
`ord(C)`	convert character to ASCII number
`string(X)`	convert `X` to a string
`reverse(S)`	return the reverse of `S`
`type(X)`	return the type of `X` as a string
`*S`	length of `S`
`?S`	random character selected from `S`
`!S`	generate characters of `S` in order
`S1 || S2`	string concatenation
`S1 >> S2`	if `S1 > S2` then `S2` else fail
`S1 >>= S2`	if `S1` $\geq$ `S2` then `S2` else fail
`S1 == S2`	if `S1 = S2` then `S2` else fail
`S1 <<= S2`	if `S1` $\leq$ `S2` then `S2` else fail
`S1 << S2`	if `S1 < S2` then `S2` else fail
`S1 ~== S2`	if `S1` $\neq$ `S2` then `S2` else fail
`S[i]`	`i`*th* character of `S`
`S[f:t]`	substring of `S` from `f` to `t`
`&clock`	time of day
`&date`	date
`&dateline`	date and time of day

Procedures and Variables

`args(P)`	return number of arguments of procedure `P`
`exit(I)`	exit program with status `I`
`getenv(S)`	return value of environment variable `S`
`name(X)`	return the name of variable `X`
`proc(S)`	return the procedure whose name is `S`
`variable(S)`	return the variable whose name is `S`
`P!L`	call procedure `P` with arguments from the list `L`
`stop(I,X1,X2,...)`	exit program with error status `I` after writing strings `X1`, `X2`, etc.

File Operations

- `F is a file variable.`

close(F)	close file F
open(S1, S2)	open and return the file whose name is S1. S2 gives the options: "r"=open for reading, "w"=open for writing, "a"=open for append, "b"=open for read & write, "c"=create.
read(F)	read the next line from file F
reads(F,i)	read the next i characters from F
rename(S1,S2)	rename file S1 to S2
remove(S)	remove the file whose name is S
where(F)	return current byte position in file F
seek(F, I)	move to byte position I in file F
write(F, X1, X2, ...)	write strings X1, X2, ... (followed by a newline character) to file F. If F is omitted, write to standard output.
writes(F, X1, X2, ...)	write strings X1, X2, ... to file F.
!F	generate the lines of F
&input	standard input
&errout	standard error
&output	standard output

Structure Operations

delete(X, x)	delete element x from set X; delete element whose key is x from table X.
get(L)	delete and return the last element from the list L
pop(L)	delete and return the first element from the list L
pull(L)	delete and return the last element from the list L ????????
push(L, X)	add element X to the beginning of list L and return the new list
put(L, X)	add element X to the end of list L and return the new list
insert(S,x)	insert element x into set S
insert(T,K,V)	insert key K with value V into table T. Same as T[K] := V.
key(T)	generate the keys of the elements of table T
list(I, X)	produce a list consisting of I copies of X
set(L)	return the set consisting of the elements of the list L
sort(X)	return the elements of the set or list X sorted in a list
sort(T,I)	return the elements of the table T sorted in a list L. • If I=1 (sort on keys) or I=2 (sort on values), then L=[[key,val],[key,val],···]. • If I=3 (sort on keys) or I=4 (sort on values), then L=[key,val,key,val,···].
table(X)	return a table with default value X.
*X	number of elements in X
?X	random element from X
!X	generate the elements of X (a table or set) in some random order
!X	generate the elements of X (a list or record) from beginning to end
L1 ||| L2	concatenate lists
R.f	field f from record R
[X1,X2,...]	create a list
T[X]	value of table T whose key is X
L[I]	Ith element of list L

Control Structures

`break E`	exit loop and return E
`case E of { ...}`	produce the value of the case clause whose key is E
`every E1 do E2`	evaluate E2 for every value generated by E1
`fail`	fail the current procedure call
`if E1 then E2 else E3`	produce E2 if E1 succeeds, otherwise produce E3
`next`	go to the beginning of the enclosing loop
`not E`	if E then fail else &null
`repeat E`	evaluate E repeatedly
`until E1 do E2`	evaluate E2 until E1 succeeds
`return E`	return E from current procedure
`while E1 do E2`	evaluate E2 until E1 fails
`E1 | E2`	generate the results of E1 followed by the results of E2
`/x`	Succeeds (and produces null) if x = null. Fails otherwise.
`\x`	Succeeds and produces x if x $\neq$ null. Fails otherwise.
`&fail`	produces no result
`&null`	null value
`&trace`	if the &trace is set to a value $n > 0$, a message is produced for each procedure call/return/suspend/resume.

String Scanning

`move(i)`	advances the position by i characters. move returns the substring of the subject that is matched as a result of changing the position.
`tab(i)`	moves to position i in the subject and returns the substring between the old and new positions.
`upto(s)`	returns the position of any of the characters in s.
`many(s)`	returns the position following the longest possible substring containing only characters in s starting at the current position.
`any(c)`	succeeds if the first character in the subject string is in the cset c.
`match(t)`	succeeds if t matches the initial characters of the subject string and returns the position after the matched prt.
`&digits:`	digits between 0 to 9.
`&letters`	all letters.
`&lcase`	lower case letters.
`&ucase`	upper case letters.