Cheatography

442 Cheat Sheet

by jenwwnewnw via cheatography.com/77170/cs/18941/

Basic Syntax

`null []`	return *True* if list is empty
`'H' `elem` " Hel l o"`	return *True* if H is in the string
`head [1,2,3]`	return 1
`tail [1,2,3]`	return [2,3]
`last [1,2,3]`	return 3
`init [1,2,3]`	return [1,2]
`:t`	return the type
`fst (5,2)`	return 5
`snd (5,2)`	return 2
`1:2:3:[]`	same as [1,2,3]
`length []`	give length of list
`reverse []`	reverse the list
`[] !! n`	gives the **n**th element
`filter test []`	return everything that passes the test
`[] ++ []`	list concat-enation
`[] : []`	list concat-enation
`drop n []`	delete the first **n** element from list
`take n []`	make a new list containing just the first N element
`splitAt n []`	split list into two lists at **n**th position
`zip [a..] [0...]`	combine tow list into tuples [(a,0]..]
`map function [[]`	apply a function to all list elements

Terminology

Terminology (cont)

Type Variable	Lower case, can be of any type. e.g. fst::(**a**,**b**)->**a**
Typeclass	A sort of interface that defines some behavior. Basic type classes: Read, Show, Ord, Eq, Enum, Num. **Num** includes *Int, Integer, Float, Double.*
Higher-ordered Functions	A function that takes other functions as arguments or returns a function as result. Ex: foldl, folder,zipWith, flip.
Module	A collection of related functions, types and typecl-asses
Refere-ntial Transp-arency	An expression is called refere-ntially transparent if it can be replaced with its corres-ponding value without changing the program's behavior.
	substituting equals for equals, different from other programing languages

Type Signatures

In type signature, specific *(String)* and general *(a,b)* types can be mixed and matched.

```
concat 3:: Str ing ->S tri ng- >St rin g->
 String
const :: a-> b->a
allEqual :: (Eq a) => a -> a -> a -> Bool
(.)::( b-> c)- >(a ->b )-> a->c
```

Type Signatures (cont)

Lambda function, lead with \, then arguments, then ->, then the computation

Data Types

Haskell uses various data types, all of them starts by a capital letter:
-**Int**: Integer number with fixed precision
-**Integer**: Integer number with virtually no limits
-**Float**: Floating number
-**Bool**: Boolean. Takes two values: True or False.
-**Char**: Character. Any character in the code is placed between quotes (').
-**String**: Strings (In fact, a list of Chars).

Properties of Haskell

Pure	No side effects in functions and expressions
	No assignment operators such as ++ and =+
	I/O is an exception
	Promotes referential transp-arency
	Once x is assigned to a value, the value stays
Functional	Use recursion instead of iteration
	Allows operations on functions
Lazy	Don't do an operation unless you need the result.

Recursive Descent Parser

```
-- our parsers generally are of
type Parser [Ptree]
data Ptree = VAR String | ID
String | FCN String [Ptree]
        der iving (Show, Eq,
Read)
data Presult a = FAIL | OK a
String deriving (Show, Eq, Read)
type Parser a = String ->
Presult a
```

Polymorphic Types	Families of types. For example, (forall a)[a] is the family of types consisting of, for every type a, the type of lists of a. Lists of integers (e.g. [1,2,3]), lists of characters (['a','b','c']), even lists of lists of integers, etc., are all members of this family.

Recursive Descent Parser (cont)

```
> -- As before, we use &> and |> as AND / OR combinators on parsers
expr = variable |> fcnCall |> identifier
fcnCall = buildCall . (identifier &> skip "(" &> arguments &> skip ")")
arguments = expr &> argTail |> empty
argTail = skip "," &> expr &> argTail |> empty
identifier input = beginsWith ID Data.C-har.isLower isTailChar (dropblank input)
variable input = beginsWith VAR Data.C-har.isUpper isTailChar (dropblank input)
empty = OK [] -- empty string parser always succeeds
--------------------------------
-- UTILITY ROUTINES
-- Parse a string but don't save it as a parse tree
skip :: String -> Parser [a]
skip want input =
   let found = take (length want) input
       remainder = dropblank (drop (length want) input)
    in
       if want == found then OK [] remainder
       else FAIL
-- Build a singleton list of a function call parse tree from a list with
-- an identifier followed by list of arguments
buildCall :: Presult [Ptree] -> Presult [Ptree]
buildCall FAIL = FAIL
buildCall (OK [] _) = FAIL
buildCall (OK (ID fcn : args) remainder) = OK [FCN fcn args] remainder
-- Build a singleton list of a parse tree given the kind of tree we want
-- and the kinds of head and tail characters we want
beginsWith :: (String -> Ptree) -> (Char -> Bool) -> (Char -> Bool) -> Parser [Ptree]
```

Recursive Descent Parser (cont)

```
> beginsWith _ _ _ "" = FAIL
beginsWith builder isHead isTail (c:cs)
   | isHead c = let tail = Data.List.takeWhile isTail cs
                in OK [builder (c:tail)] (dropblank (drop (length tail) cs))
   | otherwise = FAIL
-- Remove spaces (and tabs and newlines) from head of string.
--
dropblank :: String -> String
dropblank = Data.List.dropWhile Data.C-har.isSpace
-- kind of character that makes up 2nd - end character of an id or var
--
isTailChar :: Char -> Bool
isTailChar c = Data.Char.isAlphaNum c || c == '_'
--------------------------------
-- Concatenation and alternation operators on parsers
-- (|>) is an OR/Alternation operator for parsers.
--
infixr 2 |>
(|>) :: Parser a -> Parser a -> Parser a
(p1 |> p2) input =
   case p1 input of
      m1 @ (OK _ _) -> m1 -- if p1 succeeds, just return what it did
      FAIL -> p2 input
-- (&>) is an AND/Concatenation operator for parsers
infixr 3 &>
(&>) :: Parser [a] -> Parser [a] -> Parser [a]
(p1 &> p2) input =
   case p1 input of
      FAIL -> FAIL -- p1 fails? we fail
      OK ptrees1 remain1 ->
```

Recursive Descent Parser (cont)

```
>          case p2 remain1 of -- run p2 on remaining input
              FAIL -> FAIL -- p2 fails? we fail
              OK ptrees2 remain2 -> -- both succeeded
                 OK (ptrees1 ++ ptrees2) remain2
```

Tree

```
data Tree a = Leaf a | Branch a (T
treeEq :: (Eq a) => Tree a -> Tree
treeEq (Leaf x) (Leaf y) = x == y
treeEq (Branch x1 l1 r1) (Branch x
treeEq _ _ = False
```

treeShow

```
treeShow :: Show a => Tree a -> [C
treeShow (Leaf x) = " (Leaf " ++ s
treeShow (Branch x left right)= "
"
```

Preorder via standard recursion

```
preorder :: Tree a -> [a]
preorder (Leaf x) = [x]
preorder (Branch x left right)= x
```

Tail-recursive traversal

```
preorder' :: Tree a -> [a] -> [a]
preorder' (Leaf x) xs = x : xs
preorder' (Branch r left right) xs
```

Function Syntax

```
addFour w x y z =
 let a = w + x
         b = y + a
 in z + b
------ --- --- --- --- -----
addFour w x y z =
   z + b
```

Function Syntax (cont)

```
> where
  a = w + x
  b = y + a
-----------------------
fib n
  | n < 2 = 1
  | otherwise = fib (n - 1) + fib (n - 2)
-----------------------
fib n =
  case n of
    0 -> 1
    1 -> 1
-----------------------
fib n =
  if n < 2
    then 1
    else fib (n - 1) + fib (n - 2)
-----------------------
nameReturn :: IO String
nameReturn = do putStr "What is your name? "
               name <- getLine
               putStrLn ("Pleased to meet you, " ++ name ++ "!")
               return full
```

Regex

.	Any character except new line (\n)		
\w	Word	*	0 or more
\S	Not white space	+	1 or more
\s	White space	?	0 or 1
\W	Not word	{3}	Exactly 3
\d	Digit	{3,}	3 or more
\D	Not digit	{3,5}	3, 4 or 5
\b	Word boundary	^	Beginning of String
\B	Not word boundary	$	End of String

Regex (cont)

[^]	matches characters NOT in bracket	[ ]	matches characters in brackets
|	Either Or	( )	Group
ε	Empty string containing no characters		

^ [. $ { * (\ +) | ? < >

Matecharacters need to be escaped

Currying

Currying is the process of transforming a function that takes multiple arguments in a tuple as its argument, into a function that takes just a single argument and returns another function which accepts further arguments, one by one, that the original function would receive in the rest of that tuple.

from `g :: (a, b) -> c` to `f :: a -> (b -> c)`

`f :: a -> (b -> c)` is the same as `f :: a -> b -> c`

`g (x,y) = x + y` is an uncurried function, has the type `g :: Num a => (a, a) -> a`

`h x y = x + y` is a curried addition, has the type `h :: Num c => c -> c -> c`

`curry g` can convert it to a curried function

Fold List

Foldl takes a binary operation, a starting value, and the list to fold

`foldl (-) 0 [3,5,8]` => `(((0 - 3) - 5) - 8)` => `-16`

foldl and foldr is under the type class **`Foldable`**

```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

Notes

```
head_r epeats n x = (take n x) ==
```

returns True if the first n elements of x equals

```
swap_ends [] = []
swap_ends [y] = [y]
swap_ends x = last x : (reverse (d
```

Define a function swap_ends that takes a list a elements swapped.

iterate via standard recursion

```
iterate1 n f
 | n <= 0 = id
| otherwise = f . (iterate1 (n-1)
```

iterate via foldl

```
iterate2 n f = foldl (.) id [f | i
```

```
f1a :: (b, a) -> (a, b)
f1a = \(x, y) -> (y, x)
f1b :: a -> [a] -> [[a]]
f1b = \x y -> [[x], y]
f1c :: a -> a -> [a] -> [[a]]
f1c = \x y z -> [x : z, y : z]
f1d :: (a -> Bool) -> [a] -> Int
f1d f = length . (filter f)
```

```
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
```

++ is only used for list concatenation, whereas : is used for joining element with lists

Num class does not support /, Fractional does

Pattern Matching

`(x:xs)`	head x and tail xs
`(x:3:xs)`	list where 2nd element is 3
`myData a _ c`	ignore one of the component

Pattern Matching (cont)

```
data Pattern a = P a | POr (Pattern a) (Pattern a)| PAnd (Pattern a) (Pattern a) deriving Show
match pattern [] = (False, [])
match (P x) (y : ys) = if x == y
 then (True, ys) else (False, y
: ys)
match (POr pat1 pat2) xs =case m
atch pat1 xs of
(True, leftover) -> (True, lefto
ver)
(False, _) -> match pat2 xs
match (PAnd pat1 pat2) xs =case
match pat1 xs of
 (False, _) -> (False, xs)
(True, leftover) ->case match pa
t2 leftover of
(False, _) -> (False, xs)
(True, leftover2) -> (True, left
over2)
```

Regex Examples

Natural numbers with no leading zeros except just 0

```
0 | [1-9] \d*
```

Floating point numbers w/o leading zeros

```
(0 | [1-9] \d*.\d* | . \d+)?([eE][+-]?[0-9]+))
```

Hex numbers allowing leading zeros

```
0x[0-9a-fA-F]+
```

Strings with an even #a's or number ofb's divisible by 2

```
(b*ab*a)*b*|(a*ba*ba*b)*a*
```

Match regular expressions using backtracking

Match regular expressions using backtracking (cont)

```
> match Rend str = FAIL
match Rany "" = FAIL
match Rany (c : cs) = OK [c] cs
match (Rch ch1) "" = FAIL
match (Rch ch1) (str @ (ch2 : left))
   | ch1 == ch2 = OK [ch1] left
   | otherwise = FAIL
match (Ror exp1 exp2) str =
   case match exp1 str of
      FAIL -> match exp2 str
      result1 @ (OK match1 remain1) ->
         case match exp2 str of
            FAIL -> result1
            result2 @ (OK match2 remain2) -
>
               if length match1 >= length
match2
match (Rand exp1 exp2) str =
   case match exp1 str of
      FAIL -> FAIL
      ok @ (OK match1 remain1) ->
         extend match1 (match exp2
remain1)
match (Ropt exp) str = match (Ror exp
Rnull) str
match (Rstar exp) str =
   case match exp str of
      FAIL -> OK "" str
      OK match1 remain1 ->
         if match1 == "" then OK "" str
         else
            extend match1 (match (Ror
(Rstar exp) Rnull) remain1)
extend match1 (OK match2 remain2) = OK
(match1 ++ match2) remain2
extend match1 FAIL = FAIL
-- mkAnd string = the exp that matches
each character of the string in sequence.
--
mkAnd (c : "") = Rch c
mkAnd (c : cs) = Rand (Rch c) (mkAnd cs)
```

Match regular expressions using backtracking (cont)

```
> --
mkOr (c : "") = Rch c
mkOr (c : cs) = Ror (Rch c) (mkOr cs)
```

More Examples

```
(Find out whether a list is a
palindrome)
isPali ndr ome'' :: (Eq a) =>
[a] -> Bool
isPali ndr ome'' xs = foldl
(\acc (a,b) -> if a == b then
acc else False) True input where
input = zip xs (reverse xs)
(Eliminate consec utive
duplicates of list elements)
compress :: Eq a => [a] -> [a]
compress = map head . group
(Count the leaves of a binary
tree)
countL eaves Empty = 0
countL eaves (Branch _ Empty
Empty) = 1
countL eaves (Branch _ left
right) = countL eaves left+
countL eaves right
(User- Defined Polymo rphic
Lists)
(a) Define the function foldList
which acts on user-d efined
lists just as foldr acts on
native lists.
foldList :: (a -> b -> b) -> b -
> List a -> b
foldList f init Nil = init
foldList f init (Cons x xs) = f
x (foldList f init xs)
(b) Define the function sumList
which adds up the entries in an
argument of type (List Int).
sumList :: (List Int) -> Int
sumList = foldList (+) 0
```

Lecture 11

```
data RegExp = Rnull
                      | Rend
                      | Rany
                      | Rch
Char
                      | Ror
RegExp RegExp
                      | Rand
RegExp RegExp
                      | Ropt
RegExp
                      | Rstar
RegExp
                       der -
iving (Eq, Show)
data Mresult = FAIL | OK String
String deriving (Eq, Show)
match :: RegExp -> String ->
Mresult
match Rnull str = OK " " str
match Rend " " = OK " " " "
```

```
data ParseT = STR String | LIST
[ParseT] deriving (Show, Eq,
Read)
data PResult = FAIL | OK
[ParseT] String deriving (Show,
Eq, Read)
type Parser = String -> PResult
```

Lecture 11 (cont)

```
> type TreeBuilder = [ParseT] -> ParseT --
LIST, for these trees
-- Note use of &> as AND and |> as OR
list = parse LIST (skip "(" &> list &> sublist
&> skip ")"
                |> skip "[" &> list &> sublist &>
skip "]"
                |> identifier)
sublist = (skip ",") &> list &> sublist |> empty
identifier = literal "x"
empty = OK [] -- empty string parser always
succeeds
-- expr = expr &> literal "+" &> identifier |>
empty
-------------------------------------------------------
--------------
-- UTILITY ROUTINES
-- Parse a string and make it a parse tree
literal :: String -> Parser
literal want input =
   let found = take (length want) input
      remainder = dropblank (drop (length
want) input)
    in
      if want == found then OK [STR want]
remainder
      else FAIL
-- Parse a string but don't save it as a parse
tree
skip want input =
   case literal want input of
      FAIL -> FAIL
      OK _ remain -> OK [] remain
-- Remove spaces from head of string
dropblank = Data.List.dropWhile Data.C-
har.isSpace
-----------------------------------
-- Concatenation and alternation operators
on parsers
-- (|>) is an OR/Alternation operator for
parsers.
--
infixr 2 |>
```

Lecture 11 (cont)

```
> (|>) :: Parser -> Parser -> Parser
(p1 |> p2) input =
   case p1 input of
      m1 @ (OK _ _) -> m1 -- if p1
succeeds, just return what it did
      FAIL -> p2 input
-- (&>) is an AND/Concatenation operator
for parsers
--
infixr 3 &>
(&>) :: Parser -> Parser -> Parser
(p1 &> p2) input =
   case p1 input of
      FAIL -> FAIL -- p1 fails? we fail
      OK ptrees1 remain1 ->
         case p2 remain1 of -- run p2 on
remaining input
            FAIL -> FAIL -- p2 fails? we fail
            OK ptrees2 remain2 -> -- both
succeeded
               OK (ptrees1 ++ ptrees2)
remain2
-----------------------------------------
-- Building a parse tree from list of found
parse trees
parse :: TreeBuilder -> Parser -> Parser
parse builder parser input =
   case parser input of
      FAIL -> FAIL
      (OK [] remain) -> OK [] remain
      (OK trees remain) -> OK [builder trees]
remain
```