Haskell Live

[03] Krypto Kracker

Bong Min Kim
e0327177@student.tuwien.ac.at

Christoph Spörk
christoph.spoerk@inode.at

Florian Hassanen
florian.hassanen@googlemail.com

Bernhard Urban
lewurm@gmail.com

22. Oktober 2010

Tipps & Tricks

Pattern Matching

```haskell
fkt1 :: [Integer] -> Integer
fkt1 [x] = x
fkt1 [a, b, c, d] = c
fkt1 ganzes@(erstes : rest) = erstes + sum_alternative_1
  where sum_alternative_1 = sum (erstes : rest)
        sum_alternative_2 = sum ganzes
```

```haskell
fkt2 :: Integer -> Integer -> Integer
fkt2 10 _ = 10
fkt2 x y = x + y
```

List comprehensions

```haskell
digits :: [Integer]
digits = [1, 2, 3]
chars :: [Char]
```

```
chars = ['a', 'b', 'c']   -- this is equivalent to writing chars = "abcd". why?

simple :: [Integer]
simple = [digit | digit ← digits]

mixed :: [(Char, Integer)]
mixed = [(char, digit) | char ← chars, digit ← digits];

unmixed :: [(Char, Integer)]
unmixed =
  [(char, digit)
   | index ← [0 .. 2],
     let char = chars !! index,
     let digit = digits !! index
  ]
  -- same expression as above, but different style
unmixed2 :: [(Char, Integer)]
unmixed2 =
  [(chars !! index, digits !! index)
   | index ← [0 .. 2]
  ]

nested :: [[Integer]]
nested =
  [
    [cell * 111
     | cell ← line
    ]
   | line ← listOflists
  ]
  where listOflists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

conditional :: [String]
conditional = [item
   | (char1, digit1) ← unmixed,
     (char2, digit2) ← unmixed,
     (char3, digit3) ← unmixed,
     (digit1 + 1) `mod` 3 ≡ digit2 `mod` 3,
     (digit2 + 1) `mod` 3 ≡ digit3 `mod` 3,
     let item = char1 : char2 : char3 : ""
  ]
```

Int vs. Integer (schon wieder)

Zum Beispiel hat die Funktion `length` folgende Signatur: `length :: [a] -> Int`. Anstatt `Int`, braucht man aber `Integer` als Resultattypen. Was könnte man coden?

- Typumwandlung mit `fromInteger` oder `fromIntegral` (letzteres funktioniert sogar in beide Richtungen) bei jeder Funktionsapplikation
- Funktion selber schreiben, zum Beispiel

```
mylen :: [Integer] -> Integer
mylen [] = 0
mylen (_ : xs) = 1 + mylen xs
```

- Eine Funktion die einem die Typumwanldung uebernimmt

```
len_integer :: [Integer] -> Integer
len_integer x = toInteger (length x)
```

Krypto Kracker

```
  -- functions to ease usage
run_krypto_kracker :: [[String]]
run_krypto_kracker = krypto_kracker ciphertext clearphrase


  -- input data
clearphrase = "the quick brown fox jumps over the lazy dog"
ciphertext = [
  "vtz ud xnm xugm itr pyy jttk gmv xt otgm xt xnm puk ti xnm fprxq",
  "xnm ceuob lrtzv ita hegfd tsmr xnm ypwq ktj",
  "frtjrpgguvj otvxmdxd prm iev prmvx xnmq"
  ]


  -- substitution is a mapping from a Char into another Char
type Substitution = Char -> Char


  -- initial knowledge: essentially, we have no clue
  -- [expressed by meta symbol '?']
  -- neither how to encrypt
```

```
init_encrypt_subst :: Substitution
init_encrypt_subst _ = '?'
  -- nor how to decrypt
init_decrypt_subst :: Substitution
init_decrypt_subst _ = '?'

  -- function used to add an entry to a substitution
add_entry :: Substitution → (Char, Char) → Substitution
add_entry subst (source, dest) =
  new_subst
  where new_subst x
      | x ≡ source = dest
      | otherwise = subst x

  -- test whether a character is mapped in a substitution
contains :: Substitution → Char → Bool
contains subst key = subst key ≢ '?'

  -- actual kracking happens here
  -- input params:
  --      an encryption_subst - known so far
  --      an decryption_subst - known so far
  --      a encrypted string
  --      a cleartext string
  -- returns a triple (success, encryption_subst, decryption_subst):
  --      success = True iff kracking was successful
  --          (i.e. an encryption-/decryption_subst was found)
  --      encryption_subst, subst used for encryption
  --          (only valid if success = True)
  --      decryption_subst, subst usable for decryption
  --          (only valid if success = True)
krack :: Substitution → Substitution → String → String → (Bool, Substitution, Substitution)
krack encrypt_subst decrypt_subst "" "" =
  (True, encrypt_subst, decrypt_subst)
krack encrypt_subst decrypt_subst (cipherchar : cipherstring) (clearchar : clearstring)
 | new_char_combination =
  krack new_encrypt_subst new_decrypt_subst cipherstring clearstring
 | char_combination_already_registered =
  krack encrypt_subst decrypt_subst cipherstring clearstring
 | otherwise                 =
```

```
    (False, encrypt_subst, decrypt_subst)
  where new_char_combination = new_clearchar ∧ new_cipherchar
    new_clearchar            = ¬ (encrypt_subst ‘contains‘ clearchar)
    new_cipherchar           = ¬ (decrypt_subst ‘contains‘ cipherchar)
    char_combination_already_registered = encrypt_subst clearchar ≡ cipherchar
    new_encrypt_subst        = encrypt_subst ‘add_entry‘ (clearchar, cipherchar)
    new_decrypt_subst        = decrypt_subst ‘add_entry‘ (cipherchar, clearchar)

    -- decrypts a given encrypted text using given substitution
  decrypt :: [String] → Substitution → [String]
  decrypt text subst =
    [
      [subst char
       | char ← line
      ]
     | line ← text
    ]

    -- finds all substitution
    -- given a pair of a ciphertext and a cleartext phrase
  find_substitutions :: [String] → String → [Substitution]
  find_substitutions ciphertext clearphrase =
    substs
    where substs = [subst
           | (valid, _, subst) ← tuples, valid]
      tuples = [krack init_encrypt_subst init_decrypt_subst t clearphrase
           | t ← ciphertext, length (t) ≡ length (clearphrase)]

    -- glue for find_substitutions and decrypt
  krypto_kracker :: [String] → String → [[String]]
  krypto_kracker ciphertext clearphrase =
    [decrypt ciphertext subst | subst ← substs]
    where substs = find_substitutions ciphertext clearphrase
```

Licht, mehr Licht!

Eine weitere alternative Lösung fürs letzte Haskell Live Beispiel:

```
    -- representation of switches/lights as a function mapping an index to a Bool
    -- False = light with given index is off
```

```
  -- True = light with given index is on
licht_show :: Integer → String
licht_show n =
  if licht n
  then "an"
  else "aus"

type Lightstate = Integer → Bool
licht :: Integer → Bool
licht n = final_state n ≡ True
  where
  final_state :: Lightstate
  final_state = simulate n init_state

  -- at begin each light is turned off (regardless of the index)
init_state :: Lightstate
init_state _ = False

simulate :: Integer → Lightstate → Lightstate
simulate rounds state = simulate_turnwise from_round to_round start_state
  where
  from_round :: Integer
  from_round = 1
  to_round :: Integer
  to_round    = rounds
  start_state :: Lightstate
  start_state = state

simulate_turnwise :: Integer → Integer → Lightstate → Lightstate
simulate_turnwise turn max_turns prev_state
   | turn > max_turns = prev_state
   | otherwise = simulate_turnwise (turn + 1) max_turns next_state
  where
  next_state :: Lightstate
  next_state = flip_every turn prev_state

flip_every :: Integer → Lightstate → Lightstate
flip_every intervall prev_state = next_state
  where
  next_state index = if index `mod` intervall ≡ 0
    then ¬ (prev_state index)
    else prev_state index
```