Type Inference Exercise 3 (19% of SE3E 2004 Midterm 1)

Provide **detailed derivations** of the Haskell types of the following functions:

```
swibble x y = [ ( x , y ) , ( x ++ "", y + 1) ]
```

```
swoon g h = [ g ( (1+ )∘h ) ]
```


Type Inference Exercise 3 (a) — Solution

```
swibble x y = [ ( x , y ) , ( x ++ "", y + 1) ]
```

Assuming `1` :: *n* with *Num n*, we must have *y* :: *n* because of *y* + `1`

Since "" :: *String*, we also have *x* :: *String* because of *x* ++ "" :: *String*.

Then (*x* ,*y*) :: (*String* , *Integer*), and:

swibble :: (*Num n*) ⇒ *String* → *n* → [(*String* , *n*)]

Type Inference Exercise 3 (b) — Solution

```
swoon g h = [ g ( (1+ )∘h ) ]
```

swoon :: (*Num n*) ⇒ ((*a* → *n*) → *b*) → (*a* → *n*) → [*b*]

We assume `(1+)` :: *n* → *n* with *Num n*, and because of the composition, we must have

h :: *a* → *n* for some type *a*.

Therefore, we have `( (1+ )∘h )` :: *a* → *n*,

and may assume *g* :: (*a* → *n*) → *b* for some type *b*.

Then we have `[ g ( (1+ )∘h ) ]` :: [*b*], and therefore

swoon g :: (*a* → *n*) → [*b*]

and

swoon :: *Num n* ⇒ ((*a* → *n*) → *b*) → (*a* → *n*) → [*b*].

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

```
    4*(5+6)-2                  [subtraction & mult. impossible]
=               (addition)
    4*11-2                     [subtraction impossible]
=               (multiplication)
    44-2
=               (subtraction)
    42
```

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function *f* to a number of arguments, the value of the argument at position *i* is always needed. then *f* is called **strict in its *i*-th argument**
- Therefore: If *f* is strict in its *i*-th argument, then the *i*-th argument has to be evaluated whenever a result is needed from *f*.
- Simpler: A one-argument function *f* is strict iff *f undefined* = *undefined*.
  - **Constant functions** are **non-strict**: (*const* 5) *undefined* = 5
  - Checking a list for emptyness is **strict:** *null undefined* = *undefined*
  - **List construction** is **non-strict**: *null* (*undefined* : *undefined*) = **False**
  - **Standard arithmetic operators** are **strict in both arguments**: 0 ∗ *undefined* = *undefined*

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

```
   (answer - 1) * (magic * answer - 23)
 = (42 - 1) * (magic * 42 - 23)          (answer)
 = 41 * (magic * 42 - 23)                (subtraction)
 = 41 * (7 * 42 - 23)                    (magic)
 = 41 * (294 - 23)                       (multiplication)
 = 41 * 271                              (subtraction)
 = 11111                                 (multiplication)
```


How did I find those numbers?

Easy!

```
Prelude> [ n | n <- [1 .. 400] , 11111 `mod` n == 0 ]
[1,41,271]
```

This is a **list comprehension**:

- return all `n`
- where `n` is taken from then list `[1 .. 400]`
- and a result is returned only if `n` divides `11111`.

Conditional Expressions

```
Prelude> if 11111 `mod` 41 == 0 then 11111 `div` 41
else 5
271
```

The pattern is:

if *condition* **then** *expression1* **else** *expression2*

– If the condition evaluates to **True**, the conditional expression evaluates to the value of *expression1.*

– If the condition evaluates to **False**, the conditional expression evaluates to the value of *expression2.*

***Therefore:* "if _ then _ else"** is **strict in the condition**.

In C: (*condition* ? *expression1* : *expression2*)

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
 = 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
 = 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * if True then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * 1
 = 3 * 2 * 1
 = 3 * 2
 = 6
```


Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
   fact 3
 = 3 * fact (3-1)              (fact n)
 = 3 * fact 2                  (determining which fact rule matches)
 = 3 * (2 * fact (2-1))        (fact n)
 = 3 * (2 * fact 1)            (determining which fact rule matches)
 = 3 * (2 * (1 * fact (1-1)))  (fact n)
 = 3 * (2 * (1 * fact 0))      (determining which fact rule matches)
 = 3 * (2 * (1 * 1))           (fact 0)
 = 3 * (2 * 1)                 (multiplication)
 = 3 * 2                       (multiplication)
 = 6                           (multiplication)
```


Expression Evaluation Exercise 1 (16% of SE3E 2005 MT 1)

Let the following Haskell definition be given:

```
m :: Integer -> Integer -> Integer
m 1 y = 7
m x y = x * m (x - 1) (m (x + 1) (2 * y))
```

Simulate Haskell evaluation for the following expression, i.e., write down **the complete sequence of intermediate expressions**:

```
m (8 - 5) (6 * 9)
```

Note: You may introduce ***abbreviations for repeated subexpressions***, or use ***repetition marks for material that is unchanged from the previous line***.

Side Effects and Haskell

- Haskell is **pure:**
  - Evaluating expressions has **no side-effects**
  - Expressions are evaluated only for obtaining their **values**
- But sometimes we want our programs to affect the real world (printing, controlling a robot, drawing a picture, etc).

How do we reconcile these two aspects?

In Haskell, certain "pure values" are "worldly actions" that can be ***performed***

– **Types:** An expression with type *IO a* has as its value a **computation** (in the ***IO-monad***) that can be understood as returning a value of type *a*.

  Alternative explanation: An expression with type *IO a* has possible ***actions*** associated with its execution, while returning a value of type *a*

– **Syntax:** The **do** syntax sequences several actions (using layout)

The do Syntax

- *readFile* "/etc/passwd" :: *IO String* is an action.
- We use the **do** syntax to bind the result of that action to the variable *s*, and sequence this action with other actions that depend on *s*:

```
main = do
  s ← readFile "/etc/passwd"
  putStrLn $ "/etc/passwd has " ++ shows (length s) " characters"
  let logins = map (takeWhile (':' ≠)) $ lines s
  putStrLn $ "There are " ++ shows (length logins) " logins"
  let funny = filter (all (`notElem` "AEIOUaeiou")) logins
  putStrLn $ "Funny logins: " ++ concatMap (' ' :) funny
```

- Inside **do**, one may write **let** without **in**.

When IO Actions are Performed

An expression with type *IO a* has as its value a **computation** that, when performed, may return a value of type *a*.

- A value of type *IO a* is an **action**, but it is still a *value:* it will **only** have an **effect** when it is **performed**.
- In Haskell, a program's value is the value of the variable *main* in the module *Main*.

  That value has to have type *IO a*.

  It will be **performed** upon execution of the program.
- In Hugs and GHCi, you can type any expression to the prompt.

  If the expression has type *IO a* it will be performed; otherwise its value will be printed on the display.

Predefined IO Actions

```
-- get one character from keyboard
getChar :: IO Char

-- write one character to terminal
putChar :: Char → IO ()

-- write a string to terminal (without/with adding a newline)
putStr, putStrLn :: String → IO ()

-- get a whole line from keyboard
getLine :: IO String

-- read a file as a String
readFile :: FilePath → IO String

-- write a String to a file
writeFile :: FilePath → String → IO ()
```


IO Example

```
module Main ( main ) where  -- this is the default module header

main = do
  line ← getLine
  let ws = words line
  case ws of
    [] → return ()
    _  → do
      putStrLn ("You entered " ++ show (length ws) ++ " words")
      main
```

Compile and run:

```
ghc --make -o WC WC.hs
./WC
```

Another *IO* Example

```
import qualified System                                   -- Cat.hs

main = do
  args ← System.getArgs
  putStrLn (shows (length args) " arguments")
  let (flags, files) = span (("-" ≡) ∘ take 1) args
  print flags
  mapM (λ file → readFile file >>= putStrLn) files
```

Compile and run:

```
ghc --make -o Cat Cat.hs
./Cat -flag1 -q -v -flag4 file1 qwerty -what file4
```