FUNCTIONAL PROGRAMMING
NO.10 MODULE

Tatsuya Hagino
hagino@sfc.keio.ac.jp

Slide URL

https://vu5.sfc.keio.ac.jp/slide/

Module

- Module is a collection of the following entities:
  - variables
  - type constructors
  - data constructors
  - field labels
  - type classes
  - class methods

- Similar to packages in Java

- Name spaces are divided into modules.
  - Each name is unique in each module.
  - Different modules may have the same name entities.

Some Standard Modules in Haskell

module	description
`Prelude`	basic functions, types and classes
`Data.Ratio`	rational numbers
`Data.Complex`	complex numbers
`Numeric`	numerical values
`Data.Ix`	mapping between values and integers (used for array indexes)
`Data.Array`	arrays
`Data.List`	utilities related to lists
`Data.Maybe`	Maybe monad
`Data.Char`	utilities related to characters
`Control.Monad`	utilities related to monads
`System.IO`	input and output
`System.Directory`	directory manipulation
`System.Environment`	command line arguments, environment variables and others
`Data.Time`	data and time

See https://downloads.haskell.org/~ghc/latest/docs/html/libraries/

module Declaration

```
module name where
...
```

- Declaring module ***name***
  - Module name needs to start with a large alphabet.
  - Multiple names may be concatenated with '`.`'.
  - All the entities defined in the module are exported.

example

```
module FileUtils where
data SomeType = ConsA String | ConsB Int
makePath = ....
forceRemove = ...
```

Export Limited Entities

- By default, all the entities are exported.

```
module FileUtils (makePath, forceRemove) where
```

- Specify export list.
- Only **`makePath`** and **`forceRemove`** are exported and the others are hidden.

```
module FileUtils (joinPath, (+), concatPath) where
```

- Any entities except data constructors may be listed in the export list.
- Need parenthesizes for exporting binary operators.

Export Data Constructors

- Data constructors are not directory exported by themselves.
- Need to be exported with the data type.

```
module AnyModule (SomeType(ConsA), a, b, c) where
data someType = ConsA String | ConsB Int
```

- Exports data constructor **`ConsA`** along with data type **`SomeType`**.
- **`consB`** is not exported.

```
module AnyModule (SomeType(ConsA, ConsB), a, b, c) where
```

- Exports both **`ConsA`** and **`ConsB`**.

```
module AnyModule (SomeType(..), a, b, c) where
```

- All the data constructors and field labels for **`SomeType`** are exported.

Export Modules

- Imported modules may be exported as a whole.

```
module LineParser
    (module Text.ParserCombinators.Parsec.Prim,
     LineParser, indented, blank, firstChar, anyLine) where

import Text.ParserCombinators.Parsec.Prim
```

- All the entities defined in **`Text.ParserCombinators.Parsec.Prim`** are exported.

Main Module

- If the file does not start with module declaration, the following module declaration is assumed automatically.

```
module Main(main) where
```

- The module is named **`Main`**.
- Exports entity **`main`**.
- The type of **`main`** needs to be **`(IO a)`**.
- **`main`** becomes the entry point of the program.

import Declaration

- In order to use entities defined in other modules, they need to be imported.

```
import Text.Regex
```

- By default all the entities are imported.
- Specify import list to limit the entities imported.
  - Data constructors need to be specified with data type.

```
import Text.Regex(mkRegex, matchRegex)
```

- Specify entities not to import.

```
import Monad hiding (join)
```

- Imports entities defined in module **`Monad`** except **`join`**.

Qualified Names

- Entities may be specified fully with their module name.

```
moduleName.entityName
```

- Imported entities may be used as its qualified names as well as without module names.

```
import Text.Regex(mkRegex)
... mkRegex ...
... Text.Regex.mkRegex ...
```

- Use **qualified** syntax to import only qualified names.
  - Avoid name conflict.

```
import qualified Text.Regex
```

- Use **as** syntax to give alias to module name.

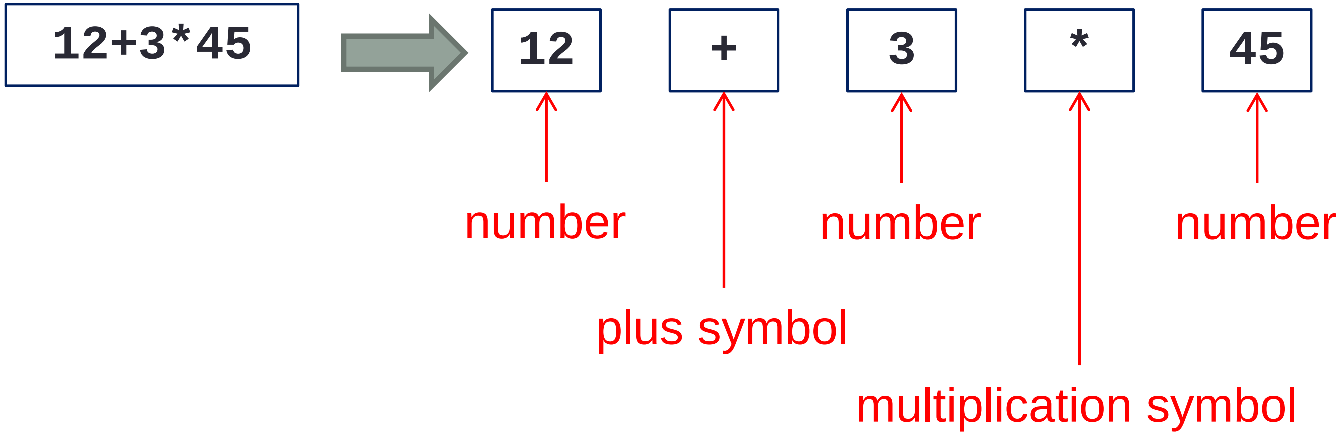
```
import qualified Distribution.Simple.GHCPackageConfig as Conf
```

Calculator Example

- Try writing a calculator program.

```
12+3*45 ⇒ 147
(1+2)*(3+4) ⇒ 21
```

- First, divide an input string into a token list.



Define Token as Data Type

```
data Token = Num Int | Add | Sub | Mul | Div
```

- Token is either a number or a symbol.

```
tokens::String -> [Token]
tokens [] = []
tokens ('+':cs) = Add:(tokens cs)
tokens ('-':cs) = Sub:(tokens cs)
tokens ('*':cs) = Mul:(tokens cs)
tokens ('/':cs) = Div:(tokens cs)
tokens (c:cs) | isDigit c = let (ds,rs) = span isDigit (c:cs)
                            in Num(read ds):(tokens rs)
              | otherwise = tokens cs
```

- **span** returns the longest prefix which satisfies the condition and the rest of the list.
  - `span :: (a -> Bool) -> [a] -> ([a], [a])`
  - `span (< 3) [1,2,3,4,1,2,3,4] = ([1,2],[3,4,1,2,3,4])`
  - `span (< 9) [1,2,3] = ([1,2,3],[])`
  - `span (< 0) [1,2,3] = ([],[1,2,3])`

Exercise 10-1

- Create a Token module and write a test program to use it.

Token.hs

The file name needs to be the same as module name.

```
module Token(Token(..),tokens) where

import Data.Char

data Token = Num Int | Add | Sub | Mul | Div
             deriving Show

tokens::String -> [Token]
tokens [] = []
tokens ('+':cs) = Add:(tokens cs)
tokens ('-':cs) = Sub:(tokens cs)
tokens ('*':cs) = Mul:(tokens cs)
tokens ('/':cs) = Div:(tokens cs)
tokens (c:cs) | isDigit c = let (ds,rs) = span isDigit (c:cs)
                            in Num(read ds):(tokens rs)
              | otherwise = tokens cs
```

tokenTest.hs

```
import Token

main = do cs <- getContents
          putStr $ unlines $ map (unwords . (map show) . tokens) $ lines cs
```

Exercise 10-2

- Write a calculator program by evaluating a token list from left to right.
  - Please add other arithmetic operations.

calc.hs

```
import Token

calc::[Token] -> Int
calc [Num x] = x
calc (Num x:Add:Num y:ts) = calc (Num (x+y):ts)
....

main = do cs <- getContents
          putStr $ unlines $ map (show . calc .tokens) $ lines cs
```

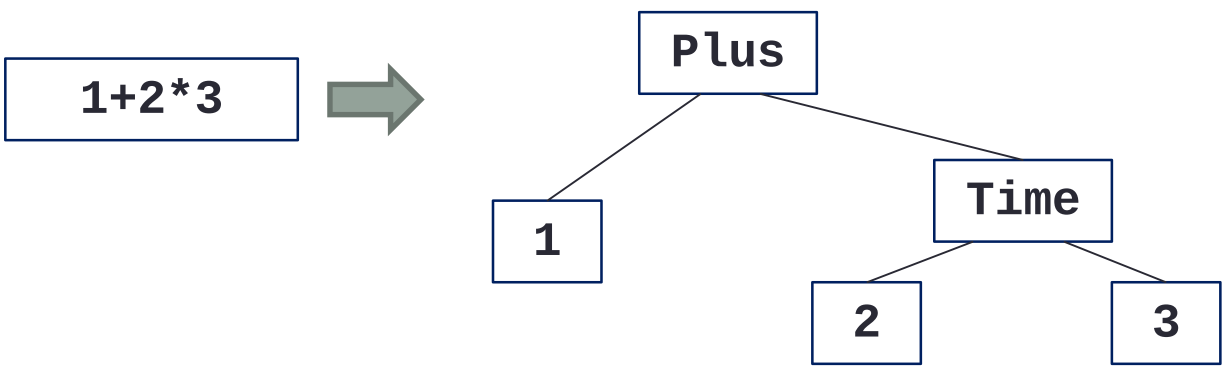
- **Example**

```
% stack ghc calc.hs
...
% ./calc
1+2
3
1+2+3+4+5+6+7+8+9
45
1+2*3-4/5
1
```

Create Parse Tree

- In order to interpret **`1+2*3`** as **`1+(2*3)`**, need to create a parse tree before calculating the result.
- Define a parse tree as data type.

```
data ParseTree = Number Int |
                 Plus ParseTree ParseTree |
                 Minus ParseTree ParseTree |
                 Time ParseTree ParseTree |
                 Divide ParseTree ParseTree
```



Parser

- Parser parses a token list and creates a parse tree.
- Parser has the following type:

```
[Token] -> (ParseTree, [Token])
```

  - Given a token list, returns a parse tree and the remaining token list.

- Expression syntax in BNF:

```
expr ::= term (("+" | "-") term)*
term ::= factor (("*" | "/") factor)*
factor ::= number | "(" expr ")"
```

```
[Num 1, Mul, Num 2, Add, Num 3]
```

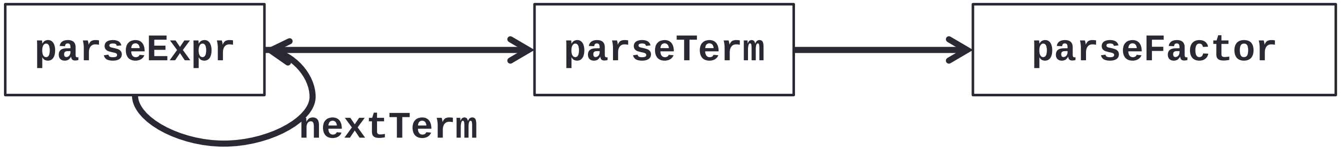
```
(Time(Number 1)(Number 2),[Add, Num 3])
```

term parser

Exercise 10-3

- Write a parser for add.

```
import Token

data ParseTree = ... deriving Show

type Parser = [Token] -> (ParseTree, [Token])

parseFactor::Parser
parseFactor(Num x:l) = (Number x, l)

parseTerm::Parser
parseTerm l = parseFactor l

parseExpr::Parser
parseExpr l = nextTerm $ parseTerm l
  where nextTerm(p1, Add:l1) = let (p2, l2) = parseTerm l1
                               in nextTerm(Plus p1 p2, l2)
        nextTerm x = x

main = do cs <- getContents
          putStr $ unlines $ map (show . fst . parseExpr .tokens) $ lines cs
```



How parseExpr works.

```
parseExpr [Num 1,Add,Num 2,Add,Num 3]
⇒ nextTerm $ parseTerm [Num 1,Add,Num 2,Add,Num 3]
⇒ nextTerm (Number 1,[Add,Num 2,Add,Num 3])
⇒ let (p2,l2) = parseTerm [Num 2,Add,Num 3]
   in nextTerm(Plus(Number 1) p2, l2)
⇒ let (p2,l2) = (Number 2,[Add,Num 3])
   in nextTerm(Plus(Number 1) p2, l2)
⇒ nextTerm(Plus(Number 1)(Number 2),[Add,Num 3])
⇒ let (p2,l2) = parseTerm [Num 3] in
   nextTerm(Plus(Plus(Number 1)(Number 2)) p2,l2)
⇒ let (p2,l2) = (Number 3,[])
   in nextTerm(Plus(Plus(Number 1)(Number 2)) p2,l2)
⇒ nextTerm(Plus(Plus(Number 1)(Number 2))(Number 3),[])
⇒ (Plus(Plus(Number 1)(Number 2))(Number 3),[])
```

Evaluate an Expression

- Evaluate an parse tree to calculate the result.

```
eval::ParseTree -> Int
eval(Number x) = x
eval(Plus p1 p2) = eval p1 + eval p2
eval(Minus p1 p2) = ...
eval(Times p1 p2) = ...
eval(Divide p1 p2) = ...
```

```
main = do cs <- getContents
          putStr $
            unlines $
            map (show . eval . fst . parseExpr . tokens) $
            lines cs
```

Exercise 10-4

- Handle other arithmetic operations.
  - Ignore spaces.
  - Handle parenthesizes (add **LPar** and **RPar** to **Token**)
  - **+** and **-** are binary as well as unary operators.
  - Expand **Tokan.hs** in **calc.hs** for submitting your homework.

calc.hs

```
import Data.Char

data Token = Num Int | Add | Sub | Mul | Div | LPar | RPar

tokens::String -> [Token]
tokens = ...

data ParseTree = ...
type Parser = [Token] -> (ParseTree, [Token])

parseFactor = ...
parseTerm = ...
parseExpr = ...

eval::parseTree -> Int
eval = ...

main = do cs <- getContents
          putStr $ unlines $ map (show . eval . fst . parseExpr . tokens) $ lines cs
```

Example

```
% ./calc
2 + 3   *  4
14
3 / 4 * 5
0
10 - 6 - 2 * 2
0
(1 + 2)*(5  - 3) / 3
2
- 3 * + 2
-6
```