Lecture 4 – Types

- Every language has types – only difference is when to check them.
  - Check types before runtime: static language
  - Check types during runtime: dynamic language
  - Static language has a compiler: part of its job is to check (and infer) types.
- A type is a set of values – the (declared/inferred) type of an expression specifies its range of values.

Primitive and constructed types

- Primitive types are predefined by the language:

```
   Int, Char, Bool, Integer, Float, Double
```

- Constructed types are defined by composing type constructors and primitive types:

```
   [Int]   (Int, Bool)   Int -> Bool
```

- Type constructors include

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

- String is a constructed type: [Char]
- Find Haskell type with `:t`.

Type annotation

Every declaration can be annotated with a type.

```haskell
x :: Int
x = 1

x' :: Integer
x' = 1

fact :: Int -> Int
fact 0 = 1
fact n = n * fact(n-1)

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

Type variable and polymorphism

A function type that contains type variables is a polytype.
The type variables in a polytype can be instantiated with other types.

```
f x = fst x

:t f    -- f :: (a, b) -> a

f (1, True) -- evaluates to 1

f (True, 1) -- evaluates to True
```

Type variable and polymorphism

Functions with polytypes allow polymorphism.

```
map :: (a -> b) -> [a] -> [b]

foldl :: (b -> a -> b) -> b -> [a] -> b

foldr :: (a -> b -> b) -> b -> [a] -> b

map (\x -> x + 1) :: [Int] -> [Int]

foldl (\c e -> c && e > 0) :: Bool -> [Int] -> Bool

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

Algebraic Data Type (ADT)

We can define new type constructors and data constructors using algebraic data types.

```haskell
data Coin = Head | Tail deriving (Show)

t: Head -- Coin

coinFlip Head = Tail
coinFlip Tail = Head

t: coinFlip -- Coin -> Coin
```

`Coin` is a type constructor (or a type if no parameters)
`Head` and `Tail` are data constructors (or data if no parameters).
`deriving (Show)` auto-derives functions for `Show`ing the data.

ADT for Enumeration

```haskell
data WeekDay = Tuesday | OpenDay | Thursday
            | MeetingDay | BlurDays deriving (Show)

nextDay Tuesday = OpenDay
nextDay OpenDay = Thursday
nextDay Thursday = MeetingDay
nextDay MeetingDay = BlurDays
nextDay BlurDays = Tuesday

:t nextDay -- WeekDay -> WeekDay
```

ADT for Enumeration

```haskell
data WeekDay = Tuesday | OpenDay | Thursday
            | MeetingDay | BlurDays deriving (Show, Eq)

canWeMeet day
   | day == OpenDay = True
   | otherwise = False
```

In order to use == on WeekDay/DayTime typed data, we need to derive operators of Eq class.

ADT for Enumeration

```
data WeekDay = Tuesday | OpenDay | Thursday
            | MeetingDay | BlurDays deriving (Show, Eq)

data DayTime = Morning | Afternoon | Night deriving (Eq)

canWeMeet day time
   | day == OpenDay = True
   | day == Tuesday && time == Morning = True
   | day == Thursday && time == Morning = True
   | otherwise = False
```

In order to use == on WeekDay typed data, we need to derive operators of Eq class.

ADT for Wrapped Data

Data constructor can wrap parameters.

```haskell
data MeetTime = Meet WeekDay DayTime deriving (Show)

:t Meet Tuesday Morning    -- MeetTime

schedule day time
   | canWeMeet day time = Meet day time
   | otherwise = schedule (nextDay day) time

schedule Tuesday Morning    -- Meet Tuesday Morning
schedule Tuesday Afternoon  -- Meet OpenDay Afternoon
schedule Thursday Afternoon -- Meet OpenDay Afternoon
```

ADT for Wrapped Data

A data type can be used to create a distinguished value.
WeekNum is more than just a number.

```haskell
data WeekNum = Week Int deriving (Show, Eq)
nextWeek (Week n) = Week (n+1)

data MeetTime = Meet WeekNum WeekDay DayTime deriving (Show, Eq)

schedule (Meet week day time)
   | canWeMeet day time = Meet week day time
   | otherwise = schedule meet'

   where meet' = Meet week' day' time
         day' = nextDay day
         week' = if (day' /= day && day' == Tuesday)
                 then nextWeek week
                 else week

m1 = Meet (Week 1) Thursday Afternoon
schedule m1     -- Meet (Week 2) OpenDay Afternoon
```

Mix builtin types with ADT

We can represent a schedule as a list of MeetTime values.

```haskell
schedule meetings meet
   | canMeet && noConflict = ((meet:meetings), meet)
   | otherwise = schedule meetings meet'

   where (Meet week day time) = meet

         canMeet = canWeMeet day time
         noConflict = not (meet `elem` meetings)

         day' = nextDay day
         week' = if (day' /= day && day' == Tuesday)
                 then nextWeek week
                 else week
         meet' = Meet week' day' time
```

Mix builtin types with ADT

We can represent a schedule as a list of MeetTime values.

```
showSchedule s = foldl (++) "" $ map (\m -> show m ++ ";\n") s

m1 = Meet (Week 1) Tuesday Afternoon
m2 = Meet (Week 1) OpenDay Morning
m3 = Meet (Week 1) Thursday Morning
(s1, m1') = schedule [] m1
(s2, m2') = schedule s1 m2
(s3, m3') = schedule s2 m3
putStrLn $ "current schedule is: \n" ++ showSchedule s3

-- current schedule is:
-- Meet (Week 1) Thursday Morning;
-- Meet (Week 1) OpenDay Morning;
-- Meet (Week 1) OpenDay Afternoon;
```

Order meetings

Define insert function to ensure that meetings are ordered.

```haskell
insertMeet meet [] = [meet]
insertMeet meet (a:b)
    | meet <= a = meet:a:b
    | otherwise = a : insertMeet meet b
```

However, to compare meet using $\leq$, we have to derive `Ord` instance for MeetTime.

Order meetings

All types involved in MeetTime has to be instances of `Ord`.

```haskell
data WeekDay = Tuesday | OpenDay | Thursday
             | MeetingDay | BlurDays deriving (Show, Eq, Ord)

data DayTime = Morning | Afternoon | Night
                                     deriving (Show, Eq, Ord)

data WeekNum = Week Int deriving (Show, Eq, Ord)

data MeetTime = Meet WeekNum WeekDay DayTime
                                     deriving (Show, Eq, Ord)
```

Order meetings

Revise schedule function to insert meet to the current schedule.

```haskell
schedule meetings meet
   | canMeet && noConflict = (insertMeeting meet meetings, meet)
   | otherwise = schedule meetings meet'

   where (Meet week day time) = meet

         canMeet = canWeMeet day time
         noConflict = not (meet `elem` meetings)

         day' = nextDay day
         week' = if (day' /= day && day' == Tuesday)
                 then nextWeek week
                 else week
         meet' = Meet week' day' time
```

Order meetings

Now the scheduled meetings are ordered.

```
showSchedule s = foldl (++) "" $ map (\m -> show m ++ ";\n") s

m1 = Meet (Week 1) Tuesday Afternoon
m2 = Meet (Week 1) OpenDay Morning
m3 = Meet (Week 1) Thursday Morning
(s1, m1') = schedule [] m1
(s2, m2') = schedule s1 m2
(s3, m3') = schedule s2 m3
putStrLn $ "current schedule is: \n" ++ showSchedule s3

-- current schedule is:
-- Meet (Week 1) OpenDay Morning;
-- Meet (Week 1) OpenDay Afternoon;
-- Meet (Week 1) Thursday Morning;
```

Recursive ADT

ADT (Algebraic Data Type) can be defined recursively.
An event type can be defined as a labelled tree with two types of nodes:

- Tag node that has tag string and a list of events
- One node that has a meeting time.

```haskell
data Event = Tag String [Event]
           | One MeetTime        deriving (Show)
```

We can represent a person's calendar using the type `[Event]`

Type alias

Type alias can improve readability and clarify intention.
The calendar type below is an alias of the type [Event]

```haskell
type Calendar = [Event]

data Event = Tag String Calendar
           | One MeetTime           deriving (Show)
```

Work with recursive ADT

The getMeetings function extracts a list of meetings from a calendar.

```haskell
type Calendar = [Event]

data Event = Tag String Calendar
           | One MeetTime           deriving (Show)

-- convert a calendar to a list of meetings
getMeetings calendar = foldr (++) [] $ map get calendar
    where get (One meet) = [meet]
          get (Tag _ calendar') = getMeetings calendar'
```

Work with recursive ADT

The makeEvent function creates an event from a list of tags and a meeting time.

```
type Calendar = [Event]

data Event = Tag String Calendar
           | One MeetTime           deriving (Show)

-- make an event from a list of tags and a meeting time
makeEvent [] meet = One meet
makeEvent (tag:tags) meet = Tag tag [makeEvent tags meet]

m1 = Meet (Week 1) Tuesday Afternoon
makeEvent ["Courses", "790", "Bob"] m1 -- evaluates to
-- Tag "Courses"
--   [Tag "790"
--      [Tag "Bob"
--         [One (Meet (Week 1) Tuesday Afternoon)]]]
```

Work with recursive ADT

The insert function inserts an event (denoted by a list of tags and meeting time) to a calendar.

```haskell
type Calendar = [Event]

data Event = Tag String Calendar
           | One MeetTime           deriving (Show)

-- insert an event to a calendar
insert [] tags meet = [makeEvent tags meet]

insert ((One _) : events) tags meet = insert events tags meet

insert ((Tag tag calendar): events) (tag':tags') meet

 | tag == tag' = (Tag tag $ insert calendar tags' meet) : events

 | otherwise = event : (insert events tags meet)

   where event = (Tag tag calendar)
         tags = (tag' : tags')
```

Work with recursive ADT

The addEvent function adds an event to a calendar (also ensures no time conflict).

```haskell
type Calendar = [Event]

data Event = Tag String Calendar
           | One MeetTime           deriving (Show)

-- calendar is a list of events
-- tags is a list, where each tag is a string
-- meet is a meeting time
-- calendar' is the new list of events
-- meet' is the scheduled meeting time
addEvent calendar tags meet = (calendar', meet')
     where meetings = getMeetings calendar
           (_, meet') = schedule meetings meet
           calendar' = insert calendar tags meet'
```

Work with recursive ADT

The showCalendar function prints each event in a calendar in a separate line and use tabs to align events of the same depth in the calendar.

```haskell
type Calendar = [Event]

data Event = Tag String Calendar
           | One MeetTime            deriving (Show)

-- format calendar
showCalendar n calendar =
    foldl (++) "" $ map (\e -> show' n e ++ "\n") calendar

  where show' n (One m) = tabs n ++ show m

        show' n (Tag tag lst) =
           tabs n ++ tag ++ "\n" ++ (showCalendar (n+1) lst)

        tabs n = take n $ repeat '\t'
```

Work with recursive ADT

```haskell
m1 = Meet (Week 1) Tuesday Afternoon
m2 = Meet (Week 1) OpenDay Morning
m3 = Meet (Week 1) Thursday Morning
(c1, _) = addEvent [] ["Research", "Adam"] m1
(c2, _) = addEvent c1 ["Research", "Alan"] m2
(c3, _) = addEvent c2 ["Courses", "790", "Bob"] m3
(c4, _) = addEvent c3 ["Courses", "431", "Carol"] m3
putStrLn $ showCalendar 0 c4

-- Research
--      Adam
--              Meet (Week 1) OpenDay Afternoon
--
--      Alan
--              Meet (Week 1) OpenDay Morning
--
-- Courses
--      790
--              Bob
--                      Meet (Week 1) Thursday Morning
--
--      431
--              Carol
--                      Meet (Week 2) Tuesday Morning
```