Just For Func!

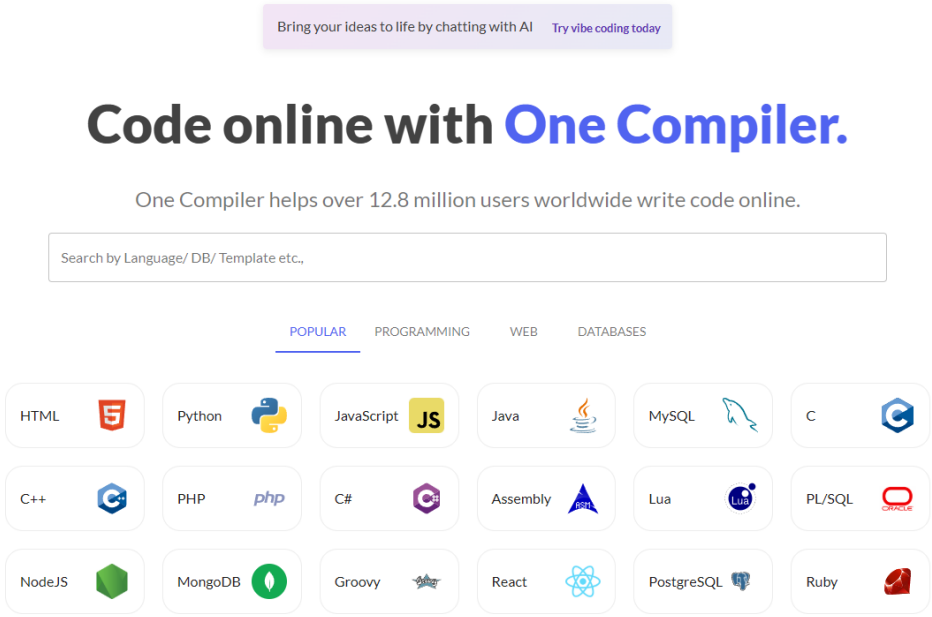
프로그래밍 언어 패러다임

프로그래밍 패러다임

- 프로그래밍 패러다임(Programming Paradigm)
- 프로그램의 상태/제어 흐름/자료 추상화등을 규정하는 설계 원칙의 형식적 체계
- 명령형 프로그래밍(Imperative Programming)
  - 문제를 해결하는 명령(절차)를 기술하는 방식의 프로그래밍
  - Ex) C, Pascal, Ada, Python 등
- 함수형 프로그래밍(Functional Programming)
  - 프로그램의 계산 과정을 수학 함수 형태로 프로그래밍
  - Ex) Lisp, Scheme, ML, Haskell 등
- 논리 프로그래밍(Logic Programming)
  - 정형 논리(Formal logic)를 기반으로 한 프로그래밍
  - Ex) Prolog 등
- 객체지향 프로그래밍(Object-Oriented Programming)
  - 객체 개념을 기반으로 하는 프로그래밍
  - Ex) C++, Java, C#, Objective-C, Swift 등

실습 환경

- OneCompiler
- 클라우드 기반 온라인 IDE(통합 개발 환경) & 컴파일러 서비스
- 웹 브라우저만 있으면 설치 없이 다양한 언어 코드를 실행 가능
  - Ex) C/C++, Java, Python, JavaScript(Node.js), Go, Rust, Kotlin, PHP, Ruby, Swift 등



명령형 프로그래밍(1)

- 명령형 프로그래밍(Imperative Programming)
- 메모리 상태를 명령의 순차적 실행으로 변경하며 목표를 달성하는 방식
  - 폰 노이만 모델에 직접적으로 적용
  - 현재 상태 → 명령 실행 → 다음 상태의 연쇄
- 장점
  - 직관적 실행 모델로 순차적 사고와 맞아 학습 진입 장벽이 낮음
  - 대다수의 언어가 사용하는 패러다임으로 컴파일러/디버거/프로파일러/라이브러리 등이 풍부.
- 단점
  - 하나의 변수에 대해서 여러 이름이나 포인터가 동일한 메모리 위치를 참조 가능하여 한 부분의 변경이 의도하지 않게 프로그램의 다른 부분에서 변경(Aliasing 문제)
  - 스레드 간 비동기적 접근 시 경쟁 조건(race condition), 교착 상태(deadlock) 같은 문제가 발생
  - 참조 투명성(Referential transparency)이 성립하지 않기 때문에, 동일한 표현식이라도 실행 시점의 상태에 따라 다른 결과를 낼 수 있어 검증이나 디버깅이 어려움

명령형 프로그래밍(2)

- 명령형/절차적/구조적 프로그래밍의 차이
- 절차적 프로그래밍(Procedural Programming)
  - 공통적으로 사용되는 코드를 프로시저나 함수로 모듈화
- 구조적 프로그래밍(Structured Programming)
  - 순차/선택/반복의 제어구조로 프로그램 구성
  - GOTO문을 지양
- 대부분의 현대 명령형 프로그래밍은 절차적이고 구조적임
  - Ex) 초창기 언어들(BASIC, FORTRAN 등)은 명령형이지만 구조적이지 않음

함수형 프로그래밍

- 함수형 프로그래밍(Functional Programming)
- 불변(Immutable) 데이터와 순수 함수(Pure Function) 의 조합으로 프로그램을 구성
  - 상태 변경 없이 입력 → 출력의 수학적 함수에 가깝게 모델링
  - 현재 상태에 의존/변경 X → 참조 투명성(Referential Transparency) 보장
- 장점
  - 예측 가능 & 검증 용이: 같은 입력 → 항상 같은 출력, 단위 테스트/증명 쉬움
  - 동시성 친화적: 공유 가변 상태가 없어 Race condition 관리 부담 감소
  - 구성 가능성: 일급 함수/고차 함수로 재사용 및 조합이 쉬움 (map, filter, fold 등)
  - 버그 감소: 부수 효과(Side effect) 최소화로 디버깅 단순
  - 지연 평가(Lazy evaluation) 지원 언어에서는 필요 연산만 수행
- 단점
  - 성능/메모리 오버헤드 가능: 불변 구조 복사, GC 부담, 박싱/언박싱 비용
  - 재귀 중심 사고의 부담: 꼬리재귀 최적화 미지원 시 스택 위험, 루프보다 직관이 떨어질 수 있음
  - 기존 명령형 라이브러리와의 상호운용 비용: 인터페이스 어댑터 필요

Haskell의 기초(1)

- Haskell
- 1990년 Haskell 1.0 공개, 논리학자 Haskell B. Curry 이름에서 유래
- 순수 함수형, 지연 평가(Lazy Evaluation), 강한 정적 타입을 핵심으로 하는 언어
- 주요 특징
  - 불변성(Immutable): 값 변경 대신 새 값 생성시켜 사이드이펙트 감소
  - 강한 정적 타입 & 타입 추론: 컴파일 타임 안전성, 대부분 타입 선언 없이도 추론
  - 고차 함수 & 함수 합성: map, fold, .(합성) 등으로 선언적 표현
  - 순수성 & 효과 모델링: IO, Maybe, Either, State 등 모나드로 부작용 명시
  - 지연 평가: 필요한 순간에만 계산 → 무한 리스트 등 표현력 향상
  - 동시성/병렬성: 경량 스레드, STM(Software Transactional Memory) 지원
- 실습
  - https://onecompiler.com/haskell

Haskell의 기초(2)

- 주의해야 할 예약어(or 키워드)

예약어	설명	예	Java와의 비교(
module	모듈 정의	module MyModule (MyModule.hs를 정의)	package, .java 파일
import	다른 모듈을 가져올 때	import Data.List	import java.util.Arrays;
where	함수 내의 지역 변수/함수를 정의	area r = pi * r * r where pi = 3.14	double area(double r) { final double pi = 3.14; }
let ... in	지역 바인딩 (in 이후에서만 사용 가능)	let x = 3 in x + 1 (x는 변수가 아님)	final int x = 3 + 1;
data	사용자 정의 자료형	data Color = Red | Blue	enum Color { RED, BLUE }
type	타입 별칭 정의	type Name = String	interface Name extends CharSequence {}

class	타입 클래스 정의
	(객체지향의 클래스가 아님)
		class Eq a where (==) :: a -> a -> Bool	interface Eq<T> {
			boolean equal(T a, T b);
			}

do	순차적인 I/O 또는 모나드 연산	main = do input <- getLine let num = read input :: Int print (num * 2)	Scanner sc = new Scanner(System.in); int num = Integer.parseInt(sc.nextLine()); System.out.println(num * 2);

Haskell의 기초(3)

- 명명 규칙
- 상수명(변수명), 함수명 규칙
  - 소문자로 시작하고, 알파벳, 숫자, ' 사용 가능
  - Haskell에서 변수는 실질적으로 상수
- 타입 이름 규칙
  - 대문자로 시작하고, 알파벳, 숫자, ' 사용 가능
  - Ex) Int, Bool
- 모듈 이름 규칙
  - 대문자로 시작하고, 점(.)으로 계층 구분
  - Ex) Data.List

```
main = do
  print "abc"
  print 13  -- 정상: do 구문 내의 print들의 들여쓰기가 같음

main = do
  print "abc"
    print 13  -- 오류: print들의 들여쓰기가 다름

main = do { print "abc"; print 13; }
-- 정상: { }으로 C처럼 블록이 구분됨
```

- 블록 구조
- 일부 키워드 뒤에서는 들여쓰기로 코드 블록을 구분 (Python과 비슷)
  - Ex) where, let ... in, do, case ... of
  - C와 같이 { } 및 ;도 사용 가능하지만 권장하지 않음

Haskell의 기초(4)

- 기본 데이터 타입(Bulit-in Type)
- 정수: Int(고정 크기), Integer(크기 제한 없음)
- 실수: Float, Double
- 문자 및 문자열: Char, [Char]
  - Ex) 'a', '한'
  - Ex) "Hello"
- Bool: True, False
- 리스트: 동일한 타입의 원소들의 집합
  - Ex) [1,2,3]
- 튜플: 다른 타입의 값들의 집합
  - 단, 고정된 개수의 원소만 가지고 길이가 변하지 않음
  - Ex) (1, "hi")

```
import Data.Typeable

x :: Float
x = 3.14

y :: [Int]
y = [1,2,3]

z :: (Int, Char)
z = (1,'a')

main = do                       --- 들여쓰기에 유의
  print (typeOf 'a')            --- Char
  print (typeOf "abc")          --- [Char]
  print (typeOf True)           --- Bool
  print (typeOf x)              --- Float
  print (typeOf y)              --- [Int]
  print (typeOf z)              --- (Int,Char)
```

Haskell의 기초(5)

- 사용자 정의 데이터 타입(User-defined Type)
- data 키워드 사용
- 하나의 타입은 여러 생성자를 정의 가능
  - 각 생성자는 0개 이상의 타입 필드를 정의 가능
  - data 타입 이름 = 생성자1 타입1 타입2 ...
    | 생성자2 타입1 타입2 ...
    | 생성자3...
  - Ex) data Shape = Circle Float | Rectangle Float Float
- 타입은 레코드로 구성 가능
  - { 필드 이름1 :: 타입1, 필드 이름2 :: 타입2, ...}
  - 필드 이름으로 접근 가능
- 지원
  - 정의 뒤에 deriving (Show, Eq)을 붙이면 print 및 비교 가능

```
import Data.Typeable

data Point = Point Int Int
  deriving (Show, Eq)
data Shape = Circle Float | Rectangle Float Float
  deriving (Show, Eq)
p = Circle 3.3

data Student = Student { name :: String, age :: Int, score ::
Int }
alice = Student { name = "Alice", age = 20, score = 95 }

main = do
  print (Point 3 5)        -- Point 3 5
  print (p)                -- Circle 3.3
  print (typeOf p)         -- Shape
  print (age alice)        -- 20
  print (typeOf alice)     -- Student
```

Haskell의 기초(6)

- 리스트
- 대괄호 [ ] 안에 원소를 나열
  - 원소는 반드시 같은 타입
- 빈 리스트: []
- 관련 연산
  - ++ 연산자: 연결
  - : 연산자: 요소 추가
  - .. 연산자: 생성
  - take, drop 연산자: 슬라이싱
  - | 연산자: 내포
- 지연 평가(Lazy Evaluation)
  - 무한 리스트 생성 가능

```
[1,2] ++ [3,4]   -- [1,2,3,4]

1 : [2,3,4]   -- [1,2,3,4]
'a' : "bc"     -- "abc"

[1..5]        -- [1,2,3,4,5]
[2,4..10]   -- [2,4,6,8,10]
['a'..'e']     -- "abcde"

take 3 [1..5]    -- [1,2,3]  (앞에서 3개)
drop 3 [1..5]    -- [4,5]  (앞에서 3개 버림)

[x*2 | x <- [1..5]]                    -- [2,4,6,8,10]
[(x,y) | x <- [1,2], y <- [3,4]]    -- [(1,3),(1,4),(2,3),(2,4)]

[1..]                -- 1부터 시작하는 무한 리스트
cycle [1,2,3]     -- 무한 반복 [1,2,3,1,2,3,...]
```

Haskell의 기초(7)

- 함수의 기초
- 함수 정의
  - 함수이름 매개변수1 매개변수2 ... = 본문
  - Ex) add x y = x + y
- 함수 호출
  - 함수명과 인자를 공백으로 구분
  - Ex) add 3 5
- 함수 타입 선언
  - 함수 정의와 별도로 타입만 선언 가능
  - Ex) add1 :: Int -> Int
- 지역 변수
  - let ... in 으로 선언 가능
  - Ex) addWithLet x y = let z = x + y in z * 2
  - Ex) (x + y) * 2

```
add1 :: Int -> Int
add1 x = x + 1        -- int add1(int x) { return x+1;}

add:: Int -> Int -> Int
add x y = x + y       -- int add(int x, int y) { return x+y;}

addWithLet x y = let z = x + y in z * 2
-- int addWithLet(int x, int y) {
--       final int z = x + y;   // let z = x + y
--       return z * 2;          // in z * 2

main = do
  print (add 3 5)
  print (add1 3)
  print (addWithLet 3 2)
```

```
8
4
10
```

Haskell의 기초(8)

- 함수의 패턴 매칭
- 값의 구조에 따라 함수를 다르게 정의
  - if나 switch문 없이 간결하게 표현 가능
- 같은 함수명에 대해서는 순서가 우선순위
- 인자가 사용되지 않을 경우 _로 무시
  - 단, _는 마지막에 두는 것이 안전
  - Ex) numberWord _ = "many"가 제일 위에 있는 경우
- 값의 분해 가능
  - 리스트를 자동으로 분해
  - Ex) first (x:xs) = x -- 첫 번째 원소 x와 나머지 xs
  - 구조체로 정의된 데이터 타입도 가능
  - Ex) Student {score = s} -- Student 생성자의 score 필드

```
numberWord :: Int -> String
numberWord 0 = "zero"
numberWord 1 = "one"
numberWord 2 = "two"
numberWord _ = "many"

main = putStrLn (numberWord 2)   --- "two"가 출력
```

Ex) Python

```
def numberWord(n: int) -> str:
    if n == 0:
        return "zero"
    elif n == 1:
        return "one"
    elif n == 2:
        return "two"
    else:
        return "many"   # 나머지 경우
```

Haskell의 기초(9)

- 함수의 패턴 매칭
- 값의 구조에 따라 함수를 다르게 정의
  - if나 switch문 없이 간결하게 표현 가능
- 같은 함수명에 대해서는 순서가 우선순위
- 인자가 사용되지 않을 경우 _로 무시
  - 단, _는 마지막에 두는 것이 안전
  - Ex) numberWord _ = "many"가 제일 위에 있는 경우
- 값의 분해 가능
  - 리스트를 자동으로 분해
  - Ex) first (x:xs) = x -- 첫 번째 원소 x와 나머지 xs
  - 구조체로 정의된 데이터 타입도 가능
  - Ex) Student {score = s} -- Student 생성자의 score 필드

```
first :: [String] -> String
first (x:xs) = x
-- first (x:xs) = xs : 에러 발생! 왜?
first []   = "empty"

notfirst :: [String] -> [String]
notfirst (x:xs) = xs
notfirst []   = ["empty"]

main = do
  print (first [])                    -- "empty"
  print (first ["ab","ef"])           -- "ab"
  print (first ["ab"])                -- "ab"
  print (notfirst [])                 -- ["empty"]
  print (notfirst ["ab","ef","gh"])   -- ["ef","gh"]
  print (notfirst ["ab"])             -- []
```

Haskell의 기초(10)

- 함수의 가드(Guard)
- 조건(Bool) 에 따라 분기하는 방식
  - if나 switch문 없이 간결하게 표현 가능
  - 각 조건을 보호(가드)
- '|' 로 조건을 구분

```
absVal n
  | n >= 0    = n
  | otherwise = -n

absVal1 n = if n >= 0 then n else -n

data Student = Student { name :: String, score :: Int }
alice = Student { name = "Alice", score = 95 }

grade :: Student -> String
grade (Student { score = s })
  | s >= 90   = "A"
  | otherwise = "B"

main = do
  print (absVal (-3))            -- 3
  print (absVal1 (3))            -- 3
  print (grade alice)             -- "A"
```

실습문제(2)

- 2) 아래의 결과가 나오도록 Haskell 코드의 완성하시오. (단, 패턴 매칭과 가드를 모두 사용할 것)
- 조건
  - ticket이 "free"이면 "free ticket", "monthly"이면 "monthly ticket"을 출력
  - ticket이 "none"이면 나이에 따라 19세 미만은 1200, 65세 이상은 0, 아니면 2000을 출력

```
data Passenger = Passenger { name :: String, age :: Int,
ticket :: String }
  deriving Show

fare :: Passenger -> String

fare (Passenger { name = n, ticket = "free" }) =
  n ++ ": free ticket"
fare (Passenger { name = n, ticket = "monthly" }) =
  n ++ ": monthly ticket"
fare (Passenger { name = n, age = a })
  | a < 19    = n ++ ": fare = 1200"
  | a < 65    = n ++ ": fare = 2000"
  | otherwise = n ++ ": fare = 0"
```

```
main = do
  let p1 = Passenger { name = "Alice", age = 4, ticket = "none" }
  let p2 = Passenger { name = "Bob", age = 16, ticket = "free" }
  let p3 = Passenger { name = "Charlie", age = 30, ticket =
"monthly" }
  let p4 = Passenger { name = "Dave", age = 70, ticket = "none" }
  print (fare p1)
  print (fare p2)
  print (fare p3)
  print (fare p4)
```

```
"Alice: fare = 1200"
"Bob: free ticket"
"Charlie: monthly ticket"
"Dave: fare = 0"
```

실습문제(2)

- 2) 아래의 결과가 나오도록 Haskell 코드의 완성하시오. (단, 패턴 매칭과 가드를 모두 사용할 것)
- 조건
  - ticket이 "free"이면 "free ticket", "monthly"이면 "monthly ticket"을 출력
  - ticket이 "none"이면 나이에 따라 19세 미만은 1200, 65세 이상은 0, 아니면 2000을 출력

```
data Passenger = Passenger { name :: String, age :: Int, ticket :: String }
  deriving Show

fare :: Passenger -> String

fare (Passenger { name = n, ticket = "free" }) =   n ++ ": free ticket"
fare (Passenger { name = n, ticket = "monthly" }) =   n ++ ": monthly ticket"
fare (Passenger { name = n, age = a })
  | a < 19    = n ++ ": fare = 1200"
  | a < 65    = n ++ ": fare = 2000"
  | otherwise = n ++ ": fare = 0"
```

```
Passenger { name = "Bob", age = 16, ticket = "free" }
```

```
"Alice: fare = 1200"
"Bob: free ticket"
"Charlie: monthly ticket"
"Dave: fare = 0"
```

함수형 반복 구조

- 함수형 반복 구조
- Haskell을 비롯한 함수형 언어들은 for, while과 같은 명시적 반복문이 없음
- 기존 반복문의 단점
  - 흐름 제어와 계산 로직이 묶여 가변되는 중간 상태 추적 때문에 분석이나 디버깅이 어려움
  - Ex) i나 total 같은 매번 변화하는 변수에 따라 중간 상태가 변화
  - 상태를 공유하거나 변경하므로 병렬 처리에 취약
- 재귀 함수(Recusrive) 및 고차 함수(map, fold, filter등)로 구현

Ex) Python

```
total = 0
for i in range(5):
    if i % 2 == 0:
        continue
    total += i
    i += 1
print("Total:", total)
```

Ex) Haskell

```
sumOdd :: [Int] -> Int
sumOdd [] = 0
sumOdd (x:xs)
  | odd x     = x + sumOdd xs
  | otherwise = sumOdd xs

main = print (sumOdd [0..4])
```

재귀 함수(1)

- 재귀 함수
- 자기 자신을 호출하여 문제를 해결하는 함수
  - 종료 조건과 재귀 호출 부분으로 분리
- 종료 조건(Base Case)
  - 더 이상 재귀하지 않고 결과를 반환
- 재귀 호출(Recursive Case)
  - 문제를 더 작은 문제로 나눠 다시 호출

Ex) C

```
int sumArray(int arr[], int n) {
   int total = 0;
   for (int i = 0; i < n; i++) {
      total += arr[i];
      // total이나 i는 중간에  지속적으로 변경
   }
return total;
}
```

```
import Debug.Trace

sumList :: [Int] -> Int
sumList [] = 0                          -- 종료 조건
sumList (x:xs) = x + sumList xs   -- 재귀 호출
-- x값은 프로그래머가 재귀도중 임의로 변경 불가

sumListD :: [Int] -> Int
sumListD [] = 0
sumListD (x:xs) =
  trace ("x = " ++ show x ++ ", xs = " ++ show xs)
  x + sumListD xs

main = do
 print (sumList [1, 2, 3, 4])   -- 10
 print (sumListD [1, 2, 3, 4])   -- 10
```

```
x = 1, xs = [2,3,4]
x = 2, xs = [3,4]
x = 3, xs = [4]
x = 4, xs = []
```

재귀 함수(2)

- 재귀 함수
- 자기 자신을 호출하여 문제를 해결하는 함수
  - 종료 조건과 재귀 호출 부분으로 분리
- 종료 조건(Base Case)
  - 더 이상 재귀하지 않고 결과를 반환
- 재귀 호출(Recursive Case)
  - 문제를 더 작은 문제로 나눠 다시 호출

```
f :: Int -> Int
f 0 = 1
f n = n * f (n - 1)

main = do
  print (f 3)   -- ??
```

```
c :: [a] -> Int
c [] = 0
c (_:xs) = 1 + c xs

main = do
  print (c [10, 20, 30, 40])   -- ??
```

고차 함수(1)

- 고차 함수(Higher-Order Function)
- 함수를 인자로 받거나 반환하는 함수
  - 즉, 함수를 값처럼 다루는 함수
- 대표적인 고차 함수
  - map: 각 원소에 개별적으로 함수를 적용
  - filter: 조건을 만족하는 원소만 남김
  - foldl / foldr: 원소들의 누적 계산 (합, 곱 등)
  - compose (.): 함수들을 하나의 함수로 합성

- map
- 각 원소에 함수를 적용하여, 결과 리스트를 반환
  - map func [x, y, z] ->[func x, func y, func z]
  - map :: (a -> b) -> [a] -> [b]
  - Ex) map (*2) [1,2,3,4] -- [2,4,6,8]
  - Ex) map length ["hi","hello","a"] -- [2,5,1]

```
import Data.Char
import Data.Typeable

capitalize :: String -> String
capitalize [] = []
capitalize (x:xs) = toUpper x : map toLower xs

main = do
  print (typeOf toUpper)
  print (typeOf toLower)
  print (map (map toUpper) ["hi", "hello"])
  -- print (map toUpper ["hi", "hello"]) -- 오류?
  print ([map toUpper "hi", map toUpper "hello"])
  print (map capitalize ["hI", "ByE"])
```

```
Char -> Char
Char -> Char
["HI","HELLO"]
["HI","HELLO"]
["Hi","Bye"]
```

고차 함수(2)

- filter
- 특정 조건을 만족하는 원소만 리스트로 반환
  - filter p [x, y] = [x | p x == True] ++[y | p y == True]
  - filter :: (a -> Bool) -> [a] -> [a]
  - Ex) filter (> 3) [1,2,3,4,5] -- [4,5]
  - Ex) filter isLower "HeLLo" -- "eo"

```
longerThan3 :: String -> Bool
longerThan3 s = length s > 3

sE :: [Int] -> [Int]
sE xs = map (^2) (filter even xs)

main = do
  print (filter longerThan3 ["hi","hello","bye","world"])
  print (sE [1..10])
```

```
["hello","world"]
[4,16,36,64,100]
```

고차 함수(3)

- fold
- 원소들을 하나의 값으로 축소(Reduce)
  - fold func init [x, y] = (func (func init x) y)
  - 리스트가 아닌 값을 반환
- fold라는 함수는 없으며, foldl/foldr만 존재
  - foldl: 왼쪽 원소부터 축소
  - foldl :: (b -> a -> b) -> b -> [a] -> b
  - foldr: 오른쪽 원소부터 축소
  - foldr :: (a -> b -> b) -> b -> [a] -> b
  - Ex) foldl (*) 4 [1,2,3] -- ((4*1)*2*3)

```
import Data.Char

myMax x y
  | x >= y    = x
  | otherwise = y

add x y = x + y

countChar acc c
  | isUpper c = acc + 1
  | otherwise = acc

main = do
  print (foldl myMax 0 [1,3,11,10,4])
  print (foldl add 0 [1,3,11,10,4])
  print (foldl countChar 0 "Hello World!")
```

```
11
29
2
```

고차 함수(4)

- compose(.)
- 두 함수의 합성하여 하나의 함수로 반환
  - 반환 값이 함수!
  - 수학의 합성과 동일: $(f \circ g)(x) = f(g(x))$
  - func.g x = func (g x)
  - (.) :: (b -> c) -> (a -> b) -> a -> c
  - Ex) (length.filter even) [1..4] -- length(filter even [1..4])

```
import Data.Char

process = map toUpper . reverse . filter isLower

step1 :: String -> String
step1 = reverse . filter isLower

step2 :: String -> String
step2 = map toUpper . step1

main = do
  print (process "Hello WoRLD")
  print (step2 "Hello WoRLD")
```

```
"OOLLE"
"OOLLE"
```

람다 함수

- 람다 함수(Lambda Function)
- 이름 없는 함수
  - 다른 함수에 간단한 함수를 전달할 때 사용
  - \인자1 인자2 -> 본문
  - Ex) (\x -> x + 1) 5 -- 6
- 일회성 함수 표현에 유리
  - 매번 함수명을 작성하지 않아도 됨
  - 고차 함수와의 결합에 유리
  - Ex) map (\x -> x*2) [1,2,3]

```
import Data.Char

isAlphaOrSpace c = isAlpha c || c == ' '

wordCountCase =
   length
 . words
--  . filter isAlphaOrSpace
 . filter (\c -> isAlpha c || c == ' ')

main = do
 print (wordCountCase "Hello, world!")
```

```
2
```

고차 함수(5)

- 영단어 개수 세기 예제
- length: 리스트의 원소의 개수를 반환
  - Ex) length [1,2,3] -- 3
- words: 문장을 단어들의 리스트로 반환
  - Ex) words "Hello World" -- ["Hello","World"]
- isAlpha: 문자가 알파벳인지 판별
  - Ex) isAlpha '!' -- False

```
import Data.Char

isAlphaOrSpace c = isAlpha c || c == ' '

wordCountCase =
   length
 . words
 . filter isAlphaOrSpace

main = do
  print (wordCountCase "Hello, world!")
  print (wordCountCase "Numbers 123 and words")
  print (wordCountCase "Hi!!  Bye??  See-you")
```

```
2
3
3
```

실습문제(3)

- 3) 입력된 문장에서 가장 긴 단어와 길이를 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
  - 영단어만 입력된다고 가정 (isAlpha 필요 없음)

```
longer :: String -> String -> String
longer acc w
```

```

```

```
longestWordWithLen :: String -> (String, Int)
longestWordWithLen text =
 let lw = foldl longer "" (words text)
 in (lw, length lw)

main = do
  print (longestWordWithLen "map filter fold composition")
```

```
("composition",11)
```

실습문제(3)

- 3) 입력된 문장에서 가장 긴 단어와 길이를 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
  - 영단어만 입력된다고 가정 (isAlpha 필요 없음)

```
longer :: String -> String -> String
longer acc w
 | length w > length acc = w
 | otherwise             = acc

longestWordWithLen :: String -> (String, Int)
longestWordWithLen text =
 let lw = foldl longer "" (words text)
 in (lw, length lw)

main = do
  print (longestWordWithLen "map filter fold composition")
```

```
("composition",11)
```

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
```

```
[("apple",3),("banana",2),("orange",1)]
```

```
toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(String, Int)]
wordFrequency =
   foldl mergeWord []
 . map toPair                    -- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
 . words                         -- ["apple","banana","apple","apple","orange","banana"]

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
mergeWord ((word, c):rest) (w, n)
 | word == w  = (word, c + n) : rest
 | otherwise  = (word, c) : mergeWord rest (w, n)

toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(String, Int)]
wordFrequency =
   foldl mergeWord []
 . map toPair                    -- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
 . words                         -- ["apple","banana","apple","apple","orange","banana"]

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

```
[("apple",3),("banana",2),("orange",1)]
```

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
mergeWord ((word, c):rest) (w, n)
 | word == w  = (word, c + n) : rest
 | otherwise  = (word, c) : mergeWord rest (w, n)

toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(St
wordFrequency =
   foldl mergeWord []
 . map toPair
 . words

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

[] ("apple",1) -> [("apple",1)]

남은 리스트 : [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
-> [("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]

mergeWord [] ("apple",1) -> [("apple",1)]

-- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
-- ["apple","banana","apple","apple","orange","banana"]

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
mergeWord ((word, c):rest) (w, n)
 | word == w  = (word, c + n) : rest
 | otherwise  = (word, c) : mergeWord rest (w, n)

toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(St
wordFrequency =
   foldl mergeWord []
 . map toPair
 . words

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

[("apple",1)] ("banana",1)
word = "apple", c = 1, rest = [], w = "banana", n =1

("apple",1) : mergeWord [] ("banana",1) -> ???

[("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
 -> [("apple",1),("apple",1),("orange",1),("banana",1)]

mergeWord [("apple",1)] ("banana",1) -> ???

-- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
-- ["apple","banana","apple","apple","orange","banana"]

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
mergeWord ((word, c):rest) (w, n)
 | word == w  = (word, c + n) : rest
 | otherwise  = (word, c) : mergeWord rest (w, n)

toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(St
wordFrequency =
   foldl mergeWord []
 . map toPair
 . words
```

[] ("banana",1) -> [("banana",1)]

("apple",1) : mergeWord [] ("banana",1) ->
[("apple",1), ("banana",1)]

[("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
 -> [("apple",1),("apple",1),("orange",1),("banana",1)]

mergeWord [("apple",1)] ("banana",1) -> ???

```
 -- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
 -- ["apple","banana","apple","apple","orange","banana"]

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
mergeWord ((word, c):rest) (w, n)
 | word == w  = (word, c + n) : rest
 | otherwise  = (word, c) : mergeWord rest (w, n)

toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(St
wordFrequency =
   foldl mergeWord []
 . map toPair
 . words

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

```
[("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
 -> [("apple",1),("apple",1),("orange",1),("banana",1)]

mergeWord [("apple",1)] ("banana",1) -> [("apple",1), ("banana",1)]
-- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
-- ["apple","banana","apple","apple","orange","banana"]
```

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
mergeWord ((word, c):rest) (w, n)
 | word == w  = (word, c + n) : rest
 | otherwise  = (word, c) : mergeWord rest (w, n)

toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(St
wordFrequency =
   foldl mergeWord []
 . map toPair
 . words

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

[("apple",1), ("banana",1)] ("apple",1)
word = "apple", c = 1, rest = [("banana",1)], w ="apple", n =1

[("apple",1),("apple",1),("orange",1),("banana",1)] ->
[("apple",1),("orange",1),("banana",1)]

mergeWord [("apple",1), ("banana",1)] ("apple",1) -> ???

-- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
-- ["apple","banana","apple","apple","orange","banana"]

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
mergeWord ((word, c):rest) (w, n)
 | word == w  = (word, c + n) : rest
 | otherwise  = (word, c) : mergeWord rest (w, n)

toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(St
wordFrequency =
   foldl mergeWord []
 . map toPair
 . words
```

word = "apple", c = 1, rest = [("banana",1)], w ="apple", n =1
("apple",2) : [("banana",1)] -> [("apple",2) : ("banana",1)]

[("apple",1),("apple",1),("orange",1),("banana",1)] ->
[("apple",1),("orange",1),("banana",1)]

mergeWord [("apple",1), ("banana",1)] ("apple",1) -> ???

```
                          -- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
                          -- ["apple","banana","apple","apple","orange","banana"]

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

실습문제(4)

- 4) 단어와 단어의 개수를 투플로 만들어 출력하는 코드이다. 다음 Haskell 코드를 완성하시오.
- 영단어만 입력된다고 가정하고, 단어의 길이는 무관(isAlpha나 length 필요 없음)

```
mergeWord :: [(String, Int)] -> (String, Int) -> [(String, Int)]
mergeWord [] (w, n) = [(w, n)]
mergeWord ((word, c):rest) (w, n)
 | word == w  = (word, c + n) : rest
 | otherwise  = (word, c) : mergeWord rest (w, n)

toPair :: String -> (String, Int)
toPair w = (w, 1)

wordFrequency :: String -> [(St
wordFrequency =
   foldl mergeWord []
 . map toPair
 . words
```

```
[("apple",1),("apple",1),("orange",1),("banana",1)] ->
[ ("apple",1),("orange",1),("banana",1)]

mergeWord [("apple",1), ("banana",1)] ("apple",1) -> [("apple",2) : ("banana",1)]
-- [("apple",1),("banana",1),("apple",1),("apple",1),("orange",1),("banana",1)]
-- ["apple","banana","apple","apple","orange","banana"]

main = do
 print (wordFrequency "apple banana apple apple orange banana")
```

Python과 함수형 프로그래밍

- Python 2.x 이후
- map, filter, reduce, lambda 같은 함수형 요소 도입
  - reduce는 foldl의 역할

```
Ex) lambda
f = lambda x: x + 1
print(f(5))  # 6

Ex) map, filter, reduce
nums = [1, 2, 3, 4]
list(map(lambda x: x*x, nums))              # [1,4,9,16]
list(filter(lambda x: x % 2 == 0, nums))    # [2,4,6]
reduce(lambda acc, x: acc + x, nums, 0)     # 10
```

```
words = ["hi", "hello", "bye", "world"]
result = []
for w in words:
   if len(w) >= 3:
      result.append(w.upper())
print(result)   # ['HELLO', 'BYE', 'WORLD']

# 함수형으로 수정
words = ["hi", "hello", "bye", "world"]
result = list(map(str.upper, filter(lambda w: len(w) >= 3, words)))
print(result)   # ['HELLO', 'BYE', 'WORLD']
```

Node.js(1) - Race condition Example

```
const processWithForAwait = async (transfers) => {
  let totalBalance = 1000000;
  console.time("For-Await-Time");

  for await (const amount of transfers) {
    const isValid = await verifyTransaction(amount);
   // 0.5초 대기
   if (isValid) {
     totalBalance += amount;
   }
  }

  console.timeEnd("For-Await-Time");
  // 결과: 약 500초
  return totalBalance;
};
```

```
const processWithPromiseAll = async (transfers) => {
  let totalBalance = 1000000;
  console.time("Promise-All-Time");

  await Promise.all(transfers.map(async (amount) => {
    const isValid = await verifyTransaction(amount);
    // 1,000개 동시 실행 (0.5초)
    if (isValid) {
      // 여러 함수가 동시에 totalBalance를 읽고 덮어씀
      const current = totalBalance;
      totalBalance = current + amount;
    }
  }));

  console.timeEnd("Promise-All-Time"); // 결과: 약 0.5초
  return totalBalance; // 결과값 오류 (일부 입금액 증발)
};
```

Node.js(2) - Race condition Example

```
const { Mutex } = require('async-mutex');
const mutex = new Mutex();

const processWithMutex = async (transfers) => {
 let totalBalance = 1000000;
 console.time("Mutex-Time");

 await Promise.all(transfers.map(async (amount) => {
  // [보안 검증]은 락 밖에서 1,000개가 동시에 진행 (Parallel)
  const isValid = await verifyTransaction(amount);

  if (isValid) {
   // [잔액 업데이트]만 뮤텍스로 보호 (Critical Section)
   await mutex.runExclusive(async () => {
    totalBalance += amount;
   // 아주 짧은 순간만 줄을 세움
   });
  }
 }));
 console.timeEnd("Mutex-Time");
  // 결과: 약 0.5~0.6초
 return totalBalance; //  결과값 정확
};
```

Node.js(3) - Race condition Example

```
const processFunctionally = async (transfers) => {
  const initialBalance = 1000000;
  console.time("Functional-Time");

  // 1. [Map] 모든 송금 요청을 '검증된 금액'으로 변환하는 비동기 작업들 생성
  // 이 단계에서는 totalBalance를 건드리지 않으므로 Race Condition이 절대 발생 안 함
  const analysisPromises = transfers.map(async (amount) => {
    const isValid = await verifyTransaction(amount); // 병렬 실행 (0.5초)
    return isValid ? amount : 0; // 유효하면 금액, 아니면 0 반환
  });

  // 2. [Parallel Execution] 모든 비동기 작업을 동시에 실행하고 결과를 배열로 받음
  const results = await Promise.all(analysisPromises);

  // 3. [Reduce] 결과 배열을 하나로 합산 (순수 함수)
  const totalAdded = results.reduce((acc, curr) => acc + curr, 0);

  console.timeEnd("Functional-Time"); // 결과: 약 0.5초
  return initialBalance + totalAdded; // 최종 잔액 반환
};
```

Node.js(4) - Abstraction Level

```
async function performDailySettlement(transactions) {
   const validTransactions = await filterInvalidTransactions(transactions);
   const settlementSummary = summarizeFinancialData(validTransactions);
   await updateMasterLedger(settlementSummary);
   console.log("일일 정산이 완료되었습니다.");
}
```

```
async function performDailySettlement(transactions) {
   const validTransactions = await filterInvalidTransactions(transactions);
   const settlementSummary = summarizeFinancialData(validTransactions);
   const release = await mutex.acquire();
   await db.query(`UPDATE ledger SET bal = bal + ${settlementSummary.amount}`);
   await fs.promises.appendFile('log.txt', `Added: ${settlementSummary.amount}`);
   release();
   console.log("일일 정산이 완료되었습니다.");
}
```

프로그래밍 언어 패러다임(2)

논리 프로그래밍(1)

- 논리 프로그래밍
- 프로그램을 논리식으로 구성하고, 계산을 논리적 추론 과정으로 수행하는 패러다임
- 1차 술어 논리(First-Order Predicate Logic)
  - 프로그램을 사실(Facts), 규칙(Rules), 질의(Queries)로 구성
  - 주어진 프로그램 P와 질의 Q에 대해, 해답이 논리적으로 만족하는지 여부를 확인하는 과정으로 도출
  - 실제 구현에서는 통일(Unification)과 역추적(Backtracking)을 사용하여 답을 구함
- 특징
  - 선언적(Declarative): 알고리즘 절차를 명시하지 않고, 문제의 성질을 서술
  - 추론 기반: 실행은 논리 추론 과정 자체
  - 비결정성: 여러 답을 찾을 수 있으며, 시스템이 탐색
- 대표 언어
  - Prolog: 가장 널리 쓰이는 논리 프로그래밍 언어
  - Datalog: 데이터베이스 질의용

논리 프로그래밍(2)

- 선언적(Declarative) 프로그래밍 패러다임
  - 무엇(What)을 기술하지 어떻게(How)를 기술하지 않음
    - 논리 프로그래밍, 함수형 프로그래밍
    - 부작용(Side Effect) 최소화 지향
  - 상태 변화나 제어 흐름을 직접 명시하는 명령형(Imperative) 프로그래밍 패러다임과 구분

- 함수형 프로그래밍과 논리 프로그래밍

구분	함수형	논리
기반	람다 계산(λ-calculus)	술어 논리(Predicate logic)
계산	함수 호출과 값 평가	논리 추론
구성 단위	함수(Function)	사실(Fact) + 규칙(Rule)
흐름 제어	재귀/고차함수로 제어	질의(Query)와 추론 엔진으로 제어
결과	함수가 입력을 값으로 변환	질의가 참/거짓, 해답(substitutions)으로 변환

Prolog(1)

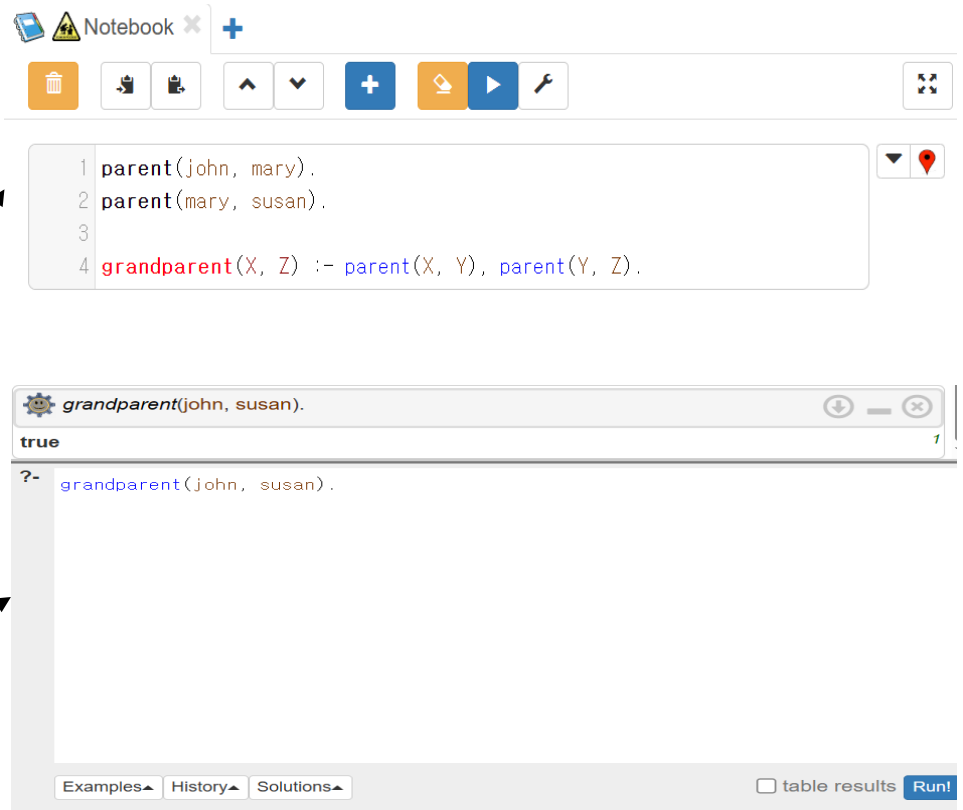
- Prolog(PROgramming in LOGic)
- 1972년 프랑스 Marseille 대학의 Alain Colmerauer와 Philippe Roussel이 개발

- SWI-Prolog
- Prolog 언어의 가장 널리 쓰이는 오픈소스 구현체
  - 1987년에 Jan Wielemaker가 개발
  - 교육, 인공지능, 지식 표현(KR), 추론 시스템 등에 많이 사용
- 실습
  - https://swish.swi-prolog.org/
  - 사실과 규칙을 입력하는 부분과 질의 부분이 분리됨

```
parent(john, mary).
parent(mary, susan).

grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

```
?- grandparent(john, susan).
true.
```



Prolog(2)

- 연산자 및 예약어

연산자	설명	예
:-	규칙 정의	Head :- Body
,	논리곱(AND)	true, true % true
;	논리합(OR)	false; true % true
\+	논리부정(NOT)	\+ true % false
-> ;	if- then-else	A->B ; C
!	백트래킹 중지	
is	계산후 대입	X is 2+3. % X는 5로 바인딩
=:=, =\=	산술 동등 비교	5 =:= 2+3. % true
<,>,>=,=<	산술 대소 비교	3 >= 1+1. % true
=	통일(Unficiation)	point(X, 2) = point(1, Y). % X=1, Y=2
==, \==	객체 동등 비교	X = 5, X == 5. % true

예약어	설명	예
true/fail	항상 참/거짓	?- true. % true
repeat	무한 반복	?- repeat, write('hello') %hello 무한 반복 출력
halt	프로그램 종료	?- halt. % 종료
listing	술어 출력	?- listing(parent). % parent(john, mary).
trace	디버깅 추적	?- trace, grandparent(john, susan) % Call: ...
dynamic	동적 변경 가능	:- dynamic fact/1. fact(a).
assert/1	사실 추가	?- assert(fact(b)).
retract/1	사실 제거	?- retract(fact(b)).

Prolog(3)

- Prolog의 기초
- 사실(Fact)
  - 어떤 것이 참임을 기술하는 문장
  - 형식: predicate(argument1, argument2, ...)
- 규칙(Rule)
  - 조건이 참일 때 결론이 성립한다는 의미
  - 형식: Head :- Body
  - 의미: Body가 true이면 Head도 true이다.
- 질의(Query)
  - 사실과 규칙에 대해 질문을 던짐
  - 형식: ?- Goal

```
parent(john, mary).
parent(mary, susan).

grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

```
?- grandparent(john, susan).
true.

?- true , false.
false.

?- true ; false.
true.

?- \+ true.
false.
```

Prolog(4)

- 데이터 타입

데이터 타입	설명	예
atom	최소 단위 (이름, 문자열, 기호)	apple, 'Hello', :=
integer	정수	25
float	실수	3.14
number	정수와 실수	120
list	리스트 [항목1, 항목2, ...]	[1,2,3]
string	문자열	'Hello'
var	바인딩 되지 않은 변수	?- var(X). % true
nonvar	바인딩 된 변수(값이 존재)	?- X=5, nonvar(X). % true
compound	복합구조(사용자 정의 구조) functor(Arg1, Arg2, ..., ArgN)	?- compound(point(3,4)). % true ?- point(3, 4), point(X, 4). % X=3

Prolog(5)

- 명명 규칙
- 규칙 이름(Rule)
  - 소문자로 시작
  - Ex) parent(a, c)
- 상수
  - 소문자로 시작 또는 ' '으로 묶음
  - Ex) john, 'HELLO'
- 변수
  - 대문자 또는 _로 시작
  - Ex) X, Person, _temp
- 익명 변수
  - _ (underscore)
  - 값이 무엇이든 상관없을 때 사용

```
parent(john, mary).
parent(mary, susan).

grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

```
?- parent(john, X).
mary.

?- parent(john, X), parent(X, susan).
X = mary.

?- parent(john, X); parent(mary, X).
X = mary
X = susan.
```

Prolog(6)

- 리스트
- 기본 정의
  - []와 ,로 표현
  - Ex) List = [1,2,3,4]
- | 연산자
  - 리스트의 첫 번째 원소나 나머지 원소를 구분
  - Ex) [H|T] = [1,2,3,4] % H = [1], T =[2,3,4]
  - Ex) X, Y, Person
- 내장 함수
  - member(X, List): X가 List의 원소인지 검사
  - length(List, N): 리스트 길이 구하기
  - append(L1, L2, L3): 리스트 연결
  - select(E,L,R): L에서 원소 E 제거 → R

```
sum_list([], 0).
sum_list([H|T], Sum) :-
   sum_list(T, Rest), Sum is H + Rest.
```

```
?- [A,B|Rest] = [10,20,30,40].
A = 10,
B = 20,
Rest = [30, 40].

?- length([a,b,c], N).
N=3

?- append([1,2],[3,4],R).
R = [1, 2, 3, 4]

?- select(2,[1,2,3],R).
R = [1, 3]

?- sum_list([1,2,3,4], S).
S = 10
```

Prolog(7)

- 통일(Unification) 연산(=)
- 비교도 아니고, 할당도 아님
- 두 항을 같아지도록 만드는 논리적 연산
  - 가능하면 변수에 바인딩이 발생
  - 불가능하면 실패
  - 목표에 해당하며 사실이나 규칙에서 사용 불가
- 방향성 없음
  - Ex) X = 3과 3 = X는 동일
- 재할당 불가
  - Prolog 변수는 불변

```
?- 5 = X.          % 순서 무관
X = 5.

?- X = 3, X = 4.  % X는 3에 할당되어서 4로 재할당 불가
false.
```

```
?- X = Y, X = 3.
X = Y, Y = 3.   % 둘 다 3으로 확정

?- point(3,4) = point(3,4).
true.
?- point(X, Y) = point(3, 4).
X = 3,
Y = 4.

?- X = 2+3.                % 식 자체가 들어감
X = 2+3.
?- X = 2+3, X = 5.         % 5로 재할당 불가
false
?- X is 2+3, X =:=5        % 계산: X = 5
X = 5.
?- X = 2+3, X = Y, Y=3+2.   % Y는 3+2로 재할당 불가
false
```

Prolog(8)

- Ex) Prolog 예제 1 - 과일 분류

```
% Facts
color(apple, red).
color(banana, yellow).
color(grape, purple).
color(kiwi, green).

taste(apple, sweet).
taste(banana, sweet).
taste(grape, sour).
taste(kiwi, sour).

% Rules
tasty(Fruit) :- color(Fruit, red).
tasty(Fruit) :- color(Fruit, yellow), taste(Fruit, sweet).
tasty(Fruit) :- color(Fruit, purple), taste(Fruit, sour).
```

```
?- tasty(apple).
true.

?- tasty(kiwi).
false.

?- taste(X, sweet).
X = apple ;
X = banana.

?- tasty(X).
X = apple ;
X = banana ;
X = grape.

?- color(Y,_), \+ tasty(Y).
Y = kiwi.
```

Prolog(9)

- Ex) Prolog 예제 2 - 방향성 있는 그래프 표현

```
% Facts
edge(a, b).
edge(b, c).
edge(c, d).
edge(a, e).
edge(e, d).

% Rules
path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
```

```
?- path(a,d).
true.
?- path(d,a).
false.

?- edge(a, X).
X = b ;
X = e.

?- edge(b, X).
X = c.

?- path(a, X).
X = b ;
X = e ;
X = c ;
X = d ;
X = d.
```

Prolog(10)

- Ex) Prolog 예제 3 - 방향성 없는 그래프 표현

```
% Facts
edge(a, b).
edge(b, c).
edge(c, d).
edge(a, e).
edge(e, d).

% Rules
connected(X,Y) :- edge(X,Y).
connected(X,Y) :- edge(Y,X).

path(X, Y) :- connected(X, Y).
path(X, Y) :- connected(X, Z), path(Z, Y).
```

```
?- path(a,d).          % 무한 루프에 주의
true ;
true ;
...

?- connected(b, X).
X = c ;
X = a.

?- path(a, X).         % 무한 루프에 주의
X = b ;
X = e ;
X = c ;
X = a ;
X = d
...
```

Prolog(11)

- Ex) Prolog 예제 4 - 무한루프 없는 방향성 없는 그래프 표현

```
% Facts
edge(a, b).
edge(b, c).
edge(c, d).
edge(a, e).
edge(e, d).

% Rules
connected(X,Y) :- edge(X,Y).
connected(X,Y) :- edge(Y,X).

path(X,Y) :- path(X,Y,[X]).
path(X,Y,_) :- connected(X,Y).
path(X,Y,Visited) :-
   connected(X,Z),
   \+ member(Z,Visited),
   path(Z,Y,[Z|Visited]).
```

```
?- path(a,d).        % 여러번 나오지만 무한 루프는 아님
true ;
true ;
...
true.

?- path(a, X).       % 여러번 나오지만 무한 루프는 아님
X = b ;
X = e ;
X = c ;
X = a ;
X = d
...
X = a.
```

실습문제(5)

- 5) 그래프에서 출발과 도착 지점 사이의 경로가 나오도록 다음 코드를 완성하시오.
- 방향성은 있는 그래프이고,무한루프는 없다고 가정

```
% Facts
edge(a, b).
edge(b, c).
edge(c, d).
edge(a, e).
edge(e, d).

% Rules
path(X, Y, [X,Y]) :-
   edge(X,Y).
```

```
?- path(a,d,P).
P = [a, b, c, d] ;
P = [a, e, d].

?- path(d,a,P).   % 경로 없음
false
```

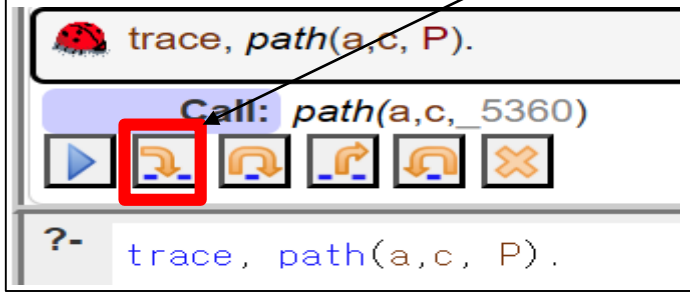
실습문제(5)

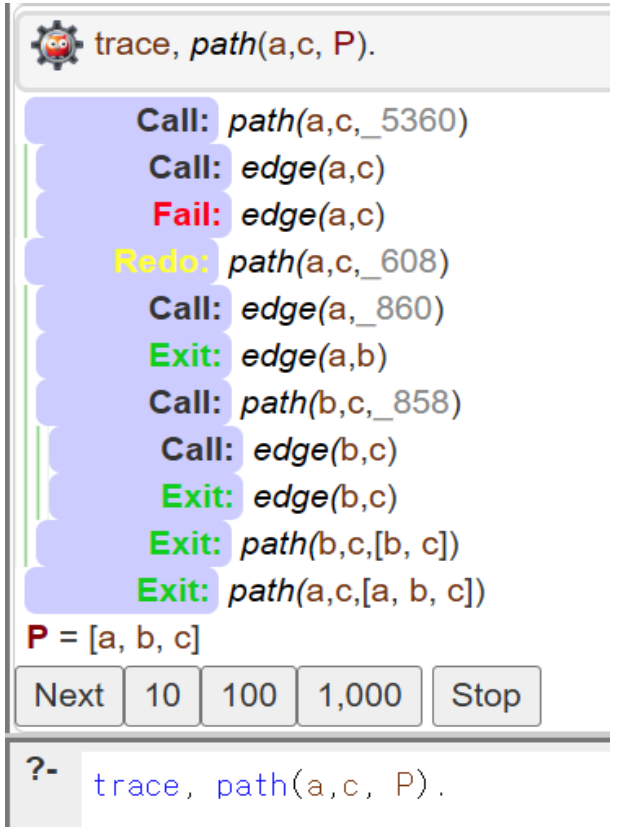
- 5) 그래프에서 출발과 도착 지점 사이의 경로가 나오도록 다음 코드를 완성하시오.
- trace를 사용하면 추적이 가능

```
% Facts
edge(a, b).
edge(b, c).
edge(c, d).
edge(a, e).
edge(e, d).

% Rules
path(X, Y, [X,Y]) :-
   edge(X,Y).
```

?- trace, path(a,c,P). % Step into 버튼 사용





실습문제(5)

- 5) 그래프에서 출발과 도착 지점 사이의 경로가 나오도록 다음 코드를 완성하시오.
- 방향성은 있는 그래프이고,무한루프는 없다고 가정

```
% Facts
edge(a, b).
edge(b, c).
edge(c, d).
edge(a, e).
edge(e, d).

% Rules
path(X, Y, [X,Y]) :-
   edge(X,Y).

path(X, Y, [X|Rest]) :-
   edge(X,Z),
   path(Z, Y, Rest).
```

```
?- path(a,d,P).
P = [a, b, c, d] ;
P = [a, e, d].

?- path(d,a,P).   % 경로 없음
false
```

Prolog(12)

- 강 건너기 문제(River Crossing Problem)
- 조건
  - 하나의 강이 있고, 강을 건널 수 있는 배가 있다.
  - 배는 농부(Farmer)가 조종해야 하므로, 농부가 항상 배에 타야 한다.
  - 배에는 농부 외에 한 번에 최대 하나의 동물만 추가로 실을 수 있다.
  - 동물은 사자(Lion), 늑대(Wolf), 염소(Goat)이 있다.
- 제약 조건
  - 사자와 늑대는 단둘이 있으면 사자가 늑대를 잡아먹는다.
  - 늑대와 염소는 단둘이 있으면 늑대가 염소를 잡아먹는다.
- 상태
  - 초기 상태는 농부를 비롯한 모든 동물은 강의 왼쪽(Left)에 있다.
  - 목표 상태는 농부를 비롯한 모든 동물은 강의 오른쪽(Right)에 있는 것이다.
  - 모든 동물이 안전하게 이동해야 하므로, 중간에 동물이 잡아먹히는 상태를 만들면 안된다.

Prolog(13)

- 강 건너기 문제 - Step 1
- 조건
  - 동물은 염소(Goat)만 있다.

```
% Facts
% state(farmer, goat) - (left/right)
start(state(left, left)).
goal(state(right, right)).

% Rules
% move farmer and goat
move(state(F,G), state(F1,G1)) :-
   opposite(F,F1), G1 = F1, G = F.

% move farmer only
move(state(F,G), state(F1,G)) :-
   opposite(F,F1).

opposite(left,right).
opposite(right,left).
```

```
path(State,State,_,[]).
path(State,Goal,Visited,[Next|Rest]) :-
   move(State,Next),
   \+ member(Next,Visited),
   path(Next,Goal,[Next|Visited],Rest).
```

```
?- start(S), goal(G), path(S,G,[S],Steps).
G = state(right,right),
S = state(left,left),
Steps = [state(right,right)]
```

Prolog(14)

- 강 건너기 문제 - Step 2
- 조건
  - 동물은 늑대(Wolf), 염소(Goat)가 있다.

```
% Facts
% state(farmer, wolf, goat) - (left/right)
start(state(left, left, left)).
goal(state(right, right, right)).

% move farmer and wolf
move(state(F,W,G), state(F1,W1,G)) :-
   opposite(F,F1), W1 = F1, W = F.

% move farmer and goat
move(state(F,W,G), state(F1,W,G1)) :-
   opposite(F,F1), G1 = F1, G = F.

% move farmer only
move(state(F,W,G), state(F1,W,G)) :-
   opposite(F,F1).
```

```
opposite(left,right).
opposite(right,left).

safe(state(F,W,G)) :-
   (W = G -> F = G ; true).

path(State,State,_,[]).
path(State,Goal,Visited,[Next|Rest]) :-
   move(State,Next),
  safe(Next),
   \+ member(Next,Visited),
   path(Next,Goal,[Next|Visited],Rest).
```

```
?- start(S), goal(G), path(S,G,[S],Steps).
G = state(right,right,right), S = state(left,left,left),
Steps = [state(right,right,left), state(left,right,left),
state(right,right,right)]
G = state(right,right,right), S = state(left,left,left),
Steps = [state(right,left,right), state(left,left,right),
state(right,right,right)]
```

Prolog(15)

- 강 건너기 문제 - Step 3
- 조건
  - 동물은 사자(Lion), 늑대(Wolf), 염소(Goat)이 있다.

```
% Facts
% state(farmer, lion, wolf, goat) - (left/right)
start(state(left,left,left,left)).
goal(state(right,right,right,right)).

% Rules
move(state(F,L,W,G), state(F1,L,W,G)) :- opposite(F,F1).
move(state(F,F,W,G), state(F1,F1,W,G)) :- opposite(F,F1).
move(state(F,L,F,G), state(F1,L,F1,G)) :- opposite(F,F1).
move(state(F,L,W,F), state(F1,L,W,F1)) :- opposite(F,F1).

safe(state(F,L,W,G)) :-
   (W = G -> F = G ; true),
   (L = W -> F = W ; true).
```

```
opposite(left,right).
opposite(right,left).

path(State,State,_,[]).
path(State,Goal,Visited,[Next|Rest]) :-
   move(State,Next),
  safe(Next),
   \+ member(Next,Visited),
   path(Next,Goal,[Next|Visited],Rest).
```

```
?- start(S), goal(G), path(S,G,[S],Steps).
G = state(right,right,right,right), S = state(left,left,left,left),
Steps = [state(right,left,right,left), state(left,left,right,left),
state(right,right,right,left), state(left,right,left,left),
state(right,right,left,right), state(left,right,left,right),
state(right,right,right,right)]).
Steps = [state(right,left,right,left), state(left,left,right,left),
state(right,left,right,right), state(left,left,left,right),
state(right,right,left,right), state(left,right,left,right),
state(right,right,right,right)]
```

실습문제(6)

- 6) 다음은 강 건너기 문제의 해답을 가독성 있도록 출력하는 코드이다. 이 코드를 완성하시오.

```
opposite(left,right).
opposite(right,left).

describe_move(state(F1,L1,W1,G1), state(F2,L2,W2,G2)) :-
    [                                                    ]
    format("Farmer takes ~w to ~w~n", [Animal, F2]).

print_path([_]).
print_path([S1,S2|Rest]) :-
    describe_move(S1,S2),
    print_path([S2|Rest]).
```

```
?- print_path([state(right,left,right,left),
state(left,left,right,left)].
Farmer takes none to left

% state(farmer, lion, wolf, goat) - (left/right)
?- print_path([state(left,left,left,left),
state(right,left,right,left), state(left,left,right,left),
state(right,right,right,left), state(left,right,left,left),
state(right,right,left,right), state(left,right,left,right),
state(right,right,right,right)]).

Farmer takes wolf to right
Farmer takes none to left
Farmer takes lion to right
Farmer takes wolf to left
Farmer takes goat to right
Farmer takes none to left
Farmer takes wolf to right
```

실습문제(6)

- 6) 다음은 강 건너기 문제의 해답을 가독성 있도록 출력하는 코드이다. 이 코드를 완성하시오.

```
opposite(left,right).
opposite(right,left).

describe_move(state(F1,L1,W1,G1), state(F2,L2,W2,G2)) :-
   F1 \= F2,
   (
      L1 \= L2 -> Animal = lion ;
      W1 \= W2 -> Animal = wolf ;
      G1 \= G2 -> Animal = goat ;
      Animal = none
   ),
   format("Farmer takes ~w to ~w~n", [Animal, F2]).

print_path([_]).
print_path([S1,S2|Rest]) :-
   describe_move(S1,S2),
   print_path([S2|Rest]).
```

```
?- print_path([state(right,left,right,left),
state(left,left,right,left)].
Farmer takes none to left

% state(farmer, lion, wolf, goat) - (left/right)
?- print_path([state(left,left,left,left),
state(right,left,right,left), state(left,left,right,left),
state(right,right,right,left), state(left,right,left,left),
state(right,right,left,right), state(left,right,left,right),
state(right,right,right,right)]).

Farmer takes wolf to right
Farmer takes none to left
Farmer takes lion to right
Farmer takes wolf to left
Farmer takes goat to right
Farmer takes none to left
Farmer takes wolf to right
```