하스켈 프로그래밍: 논리와 수학과 프로그래밍의 연관성

숭실대학교 컴퓨터학부 세미나
2020년 11월 6일

한경대학교 컴퓨터응용수학부
이계식

함수

- 인자 여러 개를 받아 하나의 값을 생성하는 일종의 계산과정

함수 정의

아래 모양을 따름.

```
함수이름 매개변수1 매개변수2 ... 매개변수n = 함수본체
```

예제: 두 배 함수

```
double x = x + x
```

사용법은 다음과 같다.

```
함수이름 인자1 인자2 ... 인자n
```

함수 호출

계산 과정

함수가 인자와 함께 호출되면 아래 계산 과정을 통해 최종 결과를 생성한다.

1. 함수 본체에서 매개변수`k`가 인자`k`에 의해 대체된다.
2. 대체 결과가 숫자이면 계산을 멈춘다.
3. 대체 결과 또다른 함수가 호출되면 위 과정을 반복한다.

예를 들어, `double 5` 의 계산과정은 다음과 같다.

```
double 5 = 5 + 5
         = 10
```

계산 순서

`quadruple 4` 은 두 가지 방식으로 계산이 가능하다.

- 첫째: 안쪽에 있는 `double` 먼저 적용

```
quadruple 4 = double (double 4)
            = double (4 + 4)
            = double 8
            = 8 + 8
            = 16
```

- 둘째: 바깥쪽에 있는 `double` 먼저 적용

```
quadruple 4 = double (double 4)
            = double 4 + double 4
            = (4 + 4) + double 4
            = 8 + double 4
            = 8 + (4 + 4)
            = 8 + 8
            = 16
```

주의:

- 첫째 방식은 적극적 계산법, 둘째 방식은 소극적 계산법을 보여줌.
- 하스켈은 둘째 방식 사용.
- 더 오래 걸리는 방식으로 보임.
- 하지만 안 그런 경우도 있음. 이후에 예를 들어 설명함.

하스켈 함수의 계산결과는 과정이 다르더라도 도출된 결과는 항상 동일하다. 반면에 명령형 언어에서는 계산 순서에 따라 값이 달라질 수 있다.

C 언어에서의 계산 순서 예제

실제 C 언어에서 (+1) 연산을 언제 하느냐에 따라 다른 결과를 보여주는 두 개의 기호를 사용한다.

- 첫째: (+1) 먼저 하기

```c
#include <stdio.h>

int main(void) {
  int n = 1;
  int m;
  m = n + (++n);
  printf("m = %d, n = %d\n", m, n);
  return 0;
}
```

결과: `m = 3, n = 2`

- 둘째: +1 나중에 하기

```c
#include <stdio.h>

int main(void) {
  int n = 1;
  int m;
  m = n + (n++);
  printf("m = %d, n = %d\n", m, n);
  return 0;
}
```

결과: `m = 2, n = 2`

주의: 실제로 위 코드를 돌리면 경고가 발생한다. ++ 기호를 사용하면 계산과정이 매우 혼란스러워지기 때문에 사용 자제를 권하는 것으로 보인다.

소극적(lazy) 계산법 vs. 적극적(eager) 계산법

프로그래밍 언어마다 계산 방식이 다르다. 동일한 함수가 언어에 따라 계산이 멈추기도 하고 그렇지 않기도 하다.

하스켈의 소극적 계산 예제

```
nonStoppingFtn x = nonStoppingFtn (x + 1)
```

주의:

- 계산법에 상관 없이 `nonStoppingFtn` 함수를 호출하면 절대로 정지하지 않는다. 즉, 어떤 프로그래밍 언어로 `nonStoppingFtn`을 구현 하더라도 실행하면 절대 멈추지 않는다.
- 하지만 다음 `stoppingFtn`과 함께 조합되어 사용된 `sometimesStoppingFtn`은 호출 될 경우 사용되는 언어의 컴파일러/인터프리터에 따라 정지여부가 달라진다. 하스켈의 경우는 특정 값을 생성하고 정지하지만, 파이썬의 경우는 그렇지 않음을 아래에서 보여준다.

```
stoppingFtn x y = x
sometimesStoppingFtn x = stoppingFtn x (nonStoppingFtn x)
sometimesStoppingFtn 2
```

`sometimesStoppingFtn 2` 의 계산과정은 다음과 같다.

```
sometimesStoppingFtn 2 = stoppingFtn 2 nonStoppingFtn(2)
                       = 2
```

파이썬의 적극적 계산 예제

파이썬의 경우 `sometimesStopping` 함수는 실행을 멈추지 않는다. 실제로 아래와 같이 정의하고 실행해 보면 바로 확인할 수 있다.

```
def nonStoppingFtn(x):
    return nonStoppingFtn(x + 1)

def stoppingFtn(x, y):
    return x

def sometimesStoppingFtn(x):
    return stoppingFtn(x, nonStoppingFtn(x))

sometimesStoppingFtn(2)
```

파이썬에서 `sometimesStoppingFtn(2)` 의 계산과정은 다음과 같다.

```
sometimesStoppingFtn(2) = stoppingFtn(2, nonStoppingFtn(2))
                        = stoppingFtn(2, nonStoppingFtn(3))
                        = stoppingFtn(2, nonStoppingFtn(4))
                        = stoppingFtn(2, nonStoppingFtn(5))
                        = ...
```

함수 호출과 인자 개수

함수를 적용할 때 인자가 모자라더라도 반드시 오류가 발생하는 것은 아니다.

예제: 덧셈 함수

인자를 두 개 모두 받으면 계산을 실행한다.

```
addition x y = x + y
addition 3 5
```

반면에 인자를 하나만 받으면 그 결과는 인자를 하나 받는 함수가 된다.

```
add3 = addition 3
add3 5
```

함수형 언어란?

- 함수형 프로그래밍: 함수에 인자를 적용하여 계산을 실행하는 프로그래밍 기법
- 함수형 언어: 함수형 프로그래밍을 지원하는 프로그래밍 언어

예제: 1에서 10까지 더하기

- 파이썬: 변수 할당 활용, 즉, `total`가 `i`에 **저장된 값**이 반복적으로 변함.

```
total = 0;
for i in range(1,11):
    total = total + i
```

- 하스켈: 연속된 함수 호출

```
sum [1..10] = sum [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
            = 1 + sum [2, 3, 4, 5, 6, 7, 8, 9, 10]
            = 1 + 2 + sum [3, 4, 5, 6, 7, 8, 9, 10]
            = 1 + 2 + 3 + sum [4, 5, 6, 7, 8, 9, 10]
            = ...
            = 1 + 2 + 3 + ... + 10
            = 55
```

하스켈 언어의 특징

타 언어와 비교했을 때 하스켈 언어의 특징은 다음과 같다.

- 간결성
  - 간결한 문법
  - 하스켈 프로그램이 타 언어 프로그램보다 2~10배 정도 간결하게 구현 가능하다.

- 강력한 유형 체계(type system)
    - 사용되는 모든 대상의 유형(types)을 미리 확인하여 프로그램의 오류를 예방한다.
    - 유형 확인은 유형 추론(type inference) 기술로 가능.
    - 다형성(polymorphism) 및 함수 중복정의(overloading) 지원

- 리스트 조건제시법(list comprehension)
    - 기존의 리스트를 이용하여 새로운 리스트 생성
    - 보다 간결한 코드 생성 가능

- 재귀 함수(recursive functions)
    - 재귀를 이용하여 반복문(loop) 처리
        - `while`, `for` 반복문 없음
    - 인자에 대한 패턴매칭(pattern matching)과 감시자(guard, 가드) 활용 지원

- 고계 함수(higer-order functions)
    - 함수가 타 함수의 인자 또는 결과물로 사용될 수 있음
    - 패턴 매칭과 함께 매우 유용한 기능 제공
    - 도메인 특화 프로그램 작성에 유용

- 부작용(side-effects) 함수
    - 하스켈은 순수한 함수형 프로그래밍 언어임. 즉, 모든 함수는 인자를 받아 결과를 생성하는 일 이외에는 아무 것도 하지 않음. 즉, 동일한 인자에 대해 항상 동일한 값을 생성.
    - 모나드(monads), 적용자(applicatives)를 활용하여 키보드 입력, 스크린 출력 등을 다룰 수 있음.

- 제네릭 함수
    - 제네릭 프로그래밍: 특정 유형이 아닌 여러 유형에 대해 공통적으로 적용될 수 있는 프로그램 작성 기법
    - 제네릭 함수: 특정 클래스에 속한 유형의 값들에 대해 일괄적으로 적용 가능한 함수
    - 하스켈에서 타 언어에 비해 보다 일반적이며 강력한 제네릭 함수 정의 가능

- 소극적 계산(lazy evaluation)
    - 결과가 요구될 때만 계산 실행
    - 특징
        - 가능한 모든 경우에 프로그램 종료 가능
        - 모듈 형식 프로그래밍 지원
        - 무한 집합, 무하 스트링, 무한 리스트 등 무한 구조(infinite structure) 활용 가능

- 방정식 논증(equational reasoning)
    - 하스켈에서는 모든 게 함수의 계산으로 처리됨.
    - 프로그램의 실행과정은 연속된 방정식으로 간주 가능.
    - 이 방법을 이용하여 함수의 성질에 대한 논증 및 검증된 프로그램 구현 가능.

하스켈의 역사

- 1030년대: 람다 대수(lambda calculus)
    - 개발자: 알론조 처치(Alonzo Church)
    - 함수 계산 이론

- 1950년대: 리스프(Lisp)
    - 개발자: 존 맥카시(John McCarthy)
    - 최초 함수형 언어
    - 람다 대수 일부 활용
    - 변수 할당 유지

- 1960년대: ISWIM
    - 개발자: 피터 랜딘(Peter Landin)
    - 최초 순수 함수형 언어
    - 람다 대수에 기초
    - 변수 할당 없음

- 1970년대: FP
    - 개발자: 존 백커스(John Backus)
    - 함수형 언어
    - 고계 함수(higher-order functions) 지원
    - 프로그램에 대한 추론 지원

- 1970년대: ML
    - 개발자: 로빈 밀르너(Robin Milner) 중심
    - 최초 현대 함수형 언어
    - 유형 추론(type inference) 지원
    - 다형 유형(polymorphic types) 지원

- 1970년대 ~ 1980년대: Miranda
    - 개발자: 데이비드 터너(David Turner)
    - 다수의 소극적(lazy) 함수형 언어 개발

- 1987년: 하스켈(Haskell) 언어 개발 시작
    - 표준 소극적 순수 함수형 언어
    - 국제 하스켈 위원회 주도

- 1990년대: 유형 클래스(type classes)와 모나드(monads) 개발
    - 필립 와들러(Phillip Wadler) 중심
    - 하스켈 언어의 핵심 기능

- 2003년: 하스켈 보고서(Haskell Report)
    - 하스켈 위원회가 하스켈 안정 버전 공개
    - 2010년에 업데이트

- 2010년 이후: 하스켈 플랫폼
    - 하스켈 표준 배포
    - 라이브러리 지원
    - 다양한 새 특성 지원
    - 개발 툴 지원
    - 산업체 활용
    - 타 프로그래밍 언어에 영향
        - 파이썬, 자바스크립트 등 많은 프로그래밍 언어가 함수형 프로그래밍 지원

하스켈 프로그래밍 예제

리스트 항목 더하기

패터 매칭과 재귀로 sum 함수 정의하기

하스켈에서 `sum` 함수를 기본으로 제공한다. 따라서 약간 다른 이름으로 `sum` 함수를 구현한다.

```
sum_ []= 0
sum_ (n:ns) = n + sum_ ns
```

주의: 위 경고문은 `foldr` 을 사용하라고 추천하고 있음.

`sum_` 함수는 재귀(recursion)를 이용하여 정의되었다. 또한 임의의 리스트에 대해 `sum_` 함수는 항상 멈추게 되어 있다(왜 그럴까?). 앞으로 재귀함수를 자주 만날 것인데, 경우에 따라 멈추지 않는 재귀함수도 다뤄야 할 수도 있다.

```
sum_ [1, 2, 3]
sum_ [1..10]
```

유형(types)

하스켈에 다루는 모든 대상은 유형(type)을 갖는다. 타 프로그래밍언어에서 제공하는 자료형(data types) 보다 일반적인 개념이며, 단순히 어떤 데이터만 유형을 갖는 것이 아니라 하스켈에서 다루는 모든 것이 유형을 갖는다.

```
:type sum_
```

주의:

- `Num`: 숫자 집합들의 클래스. 정수, 자연수, 유리수, 실수 등.
- `forall p. Num p => [p] -> p`의 의미: 임의의 유형 `p` 에 대해, `p`가 `Num` 클래스에 속하면 `sum_` 함수는 유형 `p`의 리스트를 인자로 받아, `p` 유형의 값을 하나 생성한다.
- 결론적으로 다형성(polymorphism)과 클래스를 활용한다. 따라서 `sum_` 함수는 정수 뿐만 아니라, 자연수, 유리수, 실수 등에 대해서도 동일하게 작동한다.

앞서 정의했던 덧셈 함수의 유형은 다음과 같다.

```
:type addition
```

add3 의 유형에 주의한다.

```
:type add3
```

함수의 유형 정보을 이용하면 함수를 실행하기 전에 문제를 자동을 검출할 수 있다. 예를 들어 문자열들의 리스트를 `sum_` 함수의 인자로 사용하면 바로 오류가 발생된다. 문자열은 `Num` 클래스에 속하지 않기 때문이다.

```
sum_ ['a', 'b', 'c']
```

위 오류는 `Num Char` 가 성립하지 않는다. 즉 문자열 유형은 `Num` 클래스에 포함되지 않음을 경고한다.

리스트 정렬하기

정렬함수를 단순명료하게 정의할 수 있다. 둘째 인자는 오름차순/내림차순을 결정한다.

```haskell
-- qsort_ :: Ord a => [a] -> Bool -> [a]
qsort_ [] _    = []
qsort_ (x:xs) False = qsort_ ys False ++ [x] ++ qsort_ zs False
                      where
                          ys = [a | a <- xs, a <= x]
                          zs = [b | b <- xs, b > x]
qsort_ (x:xs) True = qsort_ zs True ++ [x] ++ qsort_ ys True
                      where
                          ys = [a | a <- xs, a <= x]
                          zs = [b | b <- xs, b > x]
```

`qsort_` 의 유형은 다음과 같다.

주의: `:type` 대신에 `:t` 명령문을 사용해도 된다.

```
:t qsort_
```

주의:

- ++: 리스트 덧셈 계산자, 즉, 두 리스트 이어붙이기
- `qsort_`는 재귀로 정의됨.

오름차순 정렬함수 `qsort` 와 내림차순 정렬함수 `qsortReverse` 를 다음과 같이 정의할 수 있다.

```
qsort xs = qsort_ xs False
qsortReverse xs = qsort_ xs True
```

숫자들의 리스트 정렬하기 예제

```
qsort [5, 2, 4, 3, 1]
qsortReverse [5, 2, 4, 3, 1]
```

`qsort_`, `qsort`, `qsortReverse` 타입은 다음과 같다.

```
:type qsort_
:type qsort
:type qsortReverse
```

주의:

- `Ord`: 순서(ordering)를 지원하는 집합들의 클래스
- 정수, 실수, 문자(`Char`), 문자열(`[Char]`) 유형 등이 `Ord` 클래스에 속한다.

문자열들의 리스트 정렬하기 예제

```
qsort ["abc", "cde", "ade"]
qsortReverse ["abc", "cde", "ade"]
```

연속적인 활동(actions) 처리하기

입출력 활동(actions)을 연속적으로 처리하는 함수 `seqn` 을 아래와 같이 정의할 수 있다.

주의: 활동(actions)은 입출력과 같이 부작용을 수반하는 함수의 호출을 의미한다.

```
actionSeqn :: [IO a] -> IO [a]
actionSeqn [] = return []
actionSeqn (act:acts) = do x <- act
                           xs <- actionSeqn acts
                           return (x:xs)
```

```
:type actionSeqn
```

위 함수는 입출력(input/output) 함수의 리스트를 인자로 받아 각 활동의 결과로 이루어진 값들의 리스트를 출력한다. 예를 들어, 문자 세 개를 입력 받아, 화면에 출력하는 활동은 아래와 같이 구현된다.

```
actionSeqn [getChar, getChar, getChar]
actionSeqn [getChar, getChar, getChar]
```

주의: `actionSeqn` 이 주피터 노트북에서 작동하는 방식과 일반 편집기에서 작동하는 방식이 다르다. 이에 대해 개발환경의 다르기에 그런 것 같다는 추정만 가능하다. 또한 repl.it 사이트에서는 입출력(IO)가 제대로 작동하지 않아 보인다. 이유는 모름.

일반화 시키기

입출력 이외에 다른 활동을 연속적으로 처리하는 함수로 일반화 시킬 수 있다. 실제로 `seqn` 의 유형을 지정하지 않으면 하스켈의 유형 추론을 통해 자동으로 일반화된다.

```
actionSeqn [] = return []
actionSeqn (act:acts) = do x <- act
                           xs <- actionSeqn acts
                           return (x:xs)
```

```
:type actionSeqn
```

위 결과는 임의의 **모나드**에 대해 `actionSeqn` 함수가 작동함을 보여준다. 그리고 `IO` 는 입출력(input/output)을 담당하는 특별한 모나드이다.

모나드로 구현 가능한 활용:

- 입출력
- 저장된 값 수정하기,
- 성공에 실패하기(fail to succeed),
- log 파일에 추가하기 등