

# 프로그래밍 언어 개론

## 과제 보고서

[PL00]HW10\_201601980\_KimSeongHan

개발환경 : Ubuntu

제출일 (2021-05-11)

201601980 / 김성한(00분반)

# <과제 설명>

## Practice

- First-class function을 읽을 수 있는 interpreter를 작성하시오.
  - First-class function이므로 함수를 읽는 interpreter 함수를 구현하지 않아도 됨
- `let rec interp_e (s : Store.t) (e : Ast.expr) : Store.value = (* write your code *)`
- (\* 추상메모리를 받아 값을 반환 \*)
- (\* Multiple Parameters in Syntactic Sugar \*) -> testcase의 parser에 구현되어있음
- (\* 연산에 대해 오류가 생길 수 있음 \*)
  - "Invalid addition: %a + %a"
  - "Invalid subtraction: %a - %a"
  - "Not a function: %a"
- `let interp (p : Ast.fvae) : Store.value = (* write your code *)`

### [과제 설명]

- 이번 과제는 지난주까지 구현했던 interpreter와 다르게 First-class function을 읽을 수 있는 interpreter를 구현하는 과제이다.
- First-class function 이란 함수가 값으로써 활용이 되는 것을 말한다. 지난주까지 진행한 F1VAE는 함수를 별도의 메모리에 저장하여 특수한 타입으로 해석하고, 함수를 변수에 저장하지도, 인자로 넘기지도, 반환하지도 못한다는 제약이 있었다.
- 반면 First-class function 은 함수 이름과 변수가 구별되지 않고 함수가 값으로써 활용될 수 있기에 함수를 변수에 저장하고 다른함수의 인자로 전달도 가능하며 함수가 다른 함수를 반환값으로 반환해줄수 있다.
- 이번 과제를 통해 구현할 FVAE 는 이러한 First-class function의 기능을 할 수 있도록 구현되게 된다. 이 과정에서 지난주와 다르게 함수를 읽는 용도의 interpreter가 불필요하게 되고 interp\_e와 interp함수를 구현하는 것이 목표이다.

## <코드 구현 및 실행 결과 화면>

```
k0s0a7@DESKTOP-MLPLCFD: * X + v
module F = Format

type expr =
| Num of int
| Add of expr * expr
| Sub of expr * expr
| Id of string
| LetIn of string * expr * expr
| App of expr * expr
| Fun of string * expr

type fvae =
| Prog of expr

let rec pp_e fmt e =
  match e with
  | Num n -> F.fprintf fmt "(Num %d)" n
  | Add (e1, e2) -> F.fprintf fmt "(Add %a %a)" pp_e e1 pp_e e2
  | Sub (e1, e2) -> F.fprintf fmt "(Sub %a %a)" pp_e e1 pp_e e2
  | Id x -> F.fprintf fmt "(Id %s)" x
  | LetIn (x, e1, e2) -> F.fprintf fmt "(LetIn %s %a %a)" x pp_e e1 pp_e e2
  | App (e1, e2) -> F.fprintf fmt "(App %a %a)" pp_e e1 pp_e e2
  | Fun (p, e) -> F.fprintf fmt "(Fun %s -> %a)" p pp_e e

let pp fmt (Prog e) =
  F.fprintf fmt "(Prog %a)" pp_e e
```

[ast.ml의 내부 코드]

- ast.ml의 내부코드로 타입들이 정의되어 있다.
- 지난 F1VAE와는 다르게 (expr,expr) 형태의 튜플을 가지는 App 과 (string,expr) 형태의 튜플을 가지는 Fun 이 추가되었으며 fvae 타입의 Prog의 형식이 expr로 바뀌었음을 확인할 수 있었다.
- 타입 외에는 타입별로 출력을 위한 출력문들이 선언되어 있다.

```

module F = Format

type value =
  | NumV of int
  | ClosureV of string * Ast.expr * t
and t = (string * value) list

let empty = []

let insert x n s = (x, n) :: s

let rec find x s =
  match s with
  | [] -> failwith ("Free identifier " ^ x)
  | (x', n) :: t -> if x' = x then n else find x t

let rec pp_v fmt v =
  match v with
  | NumV i -> F.fprintf fmt "%d" i
  | ClosureV (p, e, s) -> F.fprintf fmt "<λ%s.%a, %a>" p Ast.pp_e e pp s

and pp fmt s =
  let rec pp_impl fmt s =
    match s with
    | [] -> F.fprintf fmt "]"
    | (x, v) :: t -> F.fprintf fmt "(<λ%s.%a, %a>" x pp_v v pp_impl t
  in
  F.fprintf fmt "[ %a" pp_impl s

```

[store.ml의 내부 코드]

● 지난 F1VAE와 FVAE의 store.ml이 달라진 점은 FVAE는 F1VAE와는 다르게 함수도 값으로 취급되어 별도의 추상메모리 없이 값과 함께 store에 저장되게 된다. 그렇기에 store에서 값으로써 다룰 타입을 정의하기 위해 value를 NumV는 값에대한 타입, ClosureV는 함수에 대한 타입으로 정의하여 선언하였고 이때 ClosureV의 타입정의에 사용되는 t를 참조하기 위하여 and를 이용하였다. 이렇게 store.ml 하나의 파일 내부에 두 개의 타입을 and를 통해 정의해줌으로써 순환참조를 예방하고 이어 두 타입의 상호참조를 허용한다.

● 나머지 find 및 insert 그리고 empty는 이전 store.ml과 바뀐게 없으며 정의된 타입에 맞추어 해당하는 인자들에 대하여 find는 반환하고 insert는 저장하는 역할을 한다.

```
(* practice & homework *)
let interp (p : Ast.fvae) : Store.value =
  match p with
  | Prog expr -> interp_e [] expr
~
```

[interpreter.ml내부 interp 함수]

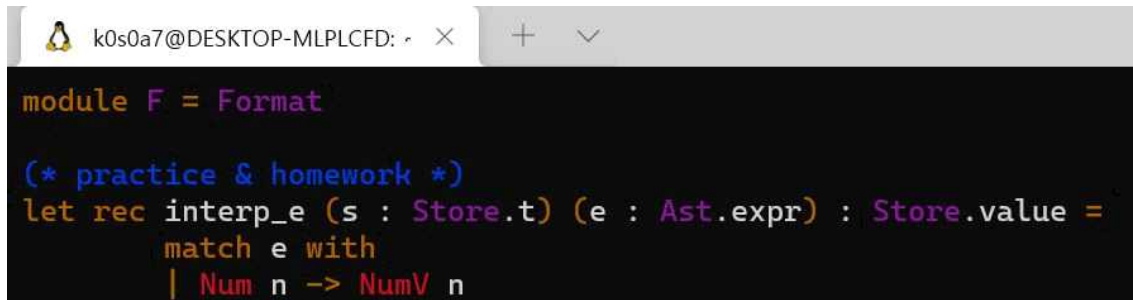
- interp함수로 초기 인자로 들어온 Prog타입을 interp\_e [] expr을 통해 interp\_e함수를 호출하고 값을 반환받는 시작함수의 기능을 한다. 이때 반환받는 값의 타입은 Store.value 타입으로 함수로 반환받을수도있고 아니라면 NumV(정수값) 타입으로 반환받을 수도 있다.

- Prog의 Bigstep-Operation은

$$\boxed{
 \begin{array}{c}
 \text{[Prog] : } e \\
 \frac{\emptyset \vdash e \Downarrow_E v}{\vdash e \Downarrow_P v}
 \end{array}
 }$$

으로 표현할 수 있으며 이는 빈 추상메모리에서 e를 계산한 결과가 v라고 가정하면 e 는 v로 계산하는 역할을 하게된다. 따라서 위의 interp함수에서 패턴매칭시 같은기능을 수행하도록 구현하였다.

(\*파일내부를 캡처하면 글씨가 너무 작아져서 부분으로 나누어 설명하였습니다.\*)



```
module F = Format

(* practice & homework *)
let rec interp_e (s : Store.t) (e : Ast.expr) : Store.value =
  match e with
  | Num n -> NumV n
```

[interpreter.ml내부 interp\_e 함수 - Num]

- interp\_e의 함수가 너무 길어 조금씩 끊어서 작성하였다. 먼저 interp\_e함수는 인자로 추상메모리와 expression을 인자로 넘겨받으며 이때 반환형타입은 Store.value이다.
- 인자로 들어온 expression e를 패턴매칭하여 재귀를 하는 방식으로 동작하면 먼저 Num타입의 expression일 경우에 대해 설명하겠다.
- Num타입의 expression의 Bigste-Operation은

$$\begin{array}{l} \text{[Num] : } n \\ \sigma \vdash n \Downarrow_E n \end{array}$$

으로 표현되며 이는 추상메모리에서  $n$ 은  $n$ 으로 계산된다는 의미이다. 단순히 Num  $n$ 이라면 반환형 타입인 Store.value 중 같은 타입을 가지는 NumV로 타입을 변환하여 반환해주면된다. 따라서 위의 코드가 작성되었다.

```

| Add (e1,e2) ->
  begin
    let exp1 = interp_e s e1 in
    let exp2 = interp_e s e2 in
    match exp1,exp2 with
    | NumV n1, NumV n2 -> NumV(n1 + n2)
    | _ , _ -> failwith (Format.asprintf "Invalid addition: %a + %a" Ast.pp_e e1 Ast.pp_e e2)
  end
| Sub (e1,e2) ->
  begin
    let exp1 = interp_e s e1 in
    let exp2 = interp_e s e2 in
    match exp1,exp2 with
    | NumV n1, NumV n2 -> NumV(n1 - n2)
    | _ , _ -> failwith (Format.asprintf "Invalid subtraction: %a - %a" Ast.pp_e e1 Ast.pp_e e2)
  end
end

```

[interpreter.ml 내부 interp\_e 함수 - Add, Sub]

● 다음은 Add와 Sub의 경우이다. 이 두 경우에는 동작이 거의 비슷하고 마지막에 더해주느냐 빼주느냐의 차이이기에 Add에 대해서만 설명하려고 한다.

● Add의 Bigstep-Operation을 보면

$$\begin{array}{c}
 e_1 \text{ 또는 } e_2 \text{의 계산 결과가 } n \text{이 아닌 } \langle \lambda x.e, \sigma \rangle \text{인 경우,} \\
 \text{runtime error!} \\
 \hline
 \hline
 \text{[Add] : } e_1 + e_2 \\
 \frac{\sigma \vdash e_1 \Downarrow_E n_1 \quad \sigma \vdash e_2 \Downarrow_E n_2}{\sigma \vdash e_1 + e_2 \Downarrow_E n_1 +_Z n_2}
 \end{array}$$

추상메모리에서  $e_1$ 을 계산한 결과가  $n_1$ 이고  $e_2$ 를 계산한 결과가  $n_2$ 이면 해당  $e_1+e_2$  는  $n_1+n_2$ 로 계산한다는 의미이다. 하지만 여기서 Store.value타입으로 반환이 될 수 있고 이 과정에서  $e_1, e_2$ 중 하나의 expression에 대해서라도 반환값이 ClosureV타입이라면 연산을 해줄 수 없다. 따라서 해당경우를 피하기위해 두 expression에 대한 연산결과가 모두 NumV인 경우에만 덧셈과 뺄셈(Sub의 경우)을 진행해 주었고 아니라면 failwith을 통해 error를 발생시켜주었다.

● 이와 같은 과정을 통해 Add와 Sub를 구현하였다.

```
| Id str -> Store.find str s
| LetIn (str,e1,e2) -> interp_e (Store.insert str (interp_e s e1) s) e2
```

[interpreter.ml 내부 interp\_e 함수 - Id, LetIn]

- 다음은 Id와 LetIn의 경우이다.
- 먼저 Id의 경우 Bigstep-Operation은

$$\boxed{\begin{array}{c} \text{[Id] : } x \\ \hline x \in \text{Domain}(\sigma) \\ \hline \sigma \vdash x \Downarrow_E \sigma(x) \end{array}}$$

으로 나타낼 수 있으며, 이는 추상메모리 s의 도메인에서 변수 x가 존재한다고 가정하면 x는 s(x)로 계산한다는 의미이고 코드에서는 Store.find x s 로 표현할 수 있다. 즉 Id str 이라면 Store.find str s 로 쉽게 구현할 수 있었다.

- LetIn의 경우 Bigstep-Operation은

$$\boxed{\begin{array}{c} \text{[LetIn] : } \text{let } x = e_1 \text{ in } e_2 \\ \hline \sigma \vdash e_1 \Downarrow_E v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Downarrow_E v_2 \\ \hline \sigma \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow_E v_2 \end{array}}$$

으로 나타낼 수 있으며, 추상메모리 s에서 e1을 계산한 결과가 v1이고, 추상메모리 s[x→v1]에서 e2를 계산한 결과가 v2라고 가정하면, 추상메모리 s에서 let x = e1 in e2 는 v2로 계산한다는 의미를 가진다. 이는 F1VAE에서의 동작과 같으며 달라진 점은 계산되는 타입이 int 형에서 Store.value타입으로 바뀐 것이다. 따라서 함수의 동작에 있어서 지난주 F1VAE에서의 코드를 수정할게 없었기에 달라진 점은 없다.



```

| App (e1,e2) ->
  begin
    let v1 = interp_e s e2 in
    match interp_e s e1 with
    | ClosureV (x,e3,s') -> interp_e (Store.insert x v1 s') e3
    | _ -> failwith (Format.asprintf "Not a function: %a" Ast.pp_e e1)

  end

| Fun (str,e) -> ClosureV(str,e,s)

```

[interpreter.ml 내부 interp\_e 함수 - App, Fun]

● 마지막은 App과 Fun이다.

● 먼저 App의 경우 두 개의 expr로 이루어진 튜플이며(e1,e2) 가지고 Bigstep-Operation은

$$\begin{array}{c}
 \text{[App]} : e_1 e_2 \\
 \frac{\sigma \vdash e_1 \Downarrow_E \langle \lambda x. e_3, \sigma' \rangle \quad \sigma \vdash e_2 \Downarrow_E v_1 \quad \sigma'[x \mapsto v_1] \vdash e_3 \Downarrow_E v_2}{\sigma \vdash e_1 e_2 \Downarrow_E v_2}
 \end{array}$$

이는 추상메모리 s에서 e1을 계산한결과가 ClosureV타입이고 추상메모리 s에서 e2를 계산한결과가 v1이고, 추상메모리 s'[x->v1]에서 e3를 계산한결과가 v2이면 추상메모리 s에서 e1 e2 는 v2로 계산된다는 의미를 가진다. 이를 위해 초기 v1에 e2를 계산한결과를 저장하였고 e1을 계산한결과를 패턴매칭하여 함수라면 함수가 정의된 해당시점에서의 추상메모리 s'에 x->v1을 insert하여 s'[x->v1]의 형태로 만들어주고 반환된 추상메모리 s'[x->v1]에서 e3를 계산한 결과를 반환해주었다. 만일 e1을 계산한 결과가 함수가 아닌 경우에는 failwith를 이용하여 예외처리를 해주었다.

● Fun의 경우 Bigstep-Operation은

$$\begin{array}{c}
 \text{[Fun]} : \lambda x. e \\
 \sigma \vdash \lambda x. e \Downarrow_E \langle \lambda x. e, \sigma \rangle
 \end{array}$$

으로 표현할 수 있으며 이는 추상메모리 s 에서 Fun(str,e)의 튜플형태를 Closure(str,e,s')로 계산한다는 의미이다. 여기서 s' 는 Fun을 통해 함수가 정의되는 시점에서의 추상메모리이다.

```

h0s0a7@DESKTOP-MLPLCFD:~/week10$ dune exec ./main.exe
AST: (Prog (Fun x -> (Add (Id x) (Num 1))))
RES: <λx.(Add (Id x) (Num 1)), [ ]>
AST: (Prog (App (Fun x -> (Add (Id x) (Num 1))) (Num 2)))
RES: 3
AST: (Prog (LetIn x (Fun x -> (Add (Id x) (Num 1))) (App (Id x) (Num 2))))
RES: 3
AST: (Prog (LetIn x (Fun x -> (Sub (Id x) (Num 1))) (App (Id x) (Num 3))))
RES: 2
AST: (Prog (LetIn foo (Fun x -> (Sub (Id x) (Num 1))) (LetIn bar (Fun x -> (Add (Id x) (Num 1))) (App (Id foo) (App (Id bar) (Num 3))))))
RES: 3
AST: (Prog (LetIn foo (Fun x -> (App (Id x) (Num 3))) (LetIn bar (Fun x -> (Add (Id x) (Num 1))) (App (Id foo) (Id bar))))))
RES: 4
AST: (Prog (LetIn foo (Fun x -> (Add (Id x) (Num 3))) (App (Id foo) (Num 7))))
RES: 10
AST: (Prog (LetIn z (Num 9) (LetIn foo (Fun x -> (Add (Id x) (Id z))) (App (Id foo) (Num 7))))))
RES: 16
AST: (Prog (LetIn z (Num 9) (LetIn foo (Fun x -> (Add (Id x) (Id z))) (LetIn z (Num 3) (App (Id foo) (Num 7))))))
RES: 16
AST: (Prog (LetIn z (Num 9) (LetIn foo (Fun x -> (Fun y -> (Add (Add (Id x) (Id z)) (Id y))) (App (App (Id foo) (Num 1)) (Num 5))))))
RES: 15
AST: (Prog (LetIn z (Num 9) (LetIn foo (Fun x -> (Fun y -> (Add (Add (Id x) (Id z)) (Id y))) (App (App (Id foo) (Num 3)) (Num 1))))))
RES: 13
AST: (Prog (LetIn z (Num 9) (LetIn foo (Fun x -> (Fun y -> (Sub (Add (Id x) (Id z)) (Id y))) (App (App (Id foo) (Num 5)) (Num 9))))))
RES: 5
AST: (Prog (LetIn x (Fun y -> (Fun z -> (App (Id y) (Id z)))) (App (App (Id x) (Fun x -> (Add (Id x) (Num 10)))) (Num 3))))
RES: 13
AST: (Prog (Fun x -> (Fun y -> (Fun z -> (Add (Add (Id x) (Id y)) (Id z))))))
RES: <λx.(Fun y -> (Fun z -> (Add (Add (Id x) (Id y)) (Id z)))) , [ ]>
AST: (Prog (App (Fun x -> (Fun y -> (Fun z -> (Add (Add (Id x) (Id y)) (Id z)))) (Num 1)))
RES: <λy.(Fun z -> (Add (Add (Id x) (Id y)) (Id z))) , [ (x, 1) ]>
AST: (Prog (App (App (Fun x -> (Fun y -> (Fun z -> (Add (Add (Id x) (Id y)) (Id z)))) (Num 1)) (Num 2)))
RES: <λz.(Add (Add (Id x) (Id y)) (Id z)) , [ (y, 2) (x, 1) ]>
AST: (Prog (App (App (App (Fun x -> (Fun y -> (Fun z -> (Add (Add (Id x) (Id y)) (Id z)))) (Num 1)) (Num 2)) (Num 3)))
RES: 6

```

[main.exe 실행 결과]

```

erp p1) in (* <λx.(Add (Id x, Num 1)) , [ ]> *)
erp p2) in (* 3 *)
erp p3) in (* 3 *)
erp p4) in (* 2 *)
erp p5) in (* 3 *)
erp p6) in (* 4 *)
erp p7) in (* 10 *)
erp p8) in (* 16 *)
erp p9) in (* 16 *)
erp p10) in (* 15 *)
erp p11) in (* 13 *)
erp p12) in (* 5 *)
erp p13) in (* 13 *)
erp p14) in (* <λx.(Fun y -> (Fun z -> (Add (Id x) (Add (Id y) (Id z))))), [ ]> *)
erp p15) in (* <λy.(Fun z -> (Add (Id x) (Add (Id y) (Id z)))) , [ (x, 1) ]> *)
erp p16) in (* <λz.(Add (Id x) (Add (Id y) (Id z))), [ (y, 2) (x, 1) ]> *)
erp p17) in (* 6 *)

```

[main.ml내부에 주석으로 작성된 결과]

- main.exe를 실행한 결과와 main.ml파일 내부에 주석으로 작성된 결과이다.
- 비교해보면 정답이 일치함을 확인할 수 있었다.

## <과제 제출 최종 코드>

```
module F = Format

(* practice & homework *)
let rec interp_e (s : Store.t) (e : Ast.expr) : Store.value =
  match e with
  | Num n -> NumV n
  | Add (e1,e2) ->
    begin
      let exp1 = interp_e s e1 in
      let exp2 = interp_e s e2 in
      match exp1,exp2 with
      | NumV n1, NumV n2 -> NumV(n1 + n2)
      | _ , _ -> failwith (Format.asprintf "Invalid addition: %a + %a" Ast.pp_e e1 Ast.pp_e e2)
    end
  | Sub (e1,e2) ->
    begin
      let exp1 = interp_e s e1 in
      let exp2 = interp_e s e2 in
      match exp1,exp2 with
      | NumV n1, NumV n2 -> NumV(n1 - n2)
      | _ , _ -> failwith (Format.asprintf "Invalid subtraction: %a + %a" Ast.pp_e e1 Ast.pp_e e2)
    end
  | Id str -> Store.find str s
  | LetIn (str,e1,e2) -> interp_e (Store.insert str (interp_e s e1) s) e2
  | App (e1,e2) ->
    begin
      let v1 = interp_e s e1 in
      match interp_e s e2 with
      | ClosureV (x,e3,s') -> interp_e (Store.insert x v1 s') e3
      | _ -> failwith (Format.asprintf "Not a function: %a" Ast.pp_e e2)
    end
  | Fun (str,e) -> ClosureV(str,e,s)

(* practice & homework *)
let interp (p : Ast.fvae) : Store.value =
  match p with
  | Prog expr -> interp_e [] expr
```

[interpreter.ml 내부 전체코드]

- 이번 과제 제출 코드에 해당하는 interpreter.ml 코드의 전체 내용이다.

## <과제 구현 중 어려웠던 부분 및 해결 과정>

```
| App (e1,e2) ->
  begin
    let v1 = interp_e s e2 in
    match interp_e s e1 with
    | ClosureV (x,e3,s') -> interp_e (Store.insert x v1 s') e3
    | _ -> failwith (Format.asprintf "Not a function: %a" Ast.pp_e e1)
  end
| Fun (str,e) -> ClosureV(str,e,s)
```

[interpreter.ml 내부 interp\_e 함수 - App, Fun]

- 과제에서 가장 어려웠던 부분은 interp\_e 함수 구현 중 패턴매칭시 App의 경우를 구현하는데에 생각을 가장 많이 하였다.
- 초기 Bigstep-Operation을 제대로 이해하지 못하고 e1의 계산결과로 반환된 함수가 가지는 추상메모리를 본래의 추상메모리 s 로 생각하여 함수가 제기능을 하지 못하였다.
- 그래서 이론강의도 다시 듣고 pdf파일도 읽어보면서 e3를 계산할 때 사용되는 추상메모리는 함수 x가 정의되는 시점에서의 추상메모리를 사용한다는 것을 간과하고 있었다는 것을 깨달았다.
- 제대로 생각하지 못하였다면 더 많은 시간을 고민했을지도 몰랐겠다는 생각이 들지만 Bigstep-Operation을 차근차근 제대로 이해하고 코드를 작성해야 한다는 점을 다시한번 머리에 새기게 되었다.

## <새로 고찰한 내용 또는 느낀점>

- 지난주에 비해 이번주의 경우 함수를 구현하는데 있어 조금은 편했던 것 같다.

지난주부터 Bigstep-Operation에 대해 조금 더 이해를하고 Bigstep-Operation을 함수작성에 사용되는 설명서로 인식을 하게되고 나니 Bigstep-Operation의 연산과정을 따라가면서 함수를 작성하게 되었다. 그렇게 따라가면서 작성하고 나면 함수에서 해당 부분에 대한 연산을 구현할 수 있었던 것 같다. Bigstep-Operation의 중요성을 다시 한번 느끼게 되었고 앞으로도 열심히 해야겠다는 생각이 든다.