프로그래밍 언어 개론
과제 보고서

[PL00]HW9_201601980_KimSeongHan

개발환경 : Ubuntu

제출일 (2021-05-06)

201601980 / 김성한(00분반)

<과제 설명>

Homework

앞서 정의한 F1VAE를 확장하여 함수 정의와 파라미터를 여러 개 받을 수 있도록 한다.
변화한 언어 정의에 맞게 interpreter를 변경하시오.

F1VAE 언어 정의

$p ::= \overline{d}\ e$
$d ::= def\ x\ \overline{x} = e$
$e ::= n \mid x \mid e + e \mid e - e \mid x(\overline{e})$

$(\overline{a} : 0\ \ldots\ more)$

```
e.g.
def add a b c = a + b + c
def mult a b c d = a * b * c * d
add(1, 3+4, x)
```

[과제 설명]

● 이번 과제는 지난주 실습에서 구현한 F1VAE를 정의한 interpreter에서 함수의 정의와 파라미터를 여러개 받을 수 있도록 수정하는 내용이다.

● 지난 실습에서는 함수의 정의와 파라미터를 하나씩 받는 interpreter를 정의하였으며 지난주의 언어의 정의와 다르게 몇가지 타입이 조금씩 수정되었다.

● Prog는 초기 함수정의에 대한 list와 expression을 가지며, funDef는 파라미터에 해당하는 부분이 파라미터의 list로 바뀌었다. 추가적으로 FCall(string*expr) 이었던 부분에서 expr은 expr list로 바뀌게 되었다.

● 기존 실습 내용과 바뀐 타입 및 이론자료에 제시된 BigStepOperation에 중점을 두고 구현을 시작하였다.

<코드 구현 및 실행 결과 화면>

k0s0a7@DESKTOP-MLPLCFD:

```
module F = Format

type expr =
  | Num of int
  | Add of expr * expr
  | Sub of expr * expr
  | Id of string
  | LetIn of string * expr * expr
  | FCall of string * expr list

type fundef =
  | FDef of string * string list * expr

type flvae =
  | Prog of fundef list * 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
  | FCall (f, elist) -> F.fprintf fmt "(FCall %s%a)" f (fun fmt y -> List.iter (fun x -> F.fprintf fmt " %a" pp_e x) y) elist

let pp_fd fmt (FDef (f, plist, e)) =
  F.fprintf fmt "(FDef %s%s %a)" f (List.fold_left (fun i x -> i ^ " " ^ x) "" plist) pp_e e

let pp fmt (Prog (fdlist, e)) =
  F.fprintf fmt "(Prog%a %a)" (fun fmt y -> List.iter (fun x -> F.fprintf fmt " %a" pp_fd x) y) fdlist pp_e e
```

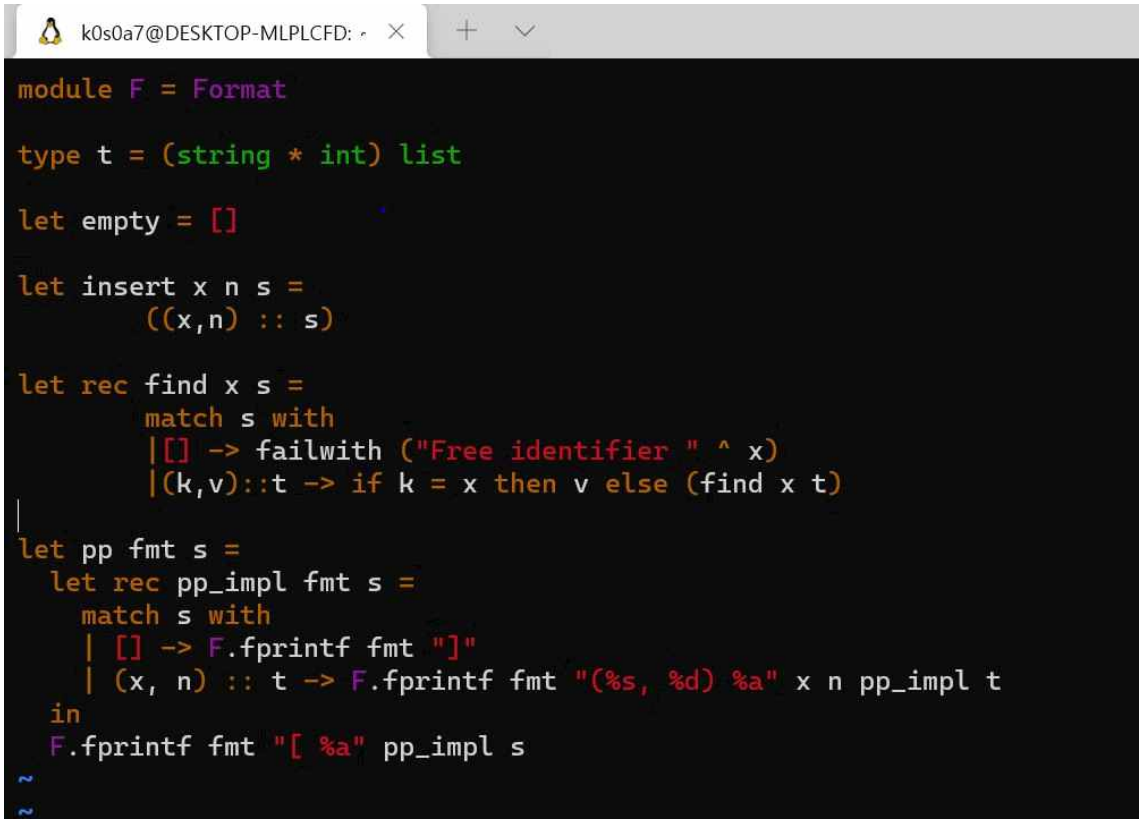
[ast.ml 내부 코드]

● 먼저 이번 과제의 ast.ml 코드이다.

● 지난주의 실습에서의 ast.ml과 크게 달라진 점은 없었지만 다른 점은 FCall 이 string 과 expr list의 튜플로 이루어졌고, FDef의 파라미터에 해당하는 부분이 string list로 바뀌게 되었다. 마지막으로 Prog는 fundef list 와 expr의 튜플로 이루어져 있는 모습을 확인할 수 있었다.

● FCall의 expr list의 원소들은 각각 함수에 인자로 들어온 expression들을 표현하는데에 이용된다.

● 위의 ast.ml 의 내용과 이론자료의 Big Step Operation을 이용하여 interpreter를 구현할 수 있었다.



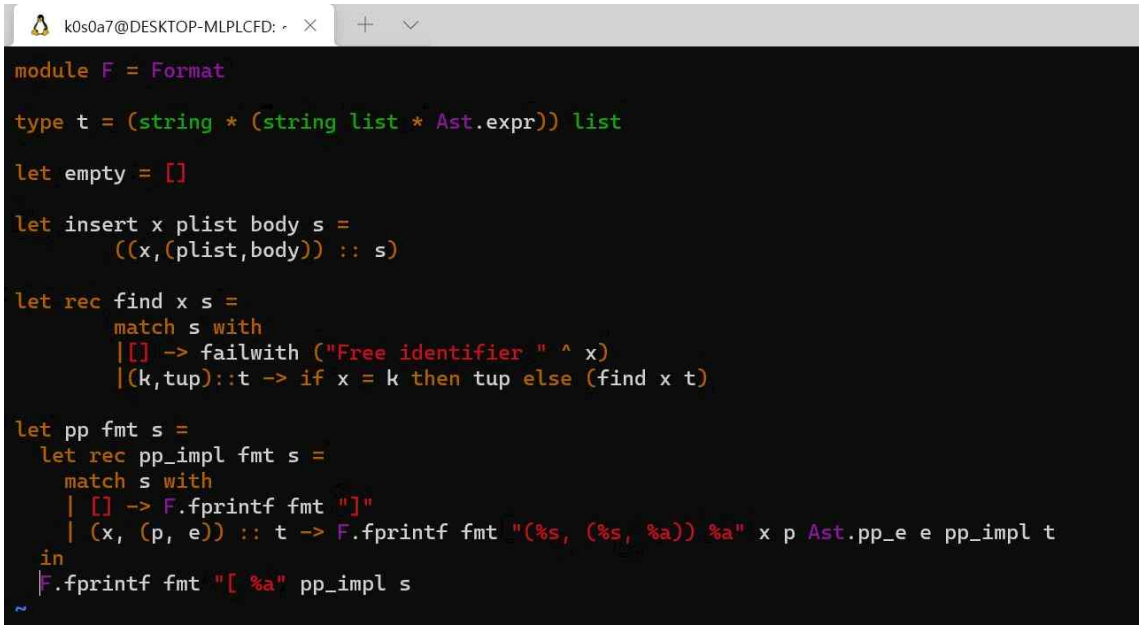
[store.ml 내부 코드]

● store.ml 은 변수를 저장하는 추상메모리로 지난 과제에서 구현했던 코드를 이용하였다.

● 초기 타입 t에 대한 정의가 있으며 (string,int)의 형태로 이루어진 튜플을 추상메모리에 저장한다는 의미이다.

● insert 함수는 (변수명,값)을 튜플로 묶어 추상메모리 s에 저장한다.

● find 함수는 변수명 x 와 추상메몰 s를 이용하여 해당 추상메모리 s에서 x와 같은 변수명을 가진 튜플을 찾아내어 해당 변수의 값을 반환하는 함수이다.



[fEnv.ml 의 내부코드]

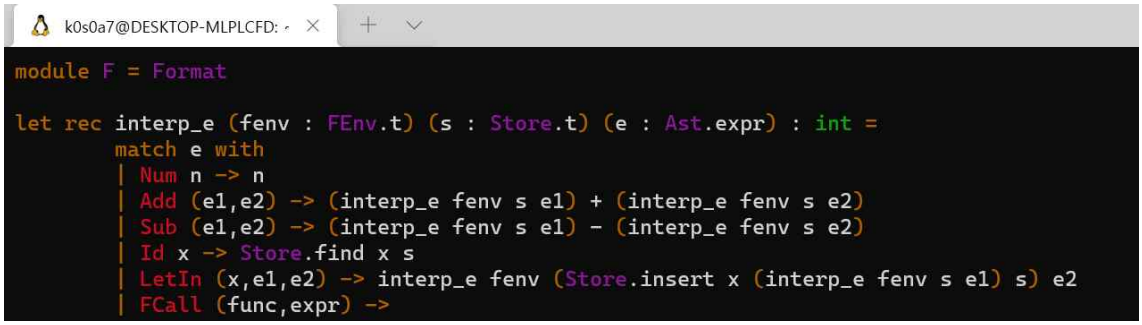
● fEnv.ml은 함수를 저장하는 추상메모리를 구현한 코드이다.

● 코드는 지난 실습시간에 구현한 코드를 사용하였으며 초기 타입 t가 정의되어 있다.

● 타입 t는 (string,(stringlist, expr)) 의 튜플 형태를 가지고 해당 타입대로 추상메모리에 저장되게 된다.

● insert함수는 인자로 들어온 함수명과 파라미터 리스트 그리고 함수몸체를 초기 정의된 타입 t 와 같은형태로 추상메모리 s에 저장하는 역할을 한다.

● find함수는 함수명 x와 추상메모리 s를 이용하여 추상메모리 s에서 x와 같은 함수명을 찾아 파라미터 리스트와 함수몸체로 이루어진 튜플을 반환하는 역할을 한다.



[interpreter.ml 의 interp_e함수 초반부]

● interp_e 함수는 인자로 함수의 추상메모리 fenv, 변수의 추상메모리 s, expression e를 인자로 받아 int 형 결과를 반환하는 함수이다.

● expression e 에대하여 pattern 매칭을 진행하여 연산을 수행한다.

● Num n 의 경우 말그대로 숫자이므로 n을 반환하고, Add와 sub에 대해서는 각각의 expression의 결과를 더해주고 빼주는 작업을 진행해준다. 또한 Id는 추상메모리 s에 저장된 변수의 값을 찾아 반환하고, LetIn의 경우 추상메모리 s에 변수에대한 expression을 계산하여 해당 변수의 값으로 s에 튜플형태로 저장한뒤 e2에 대한 연산을 진행해준다.
여기까지는 지난주까지 진행한 내용이였으며, 이번 주 추가된 내용은 FCall에 대한 내용이다.

*Fcall에 대한 내용까지 한번에 작성하면 글씨가 너무 작아져서 다음 장에 작성하였습니다.

```
| FCall (func,expr) ->
            begin
                match (FEnv.find func fenv) with
                |(param,body) ->
                                begin
                                    let rec storeData_impl param expr s=
                                            match param, expr with
                                            | para::t1, exp::t2 -> storeData_impl t1 t2 (Store.insert para (interp_e fenv s exp) s)
                                            | ([], _::_)|(_::_, []) -> failwith "Arity mismatched"
                                            | ([],[]) -> s
                                    in
                                    interp_e fenv (storeData_impl param expr s) body
                                end
            end
```

[interpreter.ml 의 interp_e함수 후반부 - FCall]

● FCall의 Big Step Operaion은

$$\textbf{[FCall]}\ :\ x(\bar{e})$$
$$\frac{\bar{e} = e_{21}, \ldots, e_{2k} \quad \Lambda,\sigma \vdash e_{21} \Downarrow_E n_{21} \ \ldots\ \Lambda,\sigma \vdash e_{2k} \Downarrow_E n_{2k} \quad \Lambda(x)=([x_1; \ldots; x_k],\ e_1) \quad \Lambda,[x_1 \mapsto n_{21}, \ldots, x_k \mapsto n_{2k}] \vdash e_1 \Downarrow_E n_1}{\Lambda,\sigma \vdash x(\bar{e}) \Downarrow_E n_1}$$

으로 표현된다. 초기 expresstion list에서 각 expression에 대한 연산결과를 얻고 함수정의에 저장된 파라미터 목록과 매칭하여 해당 파라미터와 연산된 expression의 값을 튜플의 형태로 추상메모리 s에 저장하여 전체 연산에 이용하고 있다.

● 이를 바탕으로 Fcall의 연산을 구현하였으며 초기 FCall의 구성요소인 func 함수명을 이용해 추상메모리 fenv에서 해당 함수의 파라미터 리스트와 함수몸체 튜플을 가져온다.

● 그 후 재귀문을 이용하여 해당 파라미터 리스트와 초기 주어진 expr리스트를 이용하여 각 파라미터에 expr에 해당하는 연산값을 추상메모리에 s 저장하여 끝에 모든 파라미터에 대해 expr 연산값을 저장하였다면 추상메모리 s를 반환하는 재귀함수 storeData_impl을 작성하였다.

● 그 후 마지막으로는 interp_e fenv (storeData_impl param expr s) body를 이용해 함수몸체에 대한 내용을 연산하는데 있어 파라미터의 값이 정의된 추상메모리를 이용할 수 있도록 storeData_impl함수를 호출해 인자값을 저장하고 함수몸체 연산을 해주도록 구현하였다.

```
let interp_d (fenv : FEnv.t) (fd : Ast.fundef) : FEnv.t =
          match fd with
          | FDef (str1, plist, expr) -> FEnv.insert str1 plist expr fenv
```

[interpreter.ml 의 interp_d 함수]

● interp_d 함수는 인자로 함수의 추상메모리와 함수정의에 대한 리스트가 들어오며 함수의 추상메모리를 반환한다.

● FDef 타입의 인자의 함수명 파라미터 리스트 그리고 expr을 함수의 추상메모리 fenv에 저장하고 반환하는 역할을 한다.

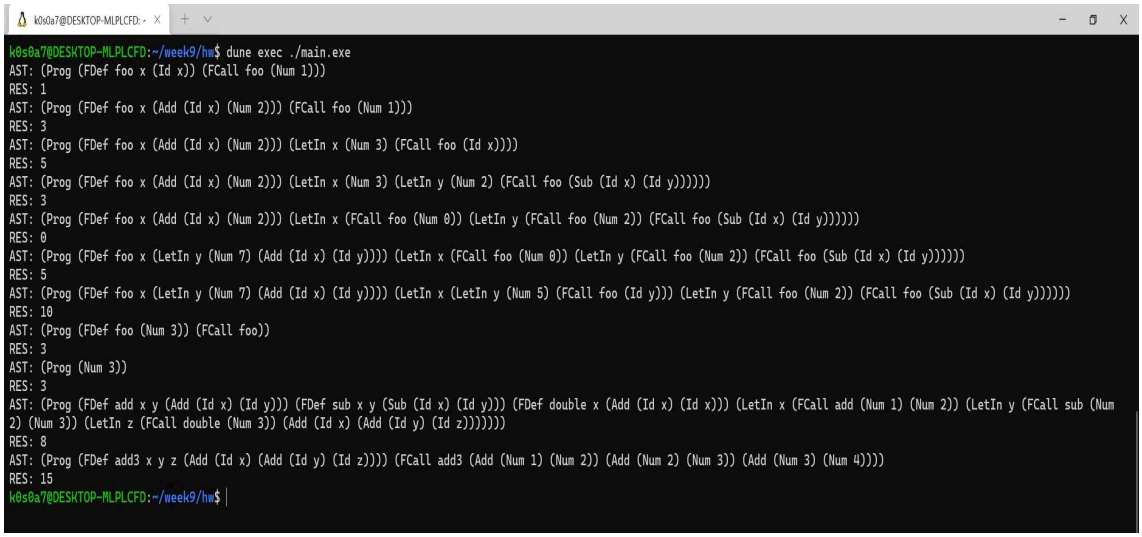
```
(* practice *)
let interp (p : Ast.f1vae) : int =
          match p with
          | Prog (fundef,expr) ->
                              begin
                                      let rec interp_dimpl fundef acc=
                                              match fundef with
                                              | [] -> acc
                                              | h::t -> interp_dimpl t (interp_d acc h)
                                      in
                                      interp_e (interp_dimpl fundef []) [] expr
                              end
```

[interpreter.ml 의 interp 함수]

● interp 함수는 인자로 Prog타입의 인자를 받아 fundef리스트와 expr을 가진다.

● 초기 실습에서의 내용과 다르게 fundef가 리스트의 형태여서 재귀문을 선언하여 해당 fundef의 내용을 추상메모리에 저장하였다. interp_dimpl은 해당 재귀를 진행하며 fundef 리스트에서 원소를 하나씩 뽑아 초기 빈 리스트로 정의될 acc에 추가하면서 마지막 fundef까지 acc에 추가되어 fundef리스트가 빈 경우에 acc를 반환하는 동작을 수행한다.

● 마지막으로는 “interp_e (interp_dimpl fundef []) [] expr”을 이용하여 함수의 추상메모리에 함수에 대한 정의들을 추가해주고, 초기 주어진 expr에 대한 연산을 진행할 수 있게 interp_e함수를 이용해주었다.



[main.exe 실행 결과]

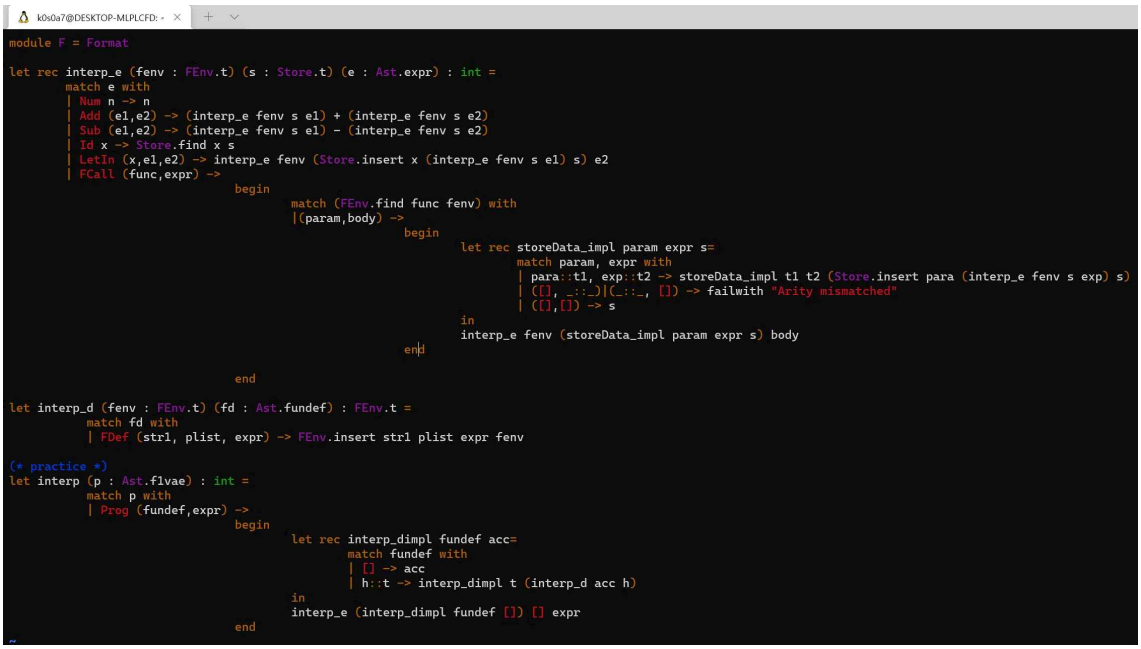
```
let _ = F.printf "AST: %a\n" pp p1 in
let _ = F.printf "RES: %n\n" (interp p1) in (* 1 *)
let _ = F.printf "AST: %a\n" pp p2 in
let _ = F.printf "RES: %n\n" (interp p2) in (* 3 *)
let _ = F.printf "AST: %a\n" pp p3 in
let _ = F.printf "RES: %n\n" (interp p3) in (* 5 *)
let _ = F.printf "AST: %a\n" pp p4 in
let _ = F.printf "RES: %n\n" (interp p4) in (* 3 *)
let _ = F.printf "AST: %a\n" pp p5 in
let _ = F.printf "RES: %n\n" (interp p5) in (* 0 *)
let _ = F.printf "AST: %a\n" pp p6 in
let _ = F.printf "RES: %n\n" (interp p6) in (* 5 *)
let _ = F.printf "AST: %a\n" pp p7 in
let _ = F.printf "RES: %n\n" (interp p7) in (* 10 *)
let _ = F.printf "AST: %a\n" pp p8 in
let _ = F.printf "RES: %n\n" (interp p8) in (* 3 *)
let _ = F.printf "AST: %a\n" pp p9 in
let _ = F.printf "RES: %n\n" (interp p9) in (* 3 *)
let _ = F.printf "AST: %a\n" pp p10 in
let _ = F.printf "RES: %n\n" (interp p10) in (* 8 *)
let _ = F.printf "AST: %a\n" pp p11 in
let _ = F.printf "RES: %n\n" (interp p11) in (* 15 *)
()
```

[main.ml 에 써있는 테스트케이스 결과]

● main.exe를 실행하였을때의 결과화면이다.

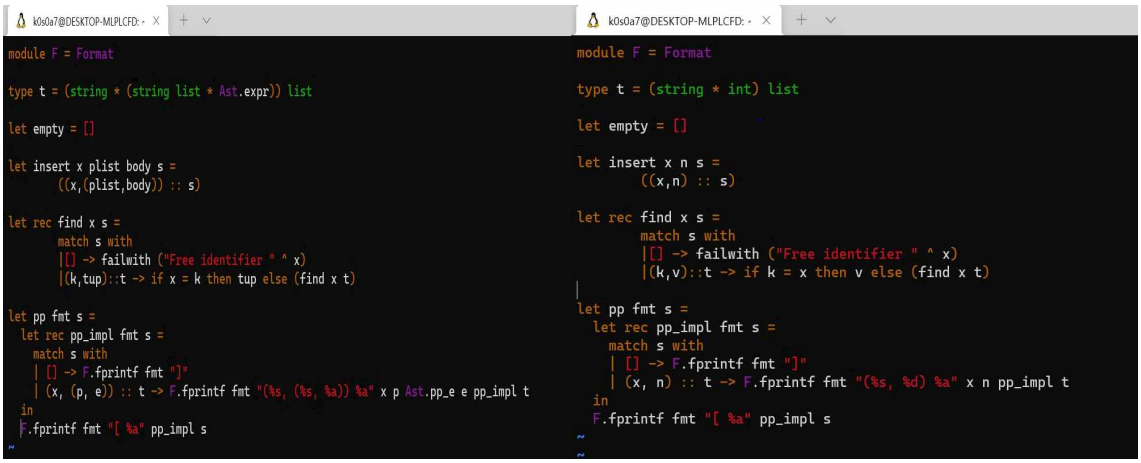
● main.ml에 써있는 테스트 케이스결과와 비교하여 똑같이 나와 옳은 결과가 나왔음을 판단할 수 있었다.

<과제 제출 최종 코드>



```
module F = Format

let rec interp_e (fenv : FEnv.t) (s : Store.t) (e : Ast.expr) : int =
        match e with
        | Num n -> n
        | Add (e1,e2) -> (interp_e fenv s e1) + (interp_e fenv s e2)
        | Sub (e1,e2) -> (interp_e fenv s e1) - (interp_e fenv s e2)
        | Id x -> Store.find x s
        | LetIn (x,e1,e2) -> interp_e fenv (Store.insert x (interp_e fenv s e1) s) e2
        | FCall (func,expr) ->
                                begin
                                        match (FEnv.find func fenv) with
                                        |(param,body) ->
                                                        begin
                                                                let rec storeData_impl param expr s=
                                                                        match param, expr with
                                                                        | para::t1, exp::t2 -> storeData_impl t1 t2 (Store.insert para (interp_e fenv s exp) s)
                                                                        | ([], _::_)|(_::_, []) -> failwith "Arity mismatched"
                                                                        | ([],[]) -> s
                                                                in
                                                                interp_e fenv (storeData_impl param expr s) body
                                                        end

                                end

let interp_d (fenv : FEnv.t) (fd : Ast.fundef) : FEnv.t =
           match fd with
           | FDef (str1, plist, expr) -> FEnv.insert str1 plist expr fenv

(* practice *)
let interp (p : Ast.flvae) : int =
           match p with
           | Prog (fundef,expr) ->
                                begin
                                        let rec interp_dimpl fundef acc=
                                                match fundef with
                                                | [] -> acc
                                                | h::t -> interp_dimpl t (interp_d acc h)
                                        in
                                        interp_e (interp_dimpl fundef []) [] expr
                                end
```

[interpreter.ml 내부 코드]



```
module F = Format

type t = (string * (string list * Ast.expr)) list

let empty = []

let insert x plist body s =
       ((x,(plist,body)) :: s)

let rec find x s =
       match s with
       |[] -> failwith ("Free identifier " ^ x)
       |(k,tup)::t -> if x = k then tup else (find x t)

let pp fmt s =
  let rec pp_impl fmt s =
    match s with
    | [] -> F.fprintf fmt "]"
    | (x, (p, e)) :: t -> F.fprintf fmt "(%s, (%s, %a)) %a" x p Ast.pp_e e pp_impl t
  in
  F.fprintf fmt "[ %a" pp_impl s
```

```
module F = Format

type t = (string * int) list

let empty = []

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

let rec find x s =
       match s with
       |[] -> failwith ("Free identifier " ^ x)
       |(k,v)::t -> if k = x then v else (find x t)

let pp fmt s =
  let rec pp_impl fmt s =
    match s with
    | [] -> F.fprintf fmt "]"
    | (x, n) :: t -> F.fprintf fmt "(%s, %d) %a" x n pp_impl t
  in
  F.fprintf fmt "[ %a" pp_impl s
```

[fEnv.ml 및 store.ml 내부 코드]

● 이번 과제 제출 코드에 해당하는 interpreter.ml 및 fEnv.ml 그리고 store.ml 코드의 전체 내용이다.

<과제 구현 중 어려웠던 부분 및 해결 과정>

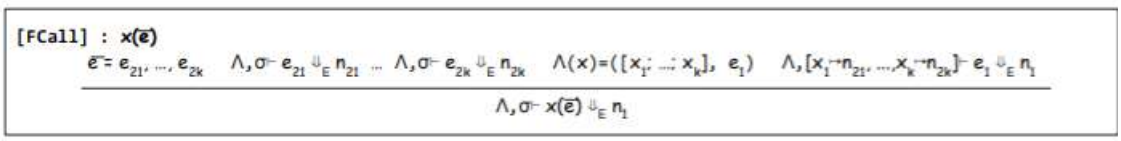
```
| FCall (func,expr) ->
            begin
                match (FEnv.find func fenv) with
                |(param,body) ->
                                begin
                                    let rec storeData_impl param expr s=
                                            match param, expr with
                                            | para::t1, exp::t2 -> storeData_impl t1 t2 (Store.insert para (interp_e fenv s exp) s)
                                            | ([], _::_)|(_::_, []) -> failwith "Arity mismatched"
                                            | ([],[]) -> s
                                    in
                                    interp_e fenv (storeData_impl param expr s) body
                                end
            end
```

[interpreter.ml 의 interp_e함수 후반부 - FCall]

● 이번 과제에서 가장 어려웠던 부분은 interp_e 함수의 FCall을 구현하는 것이었다.

● 초기 Bigstep Operation에 대해 대충 시험을 위해서만 이해를 했었는데, 이번 과제를 하면서 Bigstep Operation을 보고 함수들 및 타입에 대해 이해를 많이 하게 되었다.

● 아무렇게나 기능만 생각하고 구현 할 것이 아니라 Bigstep Operation을 보고 다시구현하자 마음먹고 코드를 처음부터 다 갈아 엎었다. 다시 한번 FCall의 BigStep Operaion을 확인해보면



으로 표현된다. 초기 expresstion list에서 각 expression에 대한 연산결과를 얻고 함수정의에 저장된 파라미터 목록과 매칭하여 해당 파라미터와 연산된 expression의 값을 튜플의 형태로 추상메모리 s에 저장하여 전체 연산에 이용하고 있다.

● 이와같이 Bigstep Operation을 이용하여 함수를 구현하고 이전 구현했던 내용 중 불확실한 내용들도 Bigstep Operation을 확인하며 수정하는 과정을 거쳐 과제 구현을 마무리하였다.

<새로 고찰한 내용 또는 느낀점>

● 이번 과제를 하면서 Bigstep Operation의 중요성이 굉장히 와닿았던 것 같다. 그동안 실습 혹은 과제를 진행하면서 함수의 기능에 초점을 맞추어 내 맘대로 구현하였는데 이번 과제를 진행하면서 Bigstep Operation을 확인하고 Bigstep Operation 나와있는 과정대로 구현하면 문제없이 구현되는 것을 확인하면서 앞으로는 이론 내용이라고 이해하고 지나치기만 했던 내용에 좀 더 집중해야겠음을 느끼게 되었다. 다음 실습 및 과제에서도 Bigstep Operation에 대한 이해를 바탕으로 함수를 구현하는 연습을 더 해야겠다.