프로그래밍 언어 개론
과제 보고서

[PL00]HW14_201601980_KimSeongHan

개발환경 : Ubuntu

제출일 (2021-06-09)

201601980 / 김성한(00분반)

<과제 설명>

Homework : interpreter.ml

```
let rec interp_e (e : Ast.expr) ((env, mem) : Env.t * Mem.t) : Mem.value = (* write your code *)


let rec interp_s (stmt : Ast.stmt) ((env, mem) : Env.t * Mem.t) : Env.t * Mem.t = (* write your code *)
          # VarDeclStmt
                  Env.mem()함수를 통해 변수할당 여부를 확인
                  Env.new_address()함수를 통해 새로운 주소를 받아옴
          # StoreStmt
                  * e = e 에 해당하는 구문을 구현

let interp (p : Ast.program) : Env.t * Mem.t = (* write your code *)
```

Testcase : https://github.com/SeoyeonKang/UPL2021/tree/master/week14

[과제 제시 내용]

● 이번 과제는 지난 주와 마찬가지로 Imperative Language인 C언어를 정의하는 것이다.

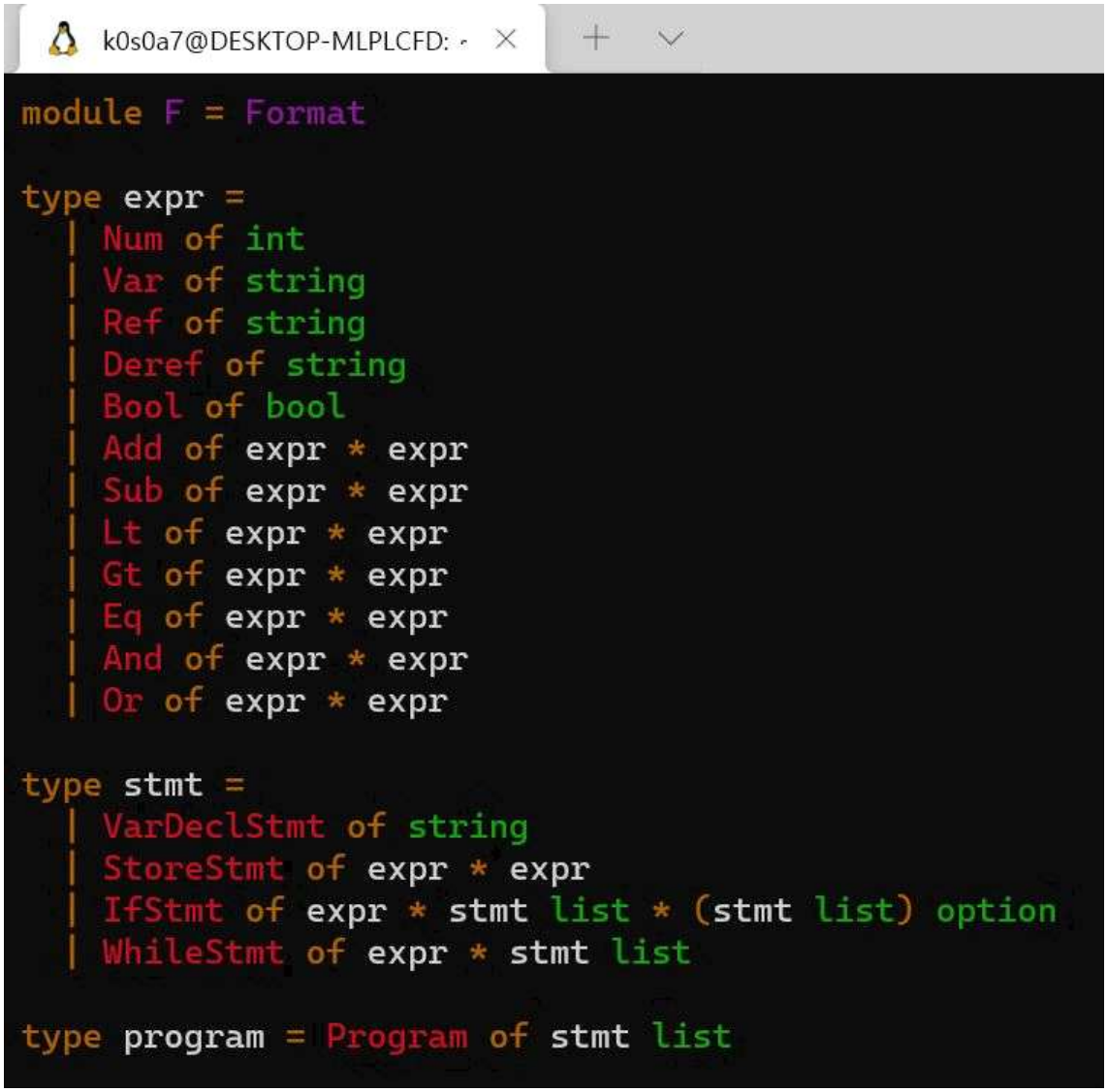
● 이번 실습에서는 지난주 실습에 이어 MiniC interpreter를 구현하되, 이번 시간에는 반복문과 포인터를 정의한 MiniC의 interpreter를 구현하는 것이다.

● 이번 시간에는 지난주까지의 추상메모리 store가 아닌 env와 mem을 사용하게 되며, 이론시간에서 배웠듯이 env는 주소를 저장하고, mem 은 그 주소에 해당하는 메모리 위치에서의 값을 저장한다.

● interpreter의 세함수 interp_e, interp_s, interp를 정의하게 되며 각각은 세가지 Sementic Relation을 정의하게 된다. 즉, 세가지 Semantic Relation이 각각의 기능을 하도록 구현해야 한다. (이번주는 반목문과 포인터 정의)

● 구현의 시작은 interp함수부터 시작하였다.

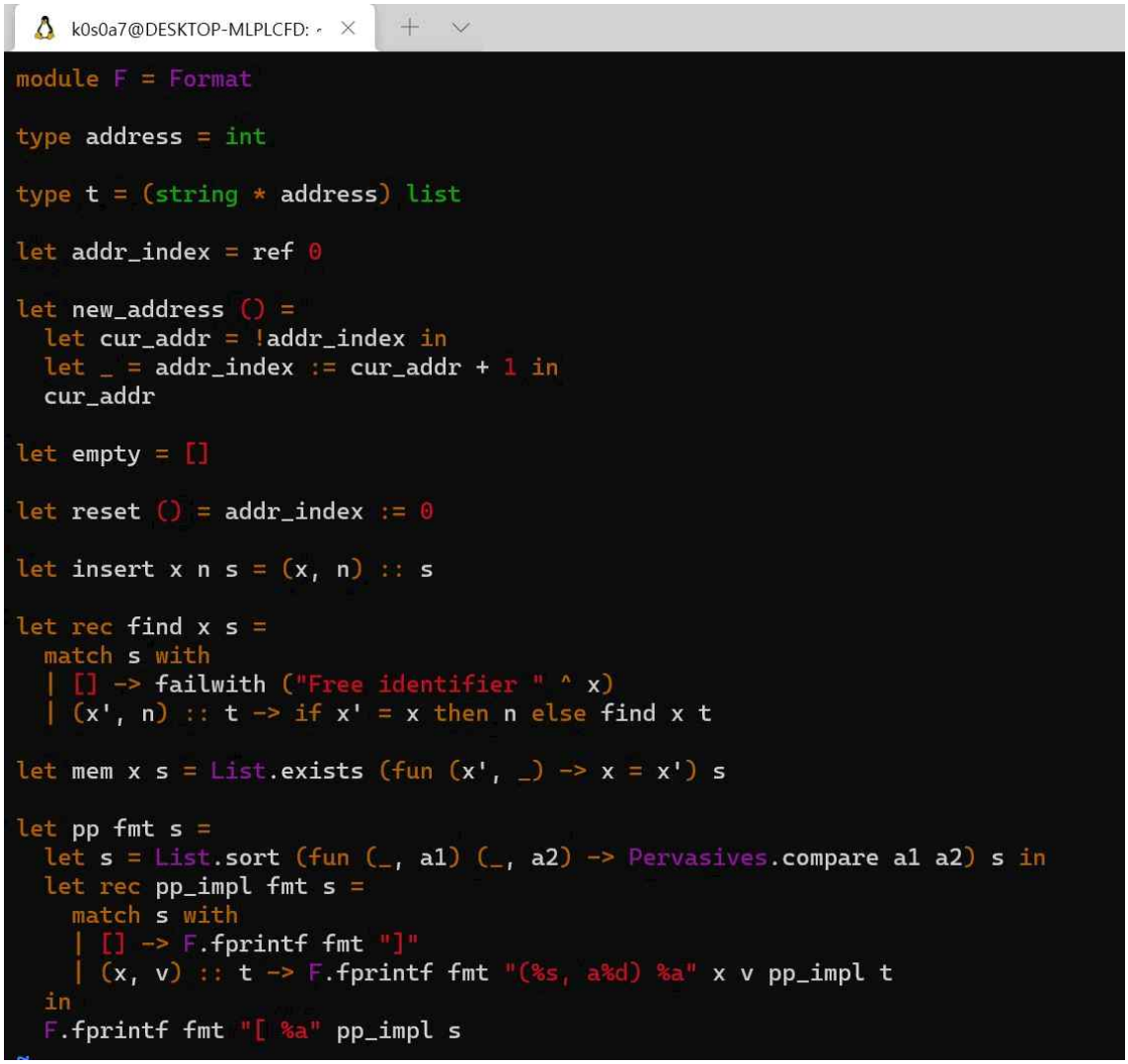
<코드 구현 - 해결 방법>



[ast.ml 내부 타입정의]

● 이번 실습에서의 ast.ml 내부 코드이다.

● 지난주와 다르게 반복문과 포인터의 정의를 위해 Ref, Deref 가 expr에 추가되었고, stmt에 VarDeclStmt, StoreStmt 그리고 WhileStmt가 추가되었음을 확인할 수 있었다.



[env.ml 내부 코드]

● env.ml의 내부코드 모습이다. env 추상메모리는 변수의 메모리 주소를 저장하며 기존의 store.ml과 다른 함수들이 추가되었음을 확인할 수 있었다.

● 이를 활용하여 interpreter에서 변수의 메모리 주소를 저장하고 확인할 수 있었으며, new_address() 함수를 이용해 새로운 주소값을 할당하고 find를 통해 해당 변수의 주소를 확인해 볼 수 있었다.

[mem.ml 내부 코드]

● mem.ml의 내부코드이다. mem.ml은 어떤 주소에 저장된 값을 참조할 수 있는 추상메모리이다.

● mem.value의 타입으로 NumV, BoolV, AddressV가 선언되어 있으며 기존 store.ml과 비슷한 동작을 하는 함수들로 구성이 되어있다.

● interpreter에서 주소값을 이용해 해당 주소의 메모리에 저장된 값을 참조하는데 사용될 수 있다.

```
(* practice & homework *)
let interp (p : Ast.program) : Env.t * Mem.t =
      match p with
      | Program(li) ->
                  begin
                        let rec interp_imp li (e, m) =
                              match li with
                              | h::t -> (interp_imp t (interp_s h (e, m)))
                              | [] -> (e, m)
                        in
                        interp_imp li ([], [])
                  end
```

[interpreter.ml - interp함수]

● interp함수의 구성 모습이다.

● Program의 Bigstep-Operation은

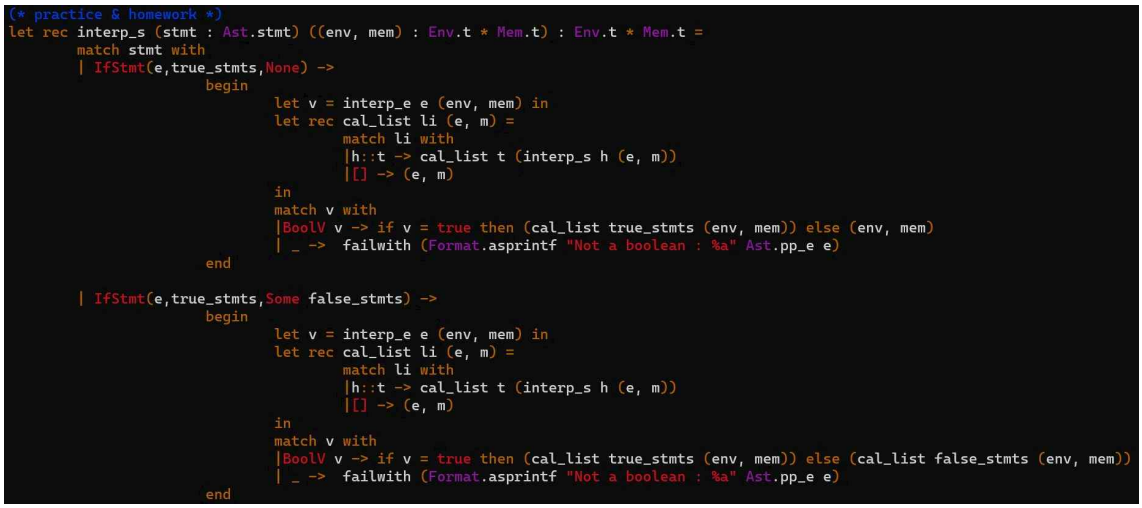
[Rule Program] : $s_1\ s_2 \dots s_n$
빈 Env와 Mem에서 s_1을 실행한 결과가 σ_1, M_1 이고,
Env σ_1와 Mem M_1 에서 s_2을 실행한 결과가 σ_2, M_2 이고, …
Env σ_{n-1}와 Mem M_{n-1} 에서 s_n을 실행한 결과가 σ_n, M_n이면,
$s_1\ s_2 \dots s_n$ 를 실행하면 σ_n, M_n 를 반환

[Program] : $s_1\ s_2 \dots s_n$

$$\frac{\sigma_0, M_0 \vdash s_1 \Rightarrow_S \sigma_1, M_1 \quad \sigma_1, M_1 \vdash s_2 \Rightarrow_S \sigma_2, M_2 \quad \dots \quad \sigma_{n-1}, M_{n-1} \vdash s_n \Rightarrow_S \sigma_n, M_n}{\vdash s_1\ s_2 \dots s_n \Rightarrow_P \sigma_n, M_n}$$

이며 초기 빈 Env와 Mem에서 s1을 실행하여 나온 Env1, Mem1에서 다시 s2를 실행하고 이를 마지막 statement까지 반복해 얻게된 Env, Mem의 튜플을 반환하게 된다.

● 비교적 간단한 동작으로 구성이 되며 인자로 주어진 statement list의 원소별로 interp_s를 진행하기 위해 interp_imp 재귀함수를 작성하여 해당 함수의 인자로 빈 Env, 빈 Mem을 할당해주었다.



```
(* practice & homework *)
let rec interp_s (stmt : Ast.stmt) ((env, mem) : Env.t * Mem.t) : Env.t * Mem.t =
        match stmt with
        | IfStmt(e,true_stmts,None) ->
                        begin
                                let v = interp_e e (env, mem) in
                                let rec cal_list li (e, m) =
                                        match li with
                                        |h::t -> cal_list t (interp_s h (e, m))
                                        |[] -> (e, m)
                                in
                                match v with
                                |BoolV v -> if v = true then (cal_list true_stmts (env, mem)) else (env, mem)
                                | _ ->  failwith (Format.asprintf "Not a boolean : %a" Ast.pp_e e)
                        end

        | IfStmt(e,true_stmts,Some false_stmts) ->
                        begin
                                let v = interp_e e (env, mem) in
                                let rec cal_list li (e, m) =
                                        match li with
                                        |h::t -> cal_list t (interp_s h (e, m))
                                        |[] -> (e, m)
                                in
                                match v with
                                |BoolV v -> if v = true then (cal_list true_stmts (env, mem)) else (cal_list false_stmts (env, mem))
                                | _ ->  failwith (Format.asprintf "Not a boolean : %a" Ast.pp_e e)
                        end
```

[interpreter.ml - interp_s_1(함수가 너무 길어 부분별로 설명하겠습니다.)]

● interpreter.ml 내부 interp_s함수의 IfStmt 패턴 부분이다.

● 지난주 실습에서 '변수 정의 및 할당 그리고 분기문' 에 대해 정의할 때의 코드에서 크게 전체적인 코드의 동작이 바뀌지는 않았다.

● 미세하게 바뀐 부분은 이번 주 실습에서는 추상메모리로써 Env 와 Mem 두가지 추상메모리를 이용하기에 interp_s의 인자로 (Env, Mem) 형태의 튜플이 들어오게 되고 이를 위해 기존 코드에서 cal_list 재귀문 그리고 interp_e 의 인자 및 기존 store를 이용했던 부분을 (Env,Mem) 튜플로 바꿔주었다.

● 간단하게 동작을 소개하자면 IfStmt패턴에서 false_stmts가 None인 경우와 Some false_stmts 인 경우로 두가지 패턴으로 나뉘는데, 크게 보면 false_stmts가 없는 경우 false인 경우에는 그 자체의 추상메모리 s를 반환하며 false_stmts가 있는 경우에는 해당 stmts에 대해 cal_list를 진행한 결과를 반환한다.

,

```
| VarDeclStmt(x) ->
            begin
                let check = Env.mem x env in
                if check = true then failwith (Format.asprintf "%s is already declared." x) else ((Env.insert x (Env.new_address ()) env), mem)
            end
```

[interpreter.ml - interp_s_2]

● interp_s함수의 VarDeclStmt 패턴부분이다.

● VarDeclStmt 의 Bigstep-Operation은

[Rule VarDeclStmt] : var x
x 가 σ의 도메인에 포함되지 않고,
a 는 새 주소 값이라고 가정하면 ($a \notin Range(\sigma)$),
Env σ와 Mem M에서 var x는 $\sigma[x \mapsto a]$, M 로 계산

[VarDeclStmt] : var x

$$\frac{x \notin Domain(\sigma) \quad a \notin Range(\sigma)}{\sigma,\ M \vdash \text{var } x \Rightarrow_S \sigma[x \mapsto a],\ M}$$

으로 정의되며 초기 check 변수에 Env.mem 함수를 통해 해당 변수 x가 env에 존재하는지에 대한 boolean 결과를 저장한다.

● 이후 check 가 true라면 이미 존재한다는 의미이므로 failwith을 이용해 runtime-error를 발생시키고, 아니라면 Env.new_address()를 이용해 새 주소값을 가져와 env에 저장하게 된다.

```
| StoreStmt(e1,e2) ->
                begin
                        let a = interp_e e1 (env, mem) in
                        let v = interp_e e2 (env, mem) in
                        match a with
                        | AddressV ad -> (env, (Mem.insert ad v mem))
                        | _ -> failwith (Format.asprintf "Not a memory address : %a" Ast.pp_e e1)
                end
```

[interpreter.ml - interp_s_3]

● StoreStmt 패턴의 구성모습이다.

● StoreStmt의 Bigstep-Operation은

[Rule StoreStmt] : $*e_1 = e_2$
Env σ와 Mem M에서 e_1이 주소 값 a 로 계산되고,
Env σ와 Mem M에서 e_2가 v 로 계산된다고 가정하면,
Env σ와 Mem M에서 $*e_1 = e_2$는 σ, M[a↦v]로 계산

[StoreStmt] : $*e_1 = e_2$

$$\frac{\sigma,\ M \vdash e_1 \Downarrow_E a \quad \sigma,\ M \vdash e_2 \Downarrow_E v}{\sigma,\ M \vdash *e_1 = e_2 \Rightarrow_S \sigma,\ M[a \mapsto v]}$$

으로 정의되며 e1을 연산한 결과를 a에 e2를 연산한 결과를 v에 저장하고 a에 대해 패턴매칭을 수행한다.

● 만일 a의 타입이 AddressV라면 mem에 a가 a의 이름(코드에서 ad)으로 v 값을 저장하고, env와 갱신된 mem의 튜플을 반환한다. 만일 a의 타입이 AddressV가 아니라면 failwith을 이용해 runtiem-error를 발생시킨다.

```
| WhileStmt(e, li) ->
            begin
                    let v = interp_e e (env, mem) in
                    let rec cal_list li (ee, mm) =
                          match li with
                          | h::t -> cal_list t (interp_s h (ee, mm))
                          | [] -> (ee, mm)
                    in
                    match v with
                    | BoolV v -> if v = true then interp_s (WhileStmt (e,li)) (cal_list li (env, mem)) else (env, mem)
                    | _ -> failwith (Format.asprintf "Not a boolean : %a" Ast.pp_e e)
            end
```

[interpreter.ml - interp_s_4]

● WhileStmt의 구성모습이다.

● WhileStmt의 Bigstep-Operation은

[Rule While1] : while(e) $\overline{s}$
추상메모리 σ에서 **e** 를 계산한 결과가 **true**이고,
추상메모리 σ에서 **s** 를 실행한 결과가 σ_1이고,
추상메모리 σ_1에서 **while(e)** $\overline{s}$를 실행한 결과가 σ_2이라고 가정하면,
추상메모리 σ에서 **while(e)** $\overline{s}$를 실행하면 σ_2를 반환

[while1] : while(e) $\overline{s}$

$$\frac{\sigma \vdash e \Downarrow_E \text{true} \quad \sigma \vdash \overline{s} \Rightarrow_S \sigma_1 \quad \sigma_1 \vdash \text{while(e)}\,\overline{s} \Rightarrow_S \sigma_2}{\sigma \vdash \text{while(e)}\,\overline{s} \Rightarrow_S \sigma_2}$$

[Rule While2] : while(e) $\overline{s}$
추상메모리 σ에서 **e** 를 계산한 결과가 **false**라고 가정하면,
추상메모리 σ에서 **while(e)** $\overline{s}$를 실행하면 σ를 반환

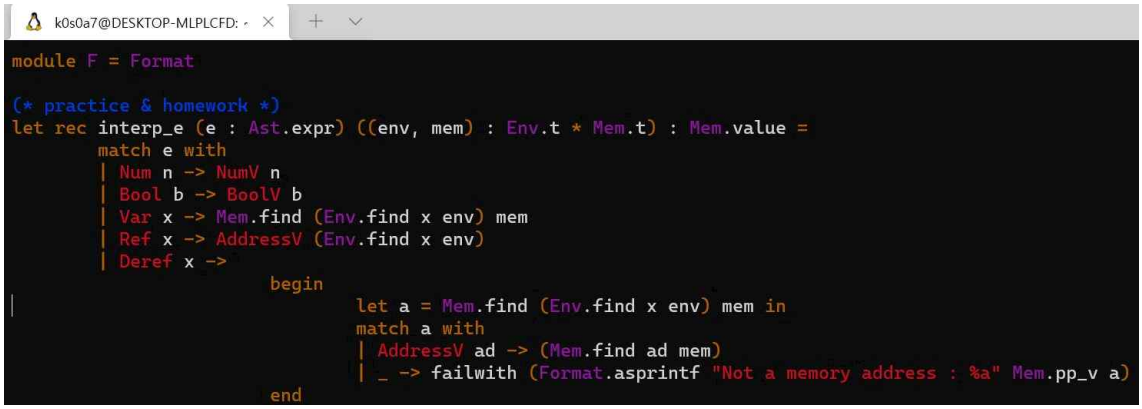
[while2] : while(e) $\overline{s}$

$$\frac{\sigma \vdash e \Downarrow_E \text{false}}{\sigma \vdash \text{while(e)}\,\overline{s} \Rightarrow_S \sigma}$$

으로 정의되며 각각의 경우는 e를 연산한 결과 v가 true인 경우와 false인 경우의 동작이다.

● v가 true인 경우 (env,mem)의 추상메모리에서 Statement 리스트 실행결과가 s1이면, s1 추상메모리 상에서의 while(e,li)를 연산한 결과를 반환해주면 된다. 이를 코드로 옮겼으며 반대로 false인 경우에는 단순히 기존 (Env,Mem)튜플을 반환해주면 된다. 이는 if-then-else 구문을 사용하여 구현하였다.

● 만일 v가 BoolV타입이 아닌 경우에는 failwith을 이용해 runtime-error를 발생시켰다.



[interpreter.ml - interp_e_초반부]

● 이번 MiniC interpreter에서는 반복문과 분기문이 추가 정의되면서 interpreter 함수들의 인자가 store에서 Env와 Mem으로 이루어진 튜플로 변화된 모습을 확인할 수 있었다.

● 위의 사진에서 보이는 코드는 interp_e 코드의 초반부이다. Num, Bool 의 동작은 지난 MiniC를 구현할 때 와 같은 형식으로 작성되었으며 각각의 타입은 Mem.value에 해당하는 NumV 그리고 BoolV로 변환되어 반환된다.

● 이번에 추가된 Var, Ref 그리고 Deref 패턴의 Bigstep-Operation은

[Rule Var] : x
Env σ와 Mem M에서 x가 σ의 도메인에 존재하면,
Env σ와 Mem M에서 x는 M(σ(x))로 계산

[Var] : x

$$\frac{x \in \mathrm{Domain}(\sigma)}{\sigma,\ M \vdash x \Downarrow_E M(\sigma(x))}$$

[Rule Ref] : &x
Env σ와 Mem M에서 x가 σ의 도메인에 존재하면,
Env σ와 Mem M에서 x는 σ(x)로 계산

[Ref] : &x

$$\frac{x \in \mathrm{Domain}(\sigma)}{\sigma,\ M \vdash x \Downarrow_E \sigma(x)}$$

[Rule DeRef] : *x
Env σ와 Mem M에서 x가 σ의 도메인에 존재하고,
M(σ(x))가 주소 값 a라고 가정하면,
Env σ와 Mem M에서 *x는 M(a)로 계산

[DeRef] : *x

$$\frac{x \in \mathrm{Domain}(\sigma) \quad M(\sigma(x)) = a}{\sigma,\ M \vdash *x \Downarrow_E M(a)}$$

으로 정의 되며 var는 x의 주소를 가져와 Mem에서 해당 주소에해당하는 값을 가져오도록 Mem.find함수를 이용해 구현하였고, Ref는 Env.find를 통해 x의 주소를 가져와 AddressV타입으로 변환하여 반환해주었다.

● 마지막으로 Deref는 x의 주소를 가져와 Mem에서 해당 주소의 값을 가져와 a에 저장하고 a에 대해 패턴매칭을 하여 만일 a가 AddressV타입인 경우 Mem에서 다시 한번 해당 주소에 대한 값을 가져와 반환해준다. 만일 a가 AddressV 타입이 아닌 경우에는 failwith을 이용하여 runtime-error를 발생시켜주었다.

```
| Add(e1,e2) ->
            begin
                    let v1 = interp_e e1 (env,mem) in
                    let v2 = interp_e e2 (env,mem) in
                    match v1,v2 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 v1 = interp_e e1 (env,mem) in
                    let v2 = interp_e e2 (env,mem) in
                    match v1,v2 with
                    | NumV n1, NumV n2 -> NumV(n1 - n2)
                    | _ , _ -> failwith (Format.asprintf "Invalid subtraction: %a - %a" Ast.pp_e e1 Ast.pp_e e2)
            end
|Lt(e1,e2) ->
            begin
                    let v1 = interp_e e1 (env,mem) in
                    let v2 = interp_e e2 (env,mem) in
                    match v1,v2 with
                    | NumV n1, NumV n2 -> BoolV(n1 < n2)
                    | _ , _ -> failwith (Format.asprintf "Invalid less-than: %a < %a" Ast.pp_e e1 Ast.pp_e e2)
            end
|Gt(e1,e2) ->
            begin
                    let v1 = interp_e e1 (env,mem) in
                    let v2 = interp_e e2 (env,mem) in
                    match v1,v2 with
                    | NumV n1, NumV n2 -> BoolV(n1 > n2)
                    | _ , _ -> failwith (Format.asprintf "Invalid greater-than: %a > %a" Ast.pp_e e1 Ast.pp_e e2)
            end
|Eq(e1,e2) ->
            begin
                    let v1 = interp_e e1 (env,mem) in
                    let v2 = interp_e e2 (env,mem) in
                    match v1,v2 with
                    | NumV n1, NumV n2 -> BoolV(n1 == n2)
                    | BoolV b1, BoolV b2 -> BoolV(b1 == b2)
                    | _ , _ -> failwith (Format.asprintf "Invalid equal-to: %a == %a" Ast.pp_e e1 Ast.pp_e e2)
            end
|And(e1,e2) ->
            begin
                    let v1 = interp_e e1 (env,mem) in
                    let v2 = interp_e e2 (env,mem) in
                    match v1,v2 with
                    | BoolV b1, BoolV b2 -> BoolV(b1 && b2)
                    | _ , _ -> failwith (Format.asprintf "Invalid logical-and: %a && %a" Ast.pp_e e1 Ast.pp_e e2)
            end
|Or(e1,e2) ->
            begin
                    let v1 = interp_e e1 (env,mem) in
                    let v2 = interp_e e2 (env,mem) in
                    match v1,v2 with
                    | BoolV b1, BoolV b2 -> BoolV(b1 || b2)
                    | _ , _ -> failwith (Format.asprintf "Invalid logical-or: %a || %a" Ast.pp_e e1 Ast.pp_e e2)
            end
```

[interpreter.ml - interp_e_후반부]

● 이번 과제에서의 Add,Sub,Lt,Gt,Eq,And,Or 패턴은 지난주 실습에서의 패턴과 같은 형태를 가지며 바뀐점은 interp_e의 인자가 Store에서 (Env,Mem)으로 구성된 튜플로 바뀌었다는 점이다.

● 대표적으로 Add 패턴의 Bigstep-Operation을 보면

[Rule Add] : $e_1 + e_2$
Env σ와 Mem M에서 e_1이 n_1으로 계산되고,
Env σ와 Mem M에서 e_2가 n_2으로 계산된다고 가정하면,
Env σ와 Mem M에서 e_1+e_2는 $n_1+_Z n_2$로 계산 ($+_Z$는 정수 더하기)

[Add] : $e_1 + e_2$

$$\frac{\sigma,\ M \vdash e_1 \Downarrow_E n_1 \quad \sigma,\ M \vdash e_2 \Downarrow_E n_2}{\sigma,\ M \vdash e_1 + e_2 \Downarrow_E n_1 +_Z n_2}$$

으로 크게 바꿔줄 내용은 없었으며 다른 Sub,Lt,Gt,Eq,And,Or 패턴들도 기존의 코드에서 interp_e의 추상메모리 인자부분을 (Env,Mem) 튜플로 바꿔주기만 하면 되었기에 크게 어려운 점은 없었다.

<실행 결과 화면>

[실행결과 전체 이미지 및 부분 확대 이미지]

- 전체 실행결과와 Env 및 Mem 결과 부분만 확대를 한 이미지이다.
- 실습자료에 제시된 결과와 같은 올바른 결과가 출력 되었음을 확인할 수 있었다.

<새로 고찰한 내용 또는 느낀점>

● 이번 마지막 과제를 진행하고 보고서를 쓰며 Ocaml을 처음 배울 때 생각이 났다. 처음 접해보는 언어였기에 생소하고 겁도 났지만 강의를 수강하고 실습을 진행하며 어느덧 기본적인 문법에 대해서는 조금 자신이 생기게 된 것 같다. 마지막 과제를 마치면서 뿌듯함과 열심히 하면 할 수 있다는 자신감을 얻은 것 같다!

한 학기 동안 정말 감사드렸습니다!

김성한 올림.