프로그래밍 언어 개론
과제 보고서

[PL00]HW13_201601980_KimSeongHan

개발환경 : Ubuntu

제출일 (2021-05-30)

201601980 / 김성한(00분반)

<과제 설명>

Homework

```
let rec interp_e (e : Ast.expr) (s : Store.t) : Store.value =
(* write your code *)

let rec interp_s (stmt : Ast.stmt) (s : Store.t) : Store.t =
(* write your code *)

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

Testcase : https://github.com/SeoyeonKang/UPL2021/tree/master/week13

[과제 제시 내용]

● 이번 과제는 기존의 우리가 구현하던 Functional Language에 대한 interpreter가 아닌 Imperative Language인 C언어를 정의하는 것이다.

● C언어 전체 개념에 대한 정의가 아닌 C의 부분집합을 정의하며 해당 언어는 MiniC로 제시를 해주셨다. 이번 과제에서 구현할 부분은 '변수 정의 및 할당' 그리고 '분기문' 에 대한 정의를 해주는 것이다.

● 해당 MiniC의 특징으로는 기존 C언어와 달리 별도의 타입이 존재하지 않아 타입 에러에 취약하며 컴파일이 아닌 interpreter로 MiniC언어 프로그램을 실행한다.

● 우리는 그러한 interpreter의 세함수 interp_e, interp_s, interp를 정의하게 되며 각각은 세가지 Sementic Relation을 정의하게 된다. 다시말해 이번 과제는 세가지 Semantic Relation이 각각의 기능을 하도록 구현하는 과제인 것이다.

● 구현의 시작은 interp함수부터 시작하였다.

<코드 구현 - 해결 방법>

```
type expr =
    | Num of int
    | Var of string
    | Bool of bool
    | Add of expr * expr
    | Sub of expr * expr
    | Lt of expr * expr
    | Gt of expr * expr
    | Eq of expr * expr
    | And of expr * expr
    | Or of expr * expr

type stmt =
    | AssignStmt of string * expr
    | IfStmt of expr * stmt list * (stmt list) option

type program = Program of stmt list
```

[ast.ml 내부 Disjoint_Union]

● 이번 실습에서의 ast.ml내부 disjoint_union부분이다.

● 크게 세가지의 타입이 정의되어 있으며 각각은 expr, stmt, program 임을 확인할 수 있었다.

● 해당 타입들은 interpreter.ml 내부 함수들에 의해 패턴매칭되며 각각의 패턴에 대한 Bigstep-Operation을 참고하여 구현하였다.

```
module F = Format

type value =
  | NumV of int
  | BoolV of bool
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 pp_v fmt v =
  match v with
  | NumV i -> F.fprintf fmt "%d" i
  | BoolV b -> F.fprintf fmt "%b" b

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

[store.ml 내부 코드]

● 이번 실습에서 사용된 store.ml 내부 코드이다.

● 이전 Funtional Language의 store.ml코드와 크게 바뀐 점은 없다.

● 다만 이번 실습에서 구현하는 MiniC, 즉 C의 부분집합에 대한 정의인 변수 정의 및 할당 분기문에서 사용될 값들(value)에 대한 정의가 조금 바뀌었을 뿐 나머지 코드는 그대로이다.

● 이번 실습에서의 value는 정수형 NumV와 boolean타입 BoolV가 선언되어 있는 모습을 확인할 수 있었다.

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

[interpreter.ml 내부 interp 함수]

● interp 함수의 구성모습이다.

● interp 함수는 인자로 p를 가지고 있으며 p는 ast.ml의 program 타입으로 명시되어 있고, 반환형은 Store.t타입으로 program실행 이후의 추상메모리를 반환함을 알 수 있다.

● Program패턴의 Bigstep-Operation은

으로 표현되며 초기 빈 추상메모리를 시작으로 각각의 s를 실행한 결과인 추상메모리를 다음 s에 대한 Semantic에서 사용하는 모습을 확인할 수 있었다.

● 이에 따라 재귀문의 필요성을 느꼈고, 인자로 들어온 p에 대하여 Program(li)를 통해 해당 s에 대한 리스트 li를 하나씩 분해하여 연산한 추상메모리를 이용하여 다음 s를 연산하는 재귀함수를 작성하였다. 이 함수가 interp_impl이며 li와 acc를 인자로 가진다.

● acc 이름으로 추상메모리를 누적하는 방법을 생각했어지만 생각해보니 누적할 필요없이 해당 추상메모리를 이용한 연산만 진행하면 되었기에 따로 누적하는 코드는 작성하지 않았다.

● 마지막에 li가 [](빈 리스트)가 되면 지금까지 연산의 결과로써 반환된 acc를 반환하고 함수를 종료한다. 또한 래퍼함수내에서의 실행을 위해 초기 조건대로 interp_impl li []를 선언해주었다.

```
(* practice & homework *)
let rec interp_s (stmt : Ast.stmt) (s : Store.t) : Store.t =
        match stmt with
        | AssignStmt(str,e) ->
                        begin
                                let v = interp_e e s in
                                (Store.insert str v s)
                        end
```

[interpreter.ml 내부 interp_s 함수 초반부]

● interp_s는 statement를 통해 현재의 추상메모리에서 statement 실행 시 변화된 추상메모리를 반환한다.

● interp_s는 인자로 stmt(=statement) s(추상메모리)를 가지며 반환형 타입은 Store.t이다.

● stmt에 대해 패턴매칭을 진행하게되며 첫 번째로는 AssignStmt패턴으로

[Rule Assign] : x = e
추상메모리σ에서 **e**을 계산한 결과가 **v**이라고 가정하면,
추상메모리σ에서 **x = e**를 실행하면 **σ[x↦v]**를 반환

[Assign] : x = e

$$\frac{\sigma \vdash e \Downarrow_E v}{\sigma \vdash x = e \Rightarrow_S \sigma[x \mapsto v]}$$

의 Bigstep-Operation을 가진다.

● AssignStmt(str,e)에 대하여 e에 대해 interp_e e s를 한결과가 v이면 해당 결과를 str변수에 저장한 추상메모리를 반환하는 동작을 수행한다.

● Bigstep-Operation에서 명시된 과정과 똑같이 구현할 수 있었다.

```
| IfStmt(e,true_stmts,None) ->
                begin
                        let v = interp_e e s in
                        let rec cal_list li acc =
                                match li with
                                |h::t -> cal_list t (interp_s h acc)
                                |[] -> acc
                        in
                        match v with
                        |BoolV v -> if v = true then (cal_list true_stmts s) else s
                        | _ ->  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 s in
                        let rec cal_list li acc =
                                match li with
                                |h::t -> cal_list t (interp_s h acc)
                                |[] -> acc
                        in
                        match v with
                        |BoolV v -> if v = true then (cal_list true_stmts s) else (cal_list false_stmts s)
                        | _ ->  failwith (Format.asprintf "Not a boolean : %a" Ast.pp_e e)
                end
```

[interpreter.ml 내부 interp_s 함수 후반부]

● AssginStmt이 후 두가지 패턴이 더 있다. IfStmt패턴에서 false_stmts가 None인 경우와 Some false_stmts 인 경우로 두가지 패턴으로 나누어 각각의 경우에 대한 동작을 정의하였다. 크게 보면 false_stmts가 없는 경우 false인 경우에는 그 자체의 추상메모리 s를 반환하며 false_stmts가 있는경우에는 해당 stmts에 대해 cal_list를 진행한 결과를 반환한다.

● 각각의 경우에대한 Bigstep-Operation은

[Rule If-True] : (e)? s_{t1} … s_{tn} : $\overline{s_{opt}}$
추상메모리σ에서 e을 계산한 결과가 true 이고,
추상메모리σ에서 s_{t1} 를 실행한 결과가 σ_1이고, …
추상메모리σ_{n-1}에서 s_{tn} 를 실행한 결과가 σ_n이라고 가정하면,
추상메모리σ에서 (e)? s_{t1} … s_{tn} : $\overline{s_{opt}}$를 실행하면 σ_n를 반환

[If-True] : (e)? s_{t1} … s_{tn} : $\overline{s_{opt}}$

$$\frac{\sigma \vdash e \Downarrow_E \text{true} \quad \sigma \vdash s_{t1} \Rightarrow_S \sigma_1 \quad \dots \quad \sigma_{n-1} \vdash s_{tn} \Rightarrow_S \sigma_n}{\sigma \vdash (e)?\ s_{t1} \dots s_{tn} : \overline{s_{opt}} \Rightarrow_S \sigma_n}$$

[Rule If-False1] : (e)? $\overline{s}$: ε
추상메모리σ에서 e을 계산한 결과가 false 이면,
추상메모리σ에서 (e)? $\overline{s}$: ε 를 실행하면 σ를 반환

[If-False1] : (e)? $\overline{s}$: ε

$$\frac{\sigma \vdash e \Downarrow_E \text{false}}{\sigma \vdash (e)?\ s : \varepsilon \Rightarrow_S \sigma}$$

[Rule If-False2] : (e)? $\overline{s}$: s_{f1} … s_{fn}
추상메모리σ에서 e을 계산한 결과가 false 이고,
추상메모리σ에서 s_{f1} 를 실행한 결과가 σ_1이고, …
추상메모리σ_{n-1}에서 s_{fn} 를 실행한 결과가 σ_n이라고 가정하면,
추상메모리σ에서 (e)? $\overline{s}$: s_{f1} … s_{fn}를 실행하면 σ_n를 반환

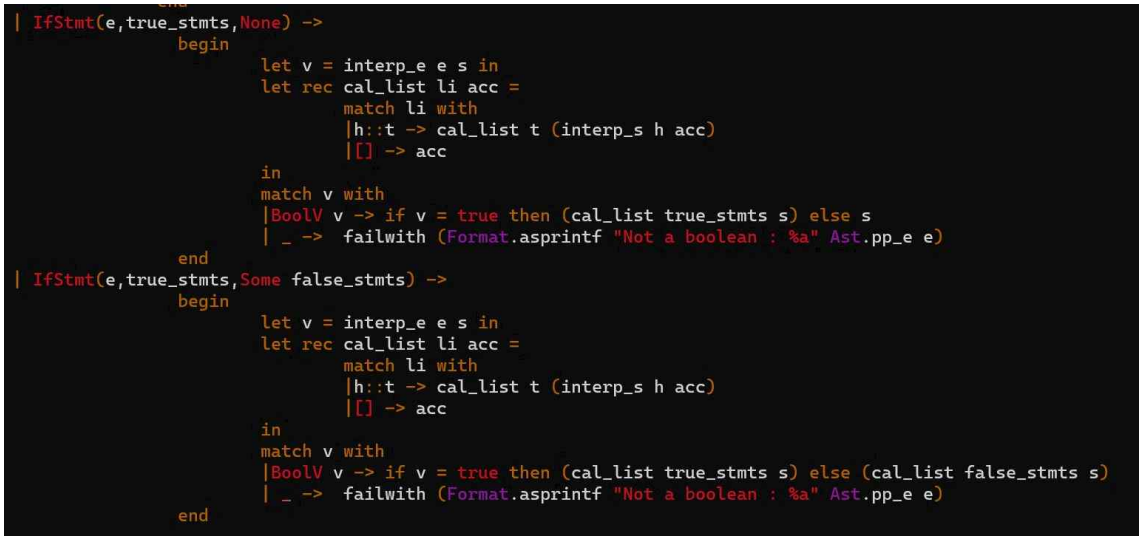
[If-False2] : (e)? $\overline{s}$: s_{f1} … s_{fn}

$$\frac{\sigma \vdash e \Downarrow_E \text{false} \quad \sigma \vdash s_{f1} \Rightarrow_S \sigma_1 \quad \dots \quad \sigma_{n-1} \vdash s_{fn} \Rightarrow_S \sigma_n}{\sigma \vdash (e)?\ \overline{s} : s_{f1} \dots s_{fn} \Rightarrow_S \sigma_n}$$

으로 나타낼 수 있으며 명시된 규칙 및 과정대로 동작하게 된다.

● 이부분에서 가장 고민했던 부분은 어떻게 stmts(stmt list)를 쪼개어 각각의 stmt를 연산하여 추상메모리를 반환해줄 수 있을까였다.

(*너무 길어 두장에 나누어 설명하겠습니다!*)



```
| IfStmt(e,true_stmts,None) ->
                begin
                        let v = interp_e e s in
                        let rec cal_list li acc =
                                match li with
                                |h::t -> cal_list t (interp_s h acc)
                                |[] -> acc
                        in
                        match v with
                        |BoolV v -> if v = true then (cal_list true_stmts s) else s
                        | _ ->  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 s in
                        let rec cal_list li acc =
                                match li with
                                |h::t -> cal_list t (interp_s h acc)
                                |[] -> acc
                        in
                        match v with
                        |BoolV v -> if v = true then (cal_list true_stmts s) else (cal_list false_stmts s)
                        | _ ->  failwith (Format.asprintf "Not a boolean : %a" Ast.pp_e e)
                end
```

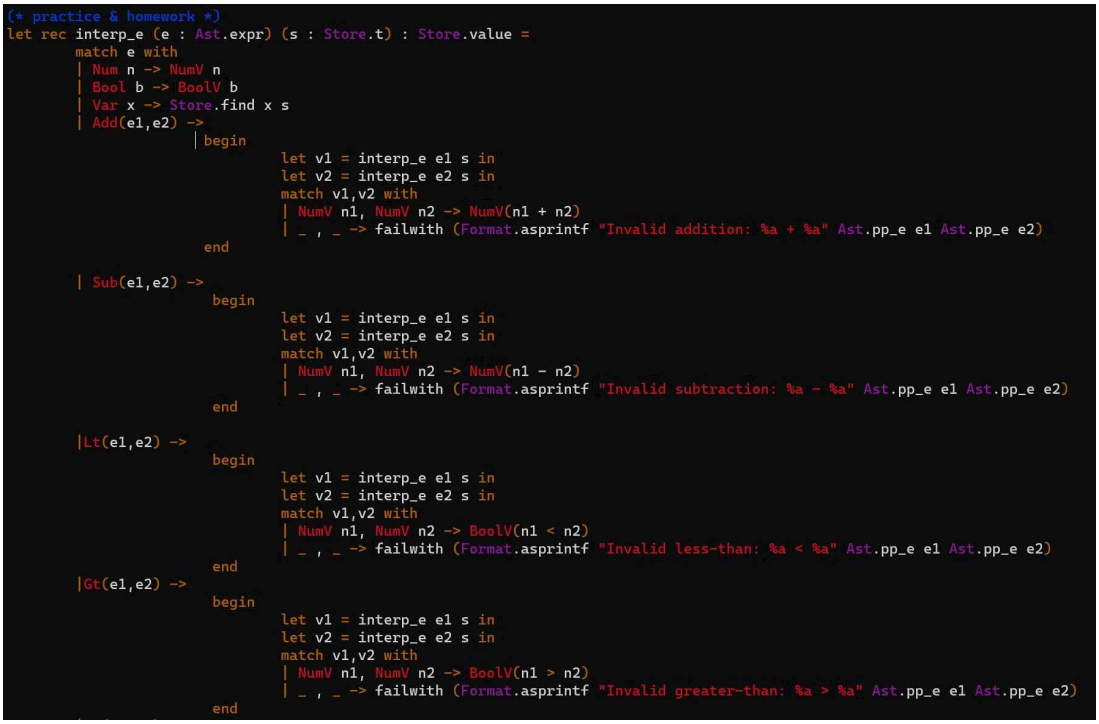
[interpreter.ml 내부 interp_s 함수 후반부]

● 거의 같은 기능이므로 마지막 패턴인 false_stmts가 있을 수 있는 경우에 대해 진행해보면 패턴은 IfStmt(e,true_stmts,Some false_stmts) 인 경우 e에 대한 interp_e 연산을 진행한 결과를 변수 v에 저장하였다. 그 후 재귀함수를 하나 선언하였으며, 함수명은 cal_list이고 인자로는 statement의 리스트인 li와 마지막 결과가 저장된 acc이다. li에 대해 패턴매칭을 진행하며 li에서 분리한 각각의 원소에 대하여 interp_s연산을 진행하여 반환된 추상메모리를 이용해 (cal_list t (h에 대한 연산으로 반환된 추상메모리)) 의 재귀를 진행하게 된다. 이렇게 하면 초기부터 연산한 h에 대해 마지막 결과를 연산할 때 마지막 이전 stmt에 대해 연산된 추상메모리를 이용해 연산에 이용할 수 있다. interp함수에서 재귀함수를 선언했을때와 비슷한 형태의 재귀문을 여기에도 써준 것이다. 그후 li가 [](빈 리스트)가 되는 경우에 최종 stmt의 추상메모리인 acc를 반환해준다.

● 이렇게 재귀 함수를 선언한 뒤에 초기 연산해 놓았던 v를 이용해 v가 BoolV타입인 경우에 대하여 v가 true이면 true_stmts에 대해 cal_list재귀를 진행하고 v가 false인 경우에는 false_stmts에 대해 cal_list재귀를 진행해 주었다. 이렇게 하면 v가 true와 false인 경우에 대하여 Bigstep-Operation과 같게 동작할 수 있게 된다.

● 혹시나 v가 BoolV타입인 아닌 경우에 대해서는 failwith을 이용해 예외처리를 해주었다.

● 이와 같이 해당 패턴을 구현할 수 있었으며 기존 추상메모리를 반환하냐 아니면 false_stmts의 실행결과 추상메모리를 반환하느냐의 차이에서 None과 Some에 대한 패턴의 구별이 생기기에 해당 부분을 제외하고는 코드 및 동작이 거의 같다.



```
(* practice & homework *)
let rec interp_e (e : Ast.expr) (s : Store.t) : Store.value =
        match e with
        | Num n -> NumV n
        | Bool b -> BoolV b
        | Var x -> Store.find x s
        | Add(e1,e2) ->
                        | begin
                                let v1 = interp_e e1 s in
                                let v2 = interp_e e2 s 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 s in
                                let v2 = interp_e e2 s 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 s in
                                let v2 = interp_e e2 s 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 s in
                                let v2 = interp_e e2 s 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
```

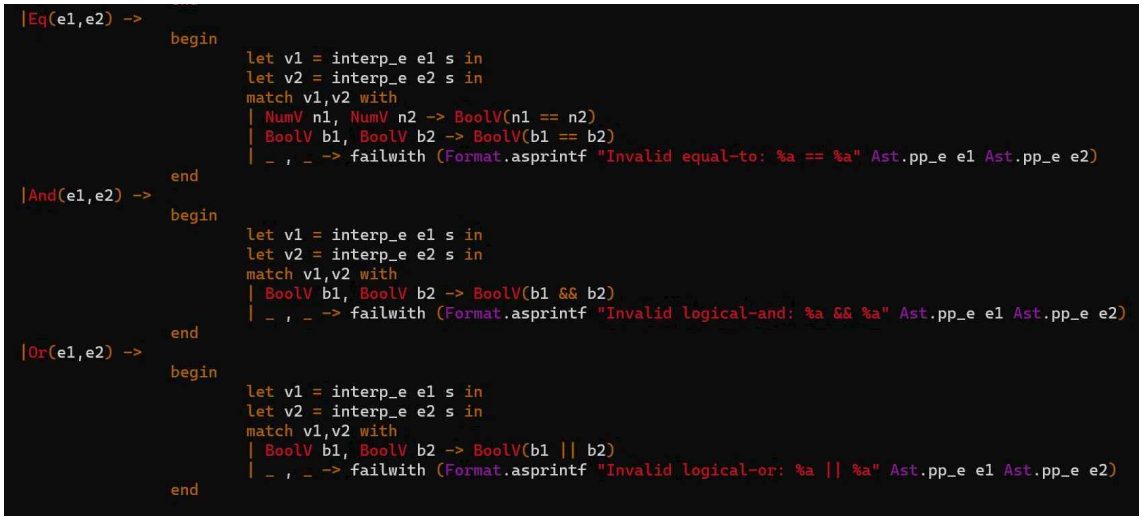
[interpreter.ml 내부 interp_e 함수 초반부]

● interp_e 함수는 expr의 연산을 담당하는 함수이다. rec키워드를 이용해 재귀함수임을 명시하였으며, 인자로는 e (Ast.expr = expression) , s(추상메모리)를 가진다. expression인 e에 대하여 패턴매칭을 진행하게 된다. 반환형은 Store.value로 NumV와 BoolV를 가진다.

● Num과 Bool 타입은 그대로의 NumV와 BoolV 타입을 반환하는 패턴으로 해당 패턴의 동작에 맞게 구현하였다.

● Add와 Sub는 덧셈과 뺄셈을 진행하는 패턴으로 각각의 expression인 e1,e2에 대해 interp_e를 진행하여 나온 값 v1,v2에 대해 각각의 v1,v2가 모두 NumV타입일 때 덧셈 그리고 뺄셈연산을 진행한다. 만일 v1,v2가 NumV타입이 아니라면 failwith을 이용해 예외처리를 해주었다.

● Lt와 Gt는 대소 비교연산을 진행하여 BoolV타입을 반환하는 연산이다. expression인 e1,e2에 대하여 interp_e를 진행해 나온 값 v1,v2가 둘다 NumV타입인 경우 해당 n1,n2에 대하여 대소비교 연산을 한 BoolV타입의 값을 반환한다. 이때에도 v1과 v2가 NumV가 아닌 경우 예외처리를 해주었다.



```
|Eq(e1,e2) ->
                begin
                        let v1 = interp_e e1 s in
                        let v2 = interp_e e2 s 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 s in
                        let v2 = interp_e e2 s 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 s in
                        let v2 = interp_e e2 s 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 내부 interp_e 함수 후반부]

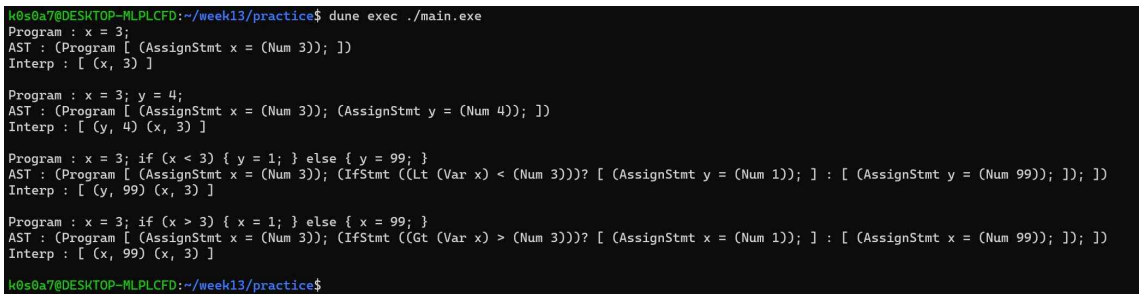
● 이후 Eq타입은 e1,e2에 대해 interp_e를 진행한 값 v1,v2에 대하여 == 동등 연산을 진행한 결과를 반환한다. 이때 각각의 비교는 NumV는 NumV끼리, BoolV는 BoolV끼리 비교해야 하므로 각각의 패턴을 나눠주었고 반환 값은 비교를 진행한 BoolV(a==b) 형태로 해당 결과를 반환하였다. 만일 두 타입이 일치하지 않는 경우는 예외처리를 해주었다.

● And 연산은 e1,e2에 대해 interp_e를 진행한 값 v1,v2에 대하여 두 값이 모두 BoolV 타입인 경우 두 값에 대해 && 연산을 진행한 BoolV타입의 결과를 반환한다. 만일 두 값 v1.v2가 BoolV타입이 아닌 경우는 && 논리연산을 진행할 수 없으므로, 예외처리를 해주었다.

● Or 연산은 e1,e2에 대해 interp_e를 진행한 값 v1,v2에 대하여 두 값이 모두 BoolV 타입인 경우 두 값에 대해 || 연산을 진행한 BoolV타입의 결과를 반환한다. 만일 두 값 v1.v2가 BoolV타입이 아닌 경우는 || 논리연산을 진행할 수 없으므로, 예외처리를 해주었다.

● 이와 같이 interp_e 함수를 구현할 수 있었으며 각각의 패턴에 대한 동작을 정의해줌으로써 Store.value타입의 결과를 반환할 수 있었다.

<실행 결과 화면>



[실행결과]

```
Program : x = 3;
AST : (Program [ (AssignStmt x = (Num 3)); ])
Interp : [ (x, 3) ]

Program : x = 3; y = 4;
AST : (Program [ (AssignStmt x = (Num 3)); (AssignStmt y = (Num 4)); ])
Interp : [ (y, 4) (x, 3) ]

Program : x = 3; if (x < 3) { y = 1; } else { y = 99; }
AST : (Program [ (AssignStmt x = (Num 3)); (IfStmt ((Lt (Var x) < (Num 3)))? [
(AssignStmt y = (Num 1)); ] : [ (AssignStmt y = (Num 99)); ]); ])
Interp : [ (y, 99) (x, 3) ]

Program : x = 3; if (x > 3) { x = 1; } else { x = 99; }
AST : (Program [ (AssignStmt x = (Num 3)); (IfStmt ((Gt (Var x) > (Num 3)))? [
(AssignStmt x = (Num 1)); ] : [ (AssignStmt x = (Num 99)); ]); ])
Interp : [ (x, 99) (x, 3) ]
```

[교수님께서 제공해주신 TestCase 결과 정답]

● dune exec ./main.exe를 통해 구현한 코드를 실행한 결과 위의 사진과 같이 결과가 나오게 되었고, 교수님께서 제공해주신 정답과 비교하였을 때 옳은 결과가 나왔음을 확인할 수 있었다.

<새로 고찰한 내용 또는 느낀점>

● 이번 과제 이전 과제까지는 함수형 언어에 대한 interpreter를 정의해왔었다. 이번 시간부터 Imperative Language 에 대한 interpreter를 정의하였다. 초기 실습과제에 대한 구현을 시작하기 전 statement에 대한 이해가 조금 안되는 부분이 있었기에 이론 내용을 이해하는 방향으로 초점을 맞추었다. 이론강의를 여러번 돌려보고 이해하고 과제를 다시 진행할 수 있었다. 지난 시간에도 이론에 대한 이해를 통해 실습을 진행하였는데, 이번에도 이론이 역시나 굉장히 중요하다고 생각하면서 과제를 진행한 것 같다. 특히 이번 과제 중 interp와 interp_s 코드를 구현할 때 statement list에서 각각의 statement를 분리하여 해당 statement에 대한 연산을 진행한 결과인 추상메모리를 다음 statement에 대한 연산에 사용하는 부분에 대한 코드가 쉽게 떠오르지 않아 고민을 하였는데 이론에 대한 이해를 통해 코드를 구현할 수 있었다. 이제 얼마 남지 않았지만 이론과 실습 모두 마지막까지 최선을 다해 마무리하고 싶다는 생각이 강하게 들었다.