

프로그래밍 언어 개론

과제 보고서

[PL00]HW6_201601980_KimSeongHan

개발환경 : Ubuntu

제출일 (2021-04-09)

201601980 / 김성한(00분반)

<과제 설명>

Homework1



주어진 semantic rule을 이용하여 interpreter를 작성하시오.

```
let rec interp (e : Ast.ae) : int =
  (* write your code *)
```

$e ::= n \mid e + e \mid e - e \mid -e$
 $n \in \mathbb{Z}$

[Rule Neg]: $-e$
 e 를 계산한 결과가 n 이라고 가정하면,
 $-e$ 는 $(-1) * n$ 으로 계산 ($*$ 는 수학에서의 정수 곱셈 연산)

```
(* ast.ml *)
type ae =
  | Num of int
  | Add of ae * ae
  | Sub of ae * ae
  | ?
```

보고서에 포함

[Num]: n
 $n \Downarrow n$

[Add]: $e_1 + e_2$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 +_Z n_2}$$

[Sub]: $e_1 - e_2$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 - e_2 \Downarrow n_1 -_Z n_2}$$

[Neg]: $-e$

$$\frac{?}{?}$$

보고서에 포함

[과제 HomeWork1]

Homework2

$4 + (- (3 - 1))$ 가 2 로 계산됨을 증명하시오.

(proof tree를 그려서 보고서에 작성)

proof tree?

Inference rule을 이용하여 결론을 증명하는 과정을 나타내는 자료구조

$4 + (- (3 - 1)) \Downarrow 2$

[과제 HomeWork2]

● 이번 과제는 제시된 semantic rule을 이용하여 interpreter.ml의 interp 함수를 완성하는 것이다. type 은

Num -> n (int)

Add -> expr + expr

Sub -> expr - expr

Neg -> - expr

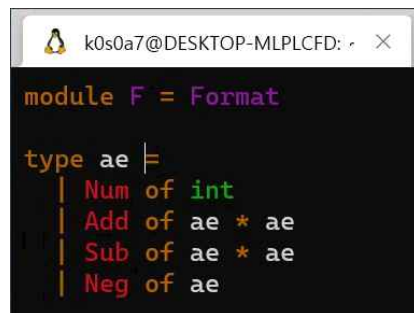
로 정리할 수 있으며 해당 조건에 맞추어 pattern matching을 통해 구현하면 되겠다고 생각하며 과제를 시작하였다.

<보고서 포함 명시 내용>

$$\frac{e_1 \Downarrow n_1}{-e_1 \Downarrow -n_1}$$

[Neg 의 Sementic Rule]

- Neg 는 하나의 expr을 받아 음의 부호를 붙여준다. 따라서 위와 같이 Sementic Rule을 정의할 수 있다.



```
module F = Format

type ae =
  | Num of int
  | Add of ae * ae
  | Sub of ae * ae
  | Neg of ae
```

[ast.ml 의 Disjoint Union 부분]

- Neg의 타입은 하나의 ae 로 정의가 될 수 있다. 구현내용은 뒤에 코드설명에 되어있다.

<HomeWork2 Draw Proof Tree>

$$4 + (-(3 - 1)) \Downarrow 2$$

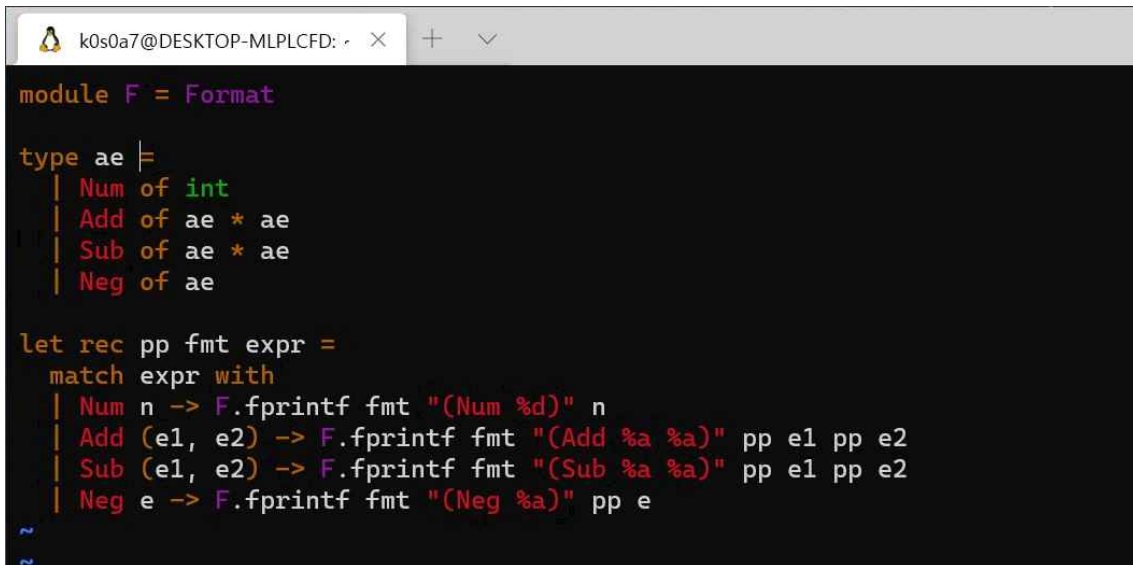
[Proof Tree를 그릴 식]

$$\frac{\frac{3 \Downarrow 3 \quad 1 \Downarrow 1 \quad 3 -_z 1 = 2}{-_z (3 - 1) \Downarrow -2} \quad 4 \Downarrow 4 \quad 4 +_z -2 = 2}{4 + (-(3 - 1)) \Downarrow 2}$$

[위의 식에 대한 Proof Tree]

- 위에 제시된 식 $4 + (-(3-1)) \Downarrow 2$ 에 대하여 Proof Tree를 그려보았다.
- 아래의 그림처럼 Proof Tree가 생성이 되며 해당 tree를 그리고 보니 함수를 실행했을 때의 재귀문을 표현한 느낌을 받았다.
- ast.ml에 정의된 disjoint union에서의 연산에는 기호아래에 z 를 붙여 표현하였다.

<코드 구현 및 실행 결과 화면>



```
module F = Format

type ae =
| Num of int
| Add of ae * ae
| Sub of ae * ae
| Neg of ae

let rec pp fmt expr =
  match expr with
  | Num n -> F.fprintf fmt "(Num %d)" n
  | Add (e1, e2) -> F.fprintf fmt "(Add %a %a)" pp e1 pp e2
  | Sub (e1, e2) -> F.fprintf fmt "(Sub %a %a)" pp e1 pp e2
  | Neg e -> F.fprintf fmt "(Neg %a)" pp e
```

[ast.ml의 내부코드 모습]

- ast.ml 파일은 초기 Format 모듈을 F로 정의하며 시작된다.
- ae type에 대한 정의가 나와있으며

Num = int형 정수

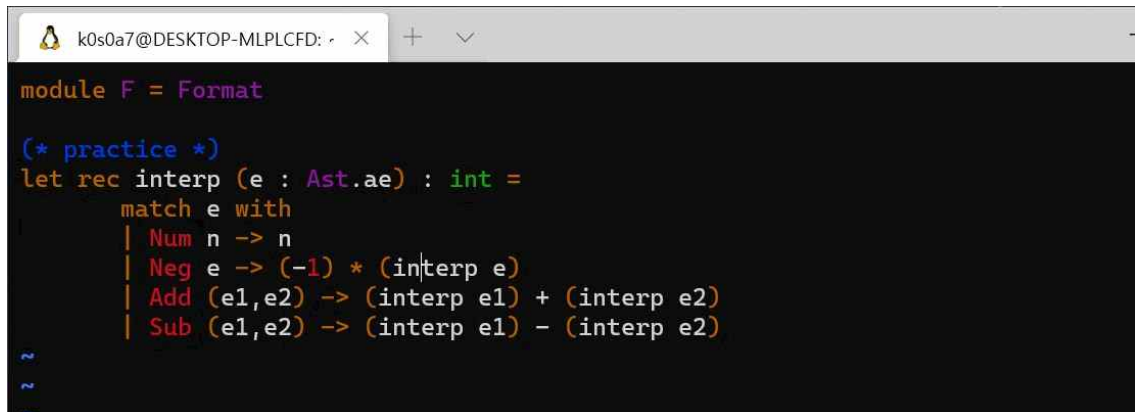
Add = ae 타입의 합 연산

Sub = ae 타입의 차 연산

Neg = ae 타입의 부호 변환

으로 정의되어 있다.

- 그리고 그 밑에는 해당 타입들에 대한 출력을 위해 pp 함수가 재귀적으로 선언되어 있다.



```
module F = Format

(* practice *)
let rec interp (e : Ast.ae) : int =
  match e with
  | Num n -> n
  | Neg e -> (-1) * (interp e)
  | Add (e1,e2) -> (interp e1) + (interp e2)
  | Sub (e1,e2) -> (interp e1) - (interp e2)
```

[interpreter.ml의 내부코드 모습]

- interpreter.ml의 구현은 생각보다 단순하였다. 초기 interp 함수는 재귀 rec으로 선언되어 있었으며 이때 인자로 들어오는 e의 타입은 Ast.ml 의 ae 타입으로 정의되어 있다. 또한 반환값은 int로 선언되어 있는 모습을 확인할 수 있었다.

- 인자로 들어온 e에 대하여 pattern matching을 진행해 주었다.

1. e 가 Num n 인 경우 정수 n을 반환하도록 하였다.

2. e 가 Neg e 인 경우 해당 e에 대한 interp 재귀의 결과에 -1을 곱해줌으로써 부호를 변환해주었다.

3. e 가 Add (e_1, e_2) 인 경우 각각의 e 에 대한 interp 재귀의 결과를 더해주었다.

3. e 가 Sub (e_1, e_2) 인 경우 각각의 e 에 대한 interp 재귀의 결과를 뺄셈을 진행해 주었다.

- 이렇게 interpreter.ml의 interp 함수를 완성할 수 있었다.

```

(* Test cases *)
let _ =
  let a = Ast.Num 1 in (* a = Num 1 *)
  let b = Ast.Num 3 in (* b = Num 3 *)
  let c = Ast.Num 4 in (* c = Num 4 *)
  let d = Ast.Add (a, b) in (* d = Add (Num 1, Num 3) *)
  let e = Ast.Sub (c, d) in (* e = Sub (Add (Num 1, Num 3), Num 4) *)
  let f = Ast.Neg (Ast.Num 7) in (* f = Neg (Num 7) *)
  let g = Ast.Sub (e, f) in (* g = Sub (Sub (Add (Num 1, Num 3), Num 4), Neg (Num 7)) *)
  let h = Ast.Add (Ast.Num 4, Neg (Sub (Ast.Num 3, Ast.Num 1))) in
  let _ = F.printf "a = %a\n" Ast.pp a in
  let _ = F.printf "interp a = %d\n" (Interpreter.interp a) in (* 1 *)
  let _ = F.printf "b = %a\n" Ast.pp b in
  let _ = F.printf "interp b = %d\n" (Interpreter.interp b) in (* 3 *)
  let _ = F.printf "c = %a\n" Ast.pp c in
  let _ = F.printf "interp c = %d\n" (Interpreter.interp c) in (* 4 *)
  let _ = F.printf "d = %a\n" Ast.pp d in
  let _ = F.printf "interp d = %d\n" (Interpreter.interp d) in (* 4 *)
  let _ = F.printf "e = %a\n" Ast.pp e in
  let _ = F.printf "interp e = %d\n" (Interpreter.interp e) in (* 0 *)
  let _ = F.printf "f = %a\n" Ast.pp f in
  let _ = F.printf "interp f = %d\n" (Interpreter.interp f) in (* -7 *)
  let _ = F.printf "g = %a\n" Ast.pp g in
  let _ = F.printf "interp g = %d\n" (Interpreter.interp g) in (* 7 *)
  let _ = F.printf "h = %a\n" Ast.pp h in
  let _ = F.printf "interp h = %d\n" (Interpreter.interp h) in (* 2 *)
  ()

```

[main.ml의 내부코드 모습]

```

k0s0a7@DESKTOP-MLPLCFD:~/week6/hw$ dune exec ./main.exe
a = (Num 1)
interp a = 1
b = (Num 3)
interp b = 3
c = (Num 4)
interp c = 4
d = (Add (Num 1) (Num 3))
interp d = 4
e = (Sub (Num 4) (Add (Num 1) (Num 3)))
interp e = 0
f = (Neg (Num 7))
interp f = -7
g = (Sub (Sub (Num 4) (Add (Num 1) (Num 3))) (Neg (Num 7)))
interp g = 7
h = (Add (Num 4) (Neg (Sub (Num 3) (Num 1))))
interp h = 2
k0s0a7@DESKTOP-MLPLCFD:~/week6/hw$

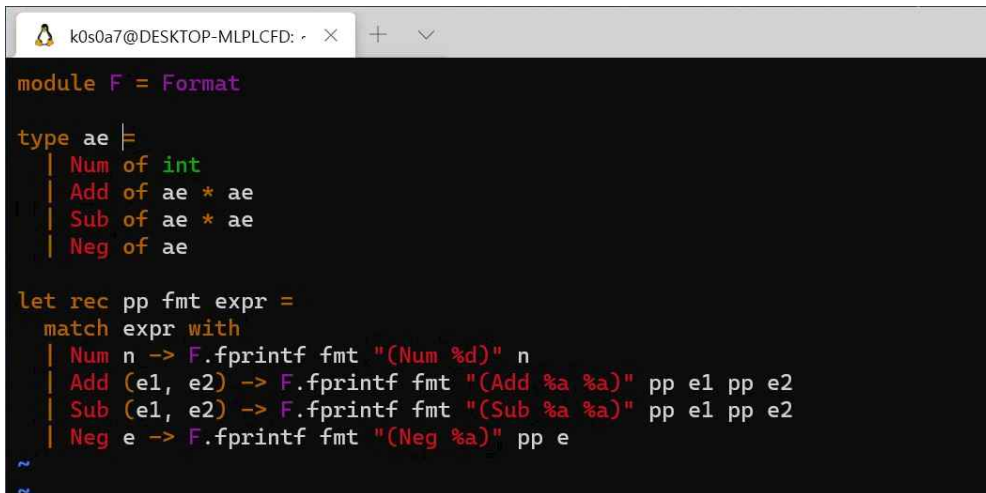
```

[main.exe의 실행 결과 출력 화면]

● 위의 사진은 main.ml 과 main.exe를 실행했을 때의 결과 사진이다. 이때 h 로 HomeWork2에서 제시한 식에 대해서도 결과 출력을 진행해 보았으며 입력된 인자들에 대하여 이상없이 결과가 출력된 모습을 확인할 수 있었다.

● 이로써 간단한 interpreter를 구현하였다.

<과제 구현 중 어려웠던 부분 및 해결 과정>



```
module F = Format

type ae =
  | Num of int
  | Add of ae * ae
  | Sub of ae * ae
  | Neg of ae

let rec pp fmt expr =
  match expr with
  | Num n -> F.fprintf fmt "(Num %d)" n
  | Add (e1, e2) -> F.fprintf fmt "(Add %a %a)" pp e1 pp e2
  | Sub (e1, e2) -> F.fprintf fmt "(Sub %a %a)" pp e1 pp e2
  | Neg e -> F.fprintf fmt "(Neg %a)" pp e
```

[ast.ml의 내부코드 모습]

- 이번 과제는 다른때에 과제보다 코드의 길이도 길지 않아서 생각보다 진행하는데 큰 어려움은 없었다.
- 하지만 고민을 가장 많이 했던부분은 Neg에 대한 타입의 정의였다. 초기 Neg의 타입을

Neg of (-1) * ae

로 정의하였었다. 역시 실행시 오류가 나게 되었고, 고민을 조금 하였다.

- 고민을 하던 중 밑을 확인해보니 pp 함수의 Neg 타입이 e 일때로 pattern matching이 진행되어 있었다. 물론 이것만 보고 한번에 이해를 하지는 못했다. 그래서 고민을 해본 끝에 해당 expr은 일단 튜플의 형태가 아니므로 ae로 정의를 하고 interpreter.ml에서 패턴매칭시 연산을 정의해 주면 되겠다고 생각하였다.



```
| Neg e -> (-1) * (interp e)
```

- 위의 사진처럼 Neg e를 처리해주었고 이렇게 interpreter.ml 또한 완성할 수 있었다.

<최종 제출 코드 및 결과 화면>

```
k0s0a7@DESKTOP-MLPLCFD: ~$  
module F = Format  
  
(* practice *)  
let rec interp (e : Ast.ae) : int =  
  match e with  
  | Num n -> n  
  | Neg e -> (-1) * (interp e)  
  | Add (e1,e2) -> (interp e1) + (interp e2)  
  | Sub (e1,e2) -> (interp e1) - (interp e2)  
  ~  
  ~  
  ~
```

[최종 제출 interpreter.ml의 내부코드 모습]

```
k0s0a7@DESKTOP-MLPLCFD:~/week6/hw$ dune exec ./main.exe  
a = (Num 1)  
interp a = 1  
b = (Num 3)  
interp b = 3  
c = (Num 4)  
interp c = 4  
d = (Add (Num 1) (Num 3))  
interp d = 4  
e = (Sub (Num 4) (Add (Num 1) (Num 3)))  
interp e = 0  
f = (Neg (Num 7))  
interp f = -7  
g = (Sub (Sub (Num 4) (Add (Num 1) (Num 3))) (Neg (Num 7)))  
interp g = 7  
h = (Add (Num 4) (Neg (Sub (Num 3) (Num 1))))  
interp h = 2  
k0s0a7@DESKTOP-MLPLCFD:~/week6/hw$
```

[main.exe의 실행 결과 출력 화면]

- 최종 제출한 interpreter.ml 파일의 내부코드와 실행 시 결과화면이다.
- main문에 선언된 인자에 대해 올바른 결과를 출력한 모습을 확인할 수 있다.

<새로 고찰한 내용 또는 느낀점>

- 이번 과제는 다른 과제들에 비해 비교적 쉬운느낌을 많이 받았던 것 같다. Disjoin union에 대해 알고 pattern matching을 어느 정도 이해한다면 무리 없이 진행할 수 있었던 것 같다. interpreter의 완성에 있어 초반부이지만 앞으로 이론 및 실습을 열심히 진행해 큰 어려움 없이 interpreter를 완성하고 싶다.