프로그래밍 언어 개론
과제 보고서

[PL00] HW2_201601980_KimSeongHan

개발환경 : Ubuntu

제출일 (2021-03-16)

201601980 / 김성한(00분반)

<과제 설명>

HomeWork1

```
type  unop = Neg
type  binop = Add | Sub |       Mul | Div
type  exp =
                  Constant of int
              | Unary of unop *  exp
          | Binary of exp * binop  * exp
```

주어진 데이터타입 정의를 이용하여, 정수형 결과를 반환하는 연산을 하는 함수 eval을 성하시오. (테스트 케이스 파일 참고)

eval : exp -> int

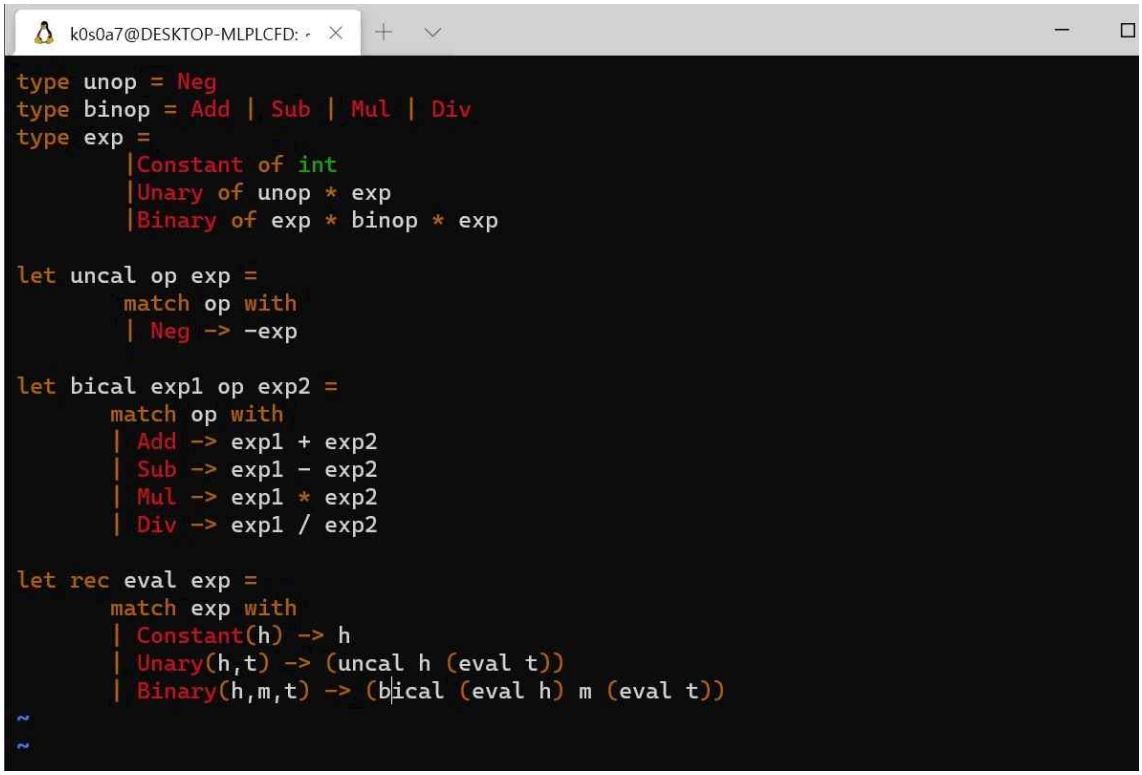
```
(* 테스트 케이스 미리보기 *)
let a = eval (Binary (Constant 1, Add, Constant 3)) in (* a = 4 *)
let b = eval (Binary (Constant 5, Add, (Unary (Neg, (Constant 3))))) in (* b = 2 *)
let c = eval (Unary (Neg, (Binary (Constant 2, Mul, (Unary (Neg, (Constant 7))))))) in (* c = 14 *)
```

[문제 HomeWork1]

● 해당 과제는 문제에 제시된 데이터타입을 정의하고, 해당 데이터타입의 Operation에 맞게 연산한 뒤 integer(정수)형 값을 반환하는 eval함수를 작성하는 과제이다. 연산의 종류로는 크게 5가지가 제시되었으며, Unary인 경우와 Binary인 경우로 나뉜다.

● 처음 문제를 보고 해당 exp값을 처리하는 함수를 eval, 그리고 각각의 unop, binop에 대한 연산은 따로 함수를 정의한뒤 match를 사용해 연산을 해줘야겠다고 구상하면서 과제를 시작하였다.

<코드 구현 및 실행 결과 화면>



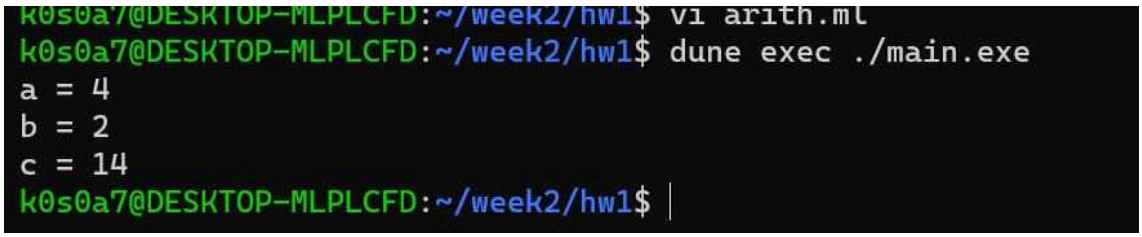
```
type unop = Neg
type binop = Add | Sub | Mul | Div
type exp =
        |Constant of int
        |Unary of unop * exp
        |Binary of exp * binop * exp

let uncal op exp =
        match op with
        | Neg -> -exp

let bical exp1 op exp2 =
       match op with
       | Add -> exp1 + exp2
       | Sub -> exp1 - exp2
       | Mul -> exp1 * exp2
       | Div -> exp1 / exp2

let rec eval exp =
       match exp with
       | Constant(h) -> h
       | Unary(h,t) -> (uncal h (eval t))
       | Binary(h,m,t) -> (bical (eval h) m (eval t))
~
~
```

[arith.ml 파일 내부에 구현된 코드이다.]



```
k0s0a7@DESKTOP-MLPLCFD:~/week2/hw1$ vi arith.ml
k0s0a7@DESKTOP-MLPLCFD:~/week2/hw1$ dune exec ./main.exe
a = 4
b = 2
c = 14
k0s0a7@DESKTOP-MLPLCFD:~/week2/hw1$
```

[main문을 통해 주어진 테스트 케이스를 실행했을때의 결과 화면이다.]

● 초기 문제에 제시된 데이터타입을 arith.ml 내부에 작성한 뒤 eval함수를 작성하였다.

● 먼저 eval함수는 하나의 expression을 인자로 받아와 재귀문을 돌려주도록 선언하였다.(rec) 이때 match를 이용하여 해당 expression의 가장 바깥쪽부터 검사하여 constant인 경우 그냥 int값을 반환하고, Unary인 경우 head, tail로 나누어 (uncal h (eval t))를 반환하도록 하였다. 마지막으로 Binary는 세 개로 이루어진 튜플을 지기에 h,m,t의 인자를 bical 함수를 이용해 처리해 주었다.

다음은 arith.ml 내부의 함수 하나하나를 들여다 보겠다.

```
let uncal op exp =
        match op with
        | Neg -> -exp
```

[unop연산을 위한 uncal함수]

● 먼저 uncal에 대한 설명이다. 사실 처음 작성한 코드에서는 eval 함수 자체에서 Unary(_,t) -> -(eval t)로 작성하여 (뒷부분에 해당 코드에 대한 사진 첨부했습니다.) unop에 대한 연산함수를 따로 작성하지 않았었다. 하지만 코드를 보면서 이렇게 작성하면 나중에 unop 연산을 추가하게 되었을 때 함수자체를 바꿔야하는 일이 생김을 금방 알 수 있었다. 물론 연산을 추가하게 된다면 현재 작성한 uncal을 수정해야 하지만 이때에는 match에 케이스를 늘려주기만 하면 되기 때문에 eval함수에서 다른부분은 건드리지 않아도 된다. 그래서 uncal을 따로 정의하여 Neg연산의 경우에 대해 작성하였다.

```
let bical exp1 op exp2 =
        match op with
        | Add -> exp1 + exp2
        | Sub -> exp1 - exp2
        | Mul -> exp1 * exp2
        | Div -> exp1 / exp2
```

[binop연산을 위한 bical함수]

● 다음은 bical함수이다. 이 함수는 정의된 binop연산을 수행하기 위해 작성된 코드이며, 각각의 binop에 맞는 연산을 수행한다. 이때 op를 매칭함으로써 연산된 결과를 반환한다.

```
let rec eval exp =
        match exp with
        | Constant(h) -> h
        | Unary(h,t) -> (uncal h (eval t))
        | Binary(h,m,t) -> (bical (eval h) m (eval t))
```

[eval함수]

● 마지막으로 eval함수는 exp하나를 인자로 받아와 재귀적으로 매칭을 수행하고 해당 연산에 값을 반환한다. Constant는 하나의 값만을 가지는 데이터 타입으로 해당 값을 반환하도록 선언하였다. Unary는 앞서 선언된 uncal을 이용하여 eval t (나머지 뒤의 expression을 재귀적으로 처리하여 값을 가져오기 위함.) 로부터 반환된 값과 h를 unop연산을 통해 값을 반환할 수 있도록 처리하였다. Binary의 경우 튜플을 3개의 덩어리로 나눌 수 있다. (exp1, op, exp2)를 각각 (h,m,t) 로 분리하여 해당 h와 t에 대해 eval을 선언해 주었고, m은 어떤 연산을 수행할지 정하는 binop 이므로 이또한 앞서 정의한 bical 함수를 사용해 처리하였다.

```
k0s0a7@DESKTOP-MLPLCFD:~/week2/hw1$ vi arith.ml
k0s0a7@DESKTOP-MLPLCFD:~/week2/hw1$ dune exec ./main.exe
a = 4
b = 2
c = 14
k0s0a7@DESKTOP-MLPLCFD:~/week2/hw1$
```

[main문을 통해 주어진 테스트 케이스를 실행했을때의 결과 화면이다.]

● 위와 같이 함수들을 작성하고 실행했을 때 결과에 맞는 값이 나왔다. 진행하면서 부딪혔던 몇가지 오류 및 어려움이 있었는데, 이들에 대해서는 다음 장에 작성하였다.

<과제 구현 중 어려웠던 부분 및 해결 과정>

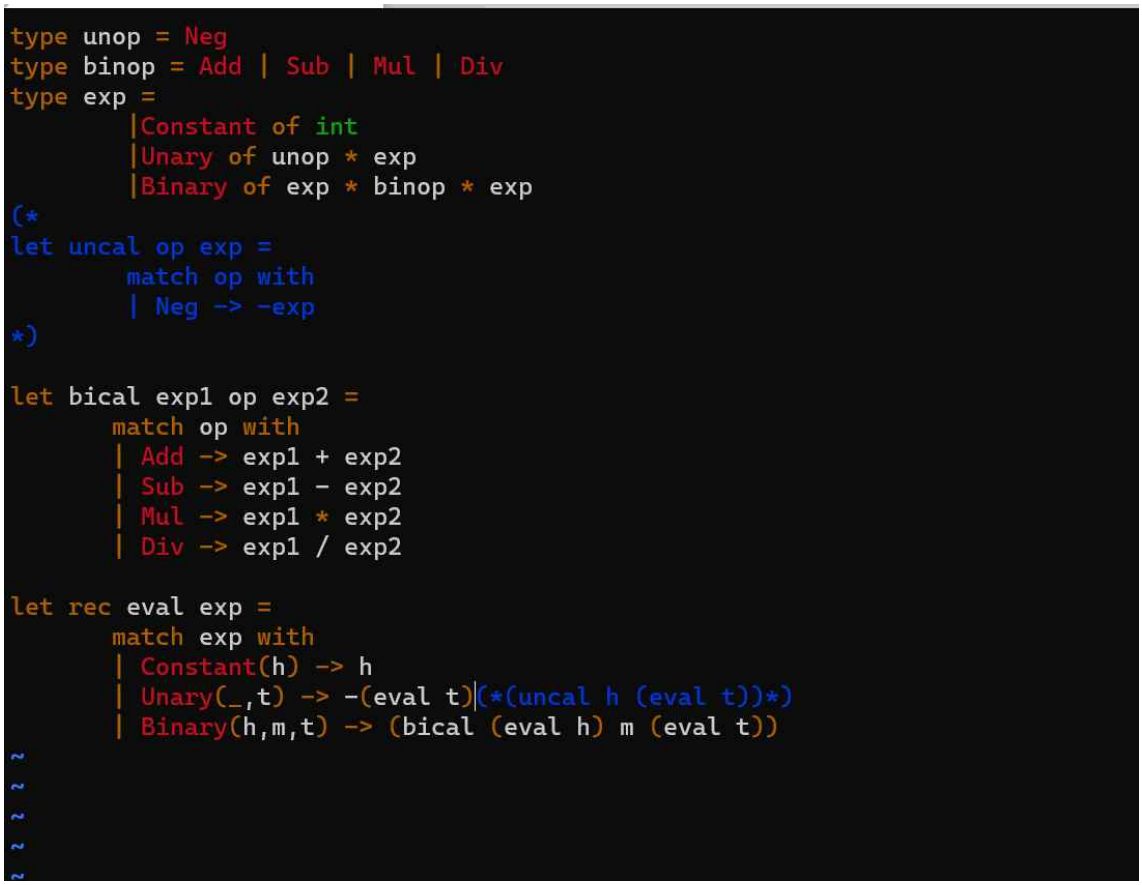
```
let bical exp1 op exp2 =
        match op with
        | "Add" -> exp1 + exp2
        | "Sub" -> exp1 - exp2
        | "Mul" -> exp1 * exp2
        | "Div" -> exp1 / exp2
```

[1. 오류가 있던 코드]

```
let bical exp1 op exp2 =
        match op with
        | Add -> exp1 + exp2
        | Sub -> exp1 - exp2
        | Mul -> exp1 * exp2
        | Div -> exp1 / exp2
```

[2. 해결된 코드]

● 초기 bical 함수를 작성하고, 오류가 나왔을 때 어디서 오류가 있다는 것인지 쉽게 찾기가 어려웠다. 한동안 eval함수를 가지고 끙끙대고 있었는데, 생각해보니 binop가 string이 아니라 데이터타입으로 이름이 정의된 것이라 생각하고 1번사진처럼 작성되었던 코드에서 "(쌍따옴표)를 제거하니 곧바로 해결되었다. 해결하고서 당연하다고 생각하고 코드를 작성했던 순간을 생각하며 반성했다. 다음에 같은 실수를 범하지 않도록 머리에 새겼다.

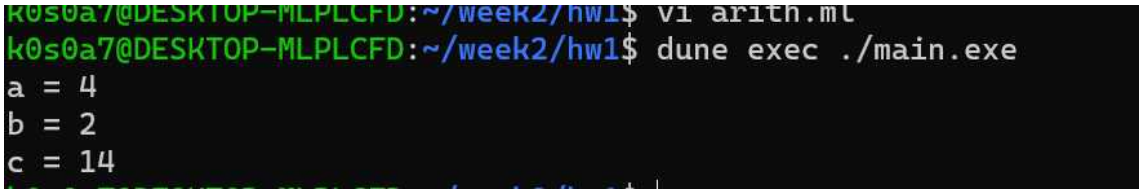


```
type unop = Neg
type binop = Add | Sub | Mul | Div
type exp =
        |Constant of int
        |Unary of unop * exp
        |Binary of exp * binop * exp
(*
let uncal op exp =
        match op with
        | Neg -> -exp
*)

let bical exp1 op exp2 =
       match op with
       | Add -> exp1 + exp2
       | Sub -> exp1 - exp2
       | Mul -> exp1 * exp2
       | Div -> exp1 / exp2

let rec eval exp =
       match exp with
       | Constant(h) -> h
       | Unary(_,t) -> -(eval t)(*(uncal h (eval t))*)
       | Binary(h,m,t) -> (bical (eval h) m (eval t))
~
~
~
~
~
```

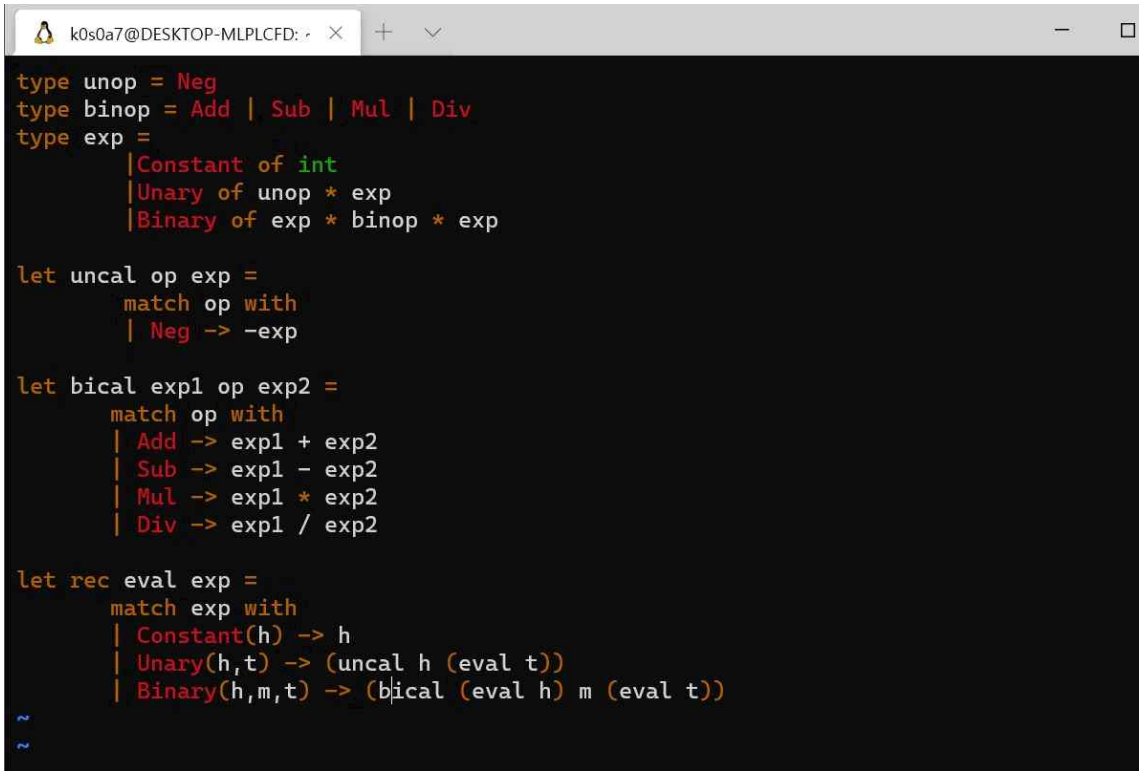
[1. 초기 작성한 arith.ml 코드(주석제외입니다!)]



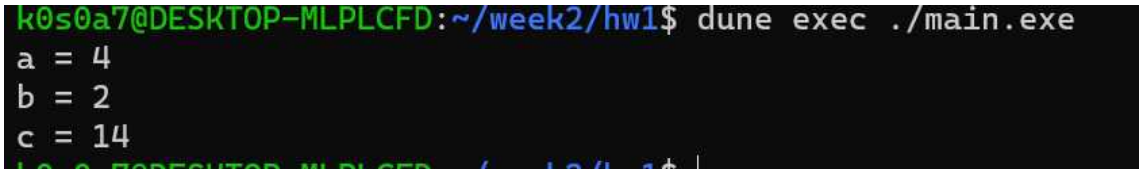
```
k0s0a7@DESKTOP-MLPLCFD:~/week2/hw1$ vi arith.ml
k0s0a7@DESKTOP-MLPLCFD:~/week2/hw1$ dune exec ./main.exe
a = 4
b = 2
c = 14
```

[2. 실행결과]

● 초기 arith.ml은 제출한 결과물과 다른 코드였다. 1번 사진의 eval함수를 보면 Unary에 대한 매칭을 Unary(_,t) -> -(eval t)로 wildcard를 이용해 처리하였다. 이유는 unop가 Neg연산 하나밖에 정의되어 있지 않았기 때문에 그냥 단순히 “아 그냥 음수로 바꿔주면 되겠구나” 생각하고 작성하였다. 또한 작성을 끝내고 몇몇 오류를 해결한 다음엔 2번사진처럼 결과까지 제대로 나왔다. 이대로 제출하려하다가 갑자기 마음에 안드는 부분이 생겼다. 만일 unop연산이 두 개 이상이라면 이렇게 작성하면 안된다는 생각이 들었다.



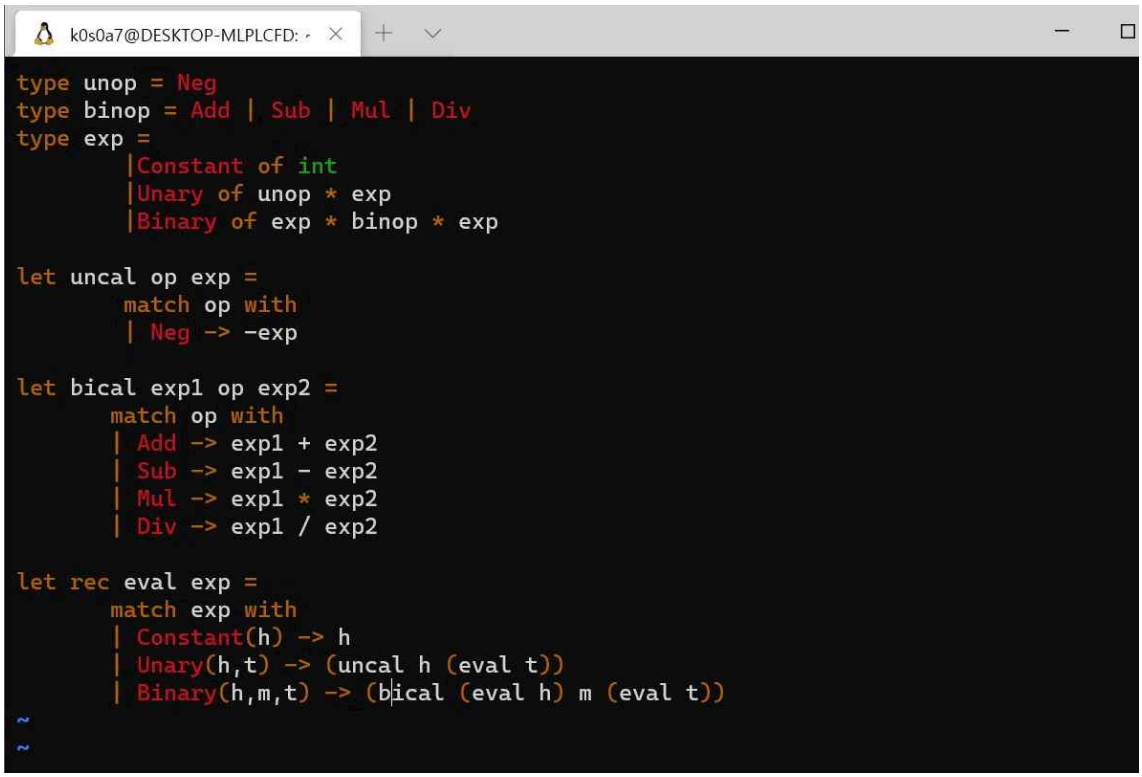
[1. unop연산이 두 개이상일 경우를 염두하고 수정한 arith.ml 코드]



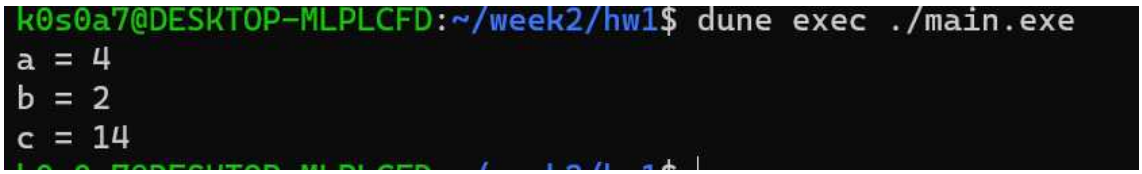
[2. 실행결과]

● 그렇게 전에 썼던 코드를 수정하자 마음먹고, 다시 작성한 코드가 위의 1번사진이다. 앞에 코드 설명에도 나와있지만 unop의 연산이 두 개 이상 혹은 추가하게 될 경우 eval함수 자체를 바꿔주면 안된다고 생각하고 eval함수에서는 wildcard를 제거한 뒤 h를 넣음으로써 uncal 함수를 호출할 준비를 해주었다. 이제 unop연산을 추가하게 되더라고 type의 정의 부분과 uncal함수의 매칭 경우만 늘려주면 되니 좀더 유지보수에 유리한 코드가 되었다고 생각했다. 결과 또한 잘나왔다(2번 사진).

<최종 제출 코드 및 결과 화면>



[1. 최종제출한 arith.ml 코드]



[2. 최종제출 코드의 실행결과]

● 제출한 arith.ml의 내부 코드와 해당 arith.ml에 대해 main.ml을 실행한 결과 화면이다. 원하는 결과를 얻을 수 있었고, 이번 과제를 진행하면서 데이터타입을 정의한다는 것의 의미가 무엇인지 조금은 더 이해된 것 같다.

<과제를 하며 어려웠던 부분 및 새로 고찰한 내용>

● 이번 과제를 하면서 어려웠던 부분은 오류에 대한 해결이었다. 앞서 작성된 글에서처럼 string으로 처리할 생각은 없었지만 무심결에 작성한 코드에서 오류를 범하고 있었다. 또한 지난주 실습에서 let과 let-in 구문에 대한 차이를 알기위해 노력하고 이번 실습에서 module을 정의하고 데이터타입을 정의하는 것을 이해하는 부분에 초점을 맞추어 진행하니 Ocaml이라는 언어가 낯설게 느껴지기 보다는 점차 흥미를 가질 수 있게 되는 것 같다. 또한 지난주에 비해 Ocaml언어를 작성하는데 어려움을 크게 느끼지 않았다. 이번주 실습 과제에서는 처음 작성한 코드를 보완하고자 노력해보면서 완벽하게 깔끔한 코드는 아닐지라도 스스로 많이 배웠음을 느낀다. 마지막으로 데이터타입을 이해하는 과정에서 튜플에 대한 정의와 타입 (int -> int) 와 (int*int) 같이 “그냥 다른가보군” 했던 내용들도 점차 머릿속에서 안정화 되어가는 것 같아 기분이 좋았고 자신감도 생기게 되었다. 앞으로도 열심히 해야겠음을 느꼈다.