Homework 2

SNU 4190.310, Fall 2013

Kwangkeun Yi

due: 9/28, 24:00

Exercise 1 (10pts) "Galculator"

다음의 계산기

```
galculator: exp -> float
```

를 만듭시다.

```
type exp = X
         | INT of int
         | REAL of float
         | ADD of exp * exp
         | SUB of exp * exp
         | MUL of exp * exp
         | DIV of exp * exp
         | SIGMA of exp * exp * exp
         | INTEGRAL of exp * exp * exp
```

예를들어 우리가 쓰는 수식이 exp타입으로는 다음과 같이 표현된다:

$\sum_{x=1}^{10}(x * x - 1)$	`SIGMA(INT 1, INT 10, SUB(MUL(X, X), INT 1))`
$\int_{x=1.0}^{10.0}(x * x - 1)\mathrm{d}x$	`INTEGRAL(REAL 1.0, REAL 10.0, SUB(MUL(X, X), INT 1))`

적분식을 계산할때의 알갱이 크기($\mathrm{d}x$)는 0.1로 정한다.

□

Exercise 2 (10pts) "Mathemadiga"

고등학교때는 손으로하고, Maple이나 Mathematica에서는 자동으로 해주던 미분식 전개를 만들어봅시다.

```
diff: ae * string -> ae
```

`diff`는 식(algebraic expression)과 변수를 받아서 주어진 식을 변수로 미분한 결과 식을 돌려 줍니다. 예를들어, 식 $ax^2 + bx + c$을 x에 대해 미분시키면 $2ax + b$를 내 놓는 것입니다. 미분된것을 될 수 있으면 최소의 꼴로 줄이거나 등등의 작업을 하는 것은 자유입니다. 미분할 식은 다음의 `ae` 타입입니다:

```
type ae = CONST of int
        | VAR of string
        | POWER of string * int
        | TIMES of ae list
        | SUM of ae list
```

□

Exercise 3 (10pts) "Queue = 2 Stacks"

큐는 반드시 하나의 리스트일 필요는 없습니다. 두개의 스택으로 큐를 효율적으로 규현할 수 있습니다. 큐에 넣고 빼는 작업이 거의 한 스텝에 이루어질 수 있습니다. (하나의 리스트위를 더듬는 두 개의 포인터를 다루었던 C의 구현과 장단점을 비교해 보세요.)

각각의 큐 연산들의 타입들은:

```
emptyQ: queue
enQ: queue * element -> queue
deQ: queue -> element * queue
```

큐를 $[a_1; \cdots; a_m; b_1; \cdots; b_n]$ 라고 합시다 (b_n이 머리). 이 큐를 두개의 리스트 L과 R로 표현할 수 있습니다:

$$L = [a_1; \cdots; a_m], \quad R = [b_n; \cdots; b_1].$$

한 원소 `x`를 삼키면 새로운 큐는 다음이 됩니다:

$$[x; a_1; \cdots; a_m], [b_n; \cdots; b_1].$$

원소를 하나 빼고나면 새로운 큐는 다음이 됩니다:

$$[a_1; \cdots; a_m], [b_{n-1}; \cdots; b_1].$$

뺄 때, 때때로 L 리스트를 뒤집어서 R로 같다 놔야하겠습니다. 빈 큐는 $([], [])$ 이겠지요.

다음과 같은 Queue 타입의 모듈을 작성합니다:

```
module type Queue =
  sig
    type element
    type queue
    exception EMPTY_Q
    val emptyQ: queue
    val enQ: queue * element -> queue
    val deQ: queue -> element * queue
  end
```

다양한 큐 모듈이 위의 Queue 타입을 만족시킬 수 있습니다. 예를들어:

```
module IntListQ =
   struct
     type element = int list
     type queue = ...
     exception EMPTY_Q
     let emptyQ = ...
     let enQ = ...
     let deQ = ...
   end
```

는 정수 리스트를 큐의 원소로 가지는 경우겠지요. 위의 모듈에서 함수 enQ와 deQ를 정의하기 바랍니다.

이 모듈에 있는 함수들을 이용해서 큐를 만드는 과정의 예는:

```
let myQ = IntListQ.emptyQ
let yourQ = IntListQ.enQ(myQ, [1])
let (x,restQ) = IntListQ.deQ yourQ
let hisQ = IntListQ.enQ(myQ, [2])
```

□

Exercise 4 (20pts) "계산실행"

아래 ZEXPR 꼴을 가지는 모듈 Zexpr를 정의해 봅시다.

```
signature ZEXPR =
sig
```

```
    exception Error of string
    type id = string
    type expr = NUM of int
              | PLUS of expr * expr
              | MINUS of expr * expr
              | MULT of expr * expr
              | DIVIDE of expr * expr
              | MAX of expr list
              | VAR of id
              | LET of id * expr * expr
    type environment
    type value
    val emptyEnv: environment
    val eval: env * expr -> value
  end
```

ZEXPR.expr 타입의 식을 E라고 하면,

ZEXPR.eval (ZEXPR.emptyEnv, E)

는 식 E를 실행시키게 되는데, 성공적으로 끝나면 최종 값을 프린트하고 끝나게 됩니다. 이때, VAR "x"는 x라고 명명된 값을 뜻합니다.

이름을 정의하고 그 유효범위를 한정하는 식은 LET("x", E_1, E_2)입니다. 이 경우 E_1값을 계산해서 x라고 이름짓게 되고, 그 이름의 유효범위는 E_2로 한정됩니다. 현재 환경에서 정의되지 않은 이름이 식에서 사용되면 그 식은 의미가 없습니다. 예를 들어,

```
LET("x", 1,
    PLUS (LET("x", 2, PLUS(VAR "x", VAR "x")),
          VAR "x")
   )
```

의 계산결과는 5입니다.

```
LET("x", 1,
    PLUS (LET("y", 2, PLUS(VAR "x", VAR "y")),
          VAR "x")
   )
```

의 계산결과는 4입니다.

```
LET("x", 1,
```

```
PLUS (LET("y", 2, PLUS(VAR "y", VAR "x")),
      VAR "y")
)
```

는 의미가 없습니다. 바깥 PLUS식의 환경에서 y가 정의되어있지 않기 때문입니다.

MAX식은 정수식 리스트에서 가장 큰 정수를 찾아내는 정수식입니다. 즉, MAX [NUM 1, NUM 3, NUM 2]의 계산결과는 3입니다. MAX가 빈 리스트를 받은 경우 결과는 0입니다. □