Homework 2

SNU 4190.310, Fall 2015

Kwangkeun Yi

due: 9/30, 24:00

Exercise 1 (10pts) "k-진수"

일반적으로 k진수($k > 1$)는 다음과 같이 표현한다.

$$d_0 \cdots d_n$$

여기서

$$\forall d_i \in \{0, \cdots, k-1\}.$$

그리고 "$d_0 \cdots d_n$"은 크기가

$$d_0 \times k^0 + \cdots + d_n \times k^n$$

인 정수를 표현한다.

이것을 살짝 확장해서 "k진수"를 다음과 같이 정의해보자. 표현은

$$d_0 \cdots d_n$$

여기서

$$\forall d_i \in \{1-k, \cdots, 0\} \cup \{0, \cdots, k-1\}.$$

그리고 "$d_0 \cdots d_n$"은 크기가

$$d_0 \times k^0 + \cdots + d_n \times k^n$$

인 정수를 표현한다.

예를 들어, 2진수의 경우를 생각하자. 베이스가 $\{-1, 0, 1\}$이 되겠다. `0`이 0을, `+`가 1을 `-`가 -1을 표현한다고 하면, `+` 는 1을, `+0+`는 5를, `+-`는 -1을, `+-0-`는 -9인 정수를 표현한다.

OCaml로 2친수라는 타입을 다음과 같이 정의했다:

```
type crazy2 = NIL | ZERO of crazy2 | ONE of crazy2 | MONE of crazy2
```

예를 들어, 0+-은

```
ZERO(ONE(MONE NIL))
```

로 표현된다.

위와 같이 표현되는 2친수를 받아서 그것의 값을 계산하는 함수 crazy2val을 정의하세요.

```
crazy2val: crazy2 -> int.
```

□

Exercise 2 (10pts) "친수의 합"

두 2친수를 받아서 2친수의 합에 해당하는 2친수를 내어놓는 함수 crazy2add를 정의하세요.

```
crazy2add: crazy2 * crazy2 -> crazy2
```

위의 crazy2add는 다음의 성질이 만족되야 한다: 임의의 2친수 z 과 z'에 대해서

$$\texttt{crazy2val (crazy2add(}z\texttt{,}z'\texttt{))} = \texttt{crazy2val(}z\texttt{)} + \texttt{crazy2val(}z'\texttt{)}.$$

□

Exercise 3 (10pts) "CheckMetroMap"

아래 metro 타입을 생각하자:

```
type metro = STATION of name
           | AREA of name * metro
           | CONNECT of metro * metro
and name = string
```

아래 checkMetro 함수를 정의하라:

```
checkMetro: metro -> bool
```

checkMetro는 주어진 metro 가 제대로 생겼는지를 확인해 준다. "metro가 제대로 생겼다"는 것은(iff) 메트로 역 이름(id in STATION(id))들이 항상 자기 이름의 지역(m in AREA(id, m))에서만 나타나는 경우를 뜻한다.

예를들어, 제대로 생긴 metro 들은:

- `AREA("a", STATION "a")`
- `AREA("a", AREA("a", STATION "a"))`
- `AREA("a", AREA("b", CONNECT(STATION "a", STATION "b")))`
- `AREA("a", CONNECT(STATION "a", AREA("b", STATION "a")))`

그렇지 못한 것들의 예들은:

- `AREA("a", STATION "b")`
- `AREA("a", CONNECT(STATION "a", AREA("b", STATION "c")))`
- `AREA("a", AREA("b", CONNECT(STATION "a", STATION "c")))`

□

Exercise 4 (15pts) "짚-짚-나무"

임의의 나무를 여러분 바지의 "지퍼"로 구현할 수 있답니다.

- 나무구조 타입은 아래와 같이 정의됩니다:

```
type item = string
type tree = LEAF of item
          | NODE of tree list
```

- 아래의 `zipper`가 나무의 줄기를 타고 자유자재로 찢어놓기도 하고 붙여놓기도 합니다.

```
type zipper = TOP
            | HAND of tree list * zipper * tree list
```

  현재 나무줄기의 어느지점에 멈춰 있는 지퍼손잡이 `HAND(l,z,r)`에서, `l`은 왼편 형제 나무들(elder siblings)이고 `r`은 오른편 형제 나무들(younger siblings)이다.

- 나뭇가지에서의 현재 위치 `location`는 현재위치를 뿌리로하는 나무자체와 지퍼(`zipper`)로 표현되는 주변 나무들로 구성된다.

```
type location = LOC of tree * zipper
```

- 예를들어, "$a \times b + c \times d$" 가 다음과 같은 나무구조로 표현될 것이다. 모든 심볼은 항상 잎새에 매달리게 된다.

```
NODE [ NODE [LEAF a; LEAF *; LEAF b];
```

```
        LEAF +;
        NODE [LEAF c; LEAF *; LEAF d]
      ]
```

두번째 곱셈표에의 위치는 다음과 같다:

```
LOC (LEAF *,
     HAND([LEAF c],
          HAND([LEAF +; NODE [LEAF a; LEAF *; LEAF b]],
               TOP,
               []),
          [LEAF d]))
```

- 자, 주어진 위치에서 이제 자유자재로 나무를 탈 수 있습니다. 왼편으로 옮겨가는 것은 다음과 같지요:

```
let goLeft loc = match loc with
    LOC(t, TOP) -> raise (NOMOVE "left of top")
  | LOC(t, HAND(l::left, up, right)) -> LOC(l, HAND(left, up, t::right))
  | LOC(t, HAND([],up,right)) -> raise NOMOVE "left of first"
```

- 다음의 나머지 함수들을 정의하세요:

```
goRight: location -> location
goUp: location -> location
goDown: location -> location
```

□

Exercise 5 (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 6 (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 x를 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 갈다 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 7 (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. □