Homework 8
SNU 4190.210 Fall 2014

Chung-Kil Hur

due: 12/10 (Wed) 24:00

이번 숙제의 목적은

- 모듈(module) 프로그래밍 연습하기.
- 자동으로 타입 검증을 해주는 환경에서 모듈 프로그래밍 연습하기.

Exercise 1 "큐 모듈"

큐는 원소들이 들어가는 순서대로 나오는(first-in-first-out) 구조인데, 큐는 반드시 하나의 리스트로 구현할 필요는 없습니다. 두개의 리스트로 큐를 효율적으로 규현할 수 있습니다. 큐에 넣고 빼는 작업이 거의 한 스텝에 이루어질 수 있지요.

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 갈다 놔야하겠습니다. 빈 큐는 $([], [])$ 이겠지요.

다음과 같은 `Queue` 타입

```
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
```

의 모듈 두 개(아래)를 위의 아이디어를 이용해서 구현하세요:

- 모듈 `StringQ`: 큐의 원소가 문자열인 경우.
- 모듈 `StringQQ`: 큐의 원소가 `StringQ.queue`인 경우.

이 때 큐의 원소들은 중복될 수 있습니다.

아래와 같이 모듈 정의에 모듈 타입(인터페이스 interface)을 명시해서 ("`: Queue`" 부분)

```
module StringQ: Queue = struct ... end
module StringQQ: Queue = struct ... end
```

다음 두 가지를 자동으로 검증하도록 OCaml시스템에 힌트를 줍시다:

- 첫째는, 모듈들이 `Queue` 타입과 어울리는 지(최소한 `Queue`타입에서 드러낸 것들을 정의하고 있는 지) 자동으로 확인하게 합니다.

  위와 같이 정의한 것이 OCaml시스템의 타입 시스템을 무사히 통과했다면, 여러분이 정의한 두 모듈은 최소한 `Queue`에서 드러낸 것들을 정의한 것입니다.

- 두 번째는, 외부에서 `StringQ`와 `StringQQ`를 사용할 때 `Queue`에서 드러낸 것만을 사용하는 지 자동으로 확인하게 합니다.

  그 이상의 것을 외부에서 사용하게 되면 타입 시스템을 통과하지 못하게 됩니다. 프로그래머가 `Queue`라는 인터페이스만을 외부에 공개해 놓고는 그 이상의 것을 사용하려고 하니까요.

  이 상황을 겪어봅시다:

  - 이 두개의 모듈에 있는 함수들을 이용해서 큐를 만드는 과정의 예로 아래같이 한다고 합시다.

    ```
    let csQ = StringQ.enq
    ```

```
(StringQ.enq
 (StringQ.emptyq, "길동"),
 "춘향")
```

위와 같은 프로그램은 Queue에서 정한 것 이상을 가지고 프로그램한 경우입니다. 모듈 타입 Queue에서는 큐의 원소 타입 element이 무엇인지를 공개하지 않고 있습니다. 타입이라는 사실이외에는 정보가 없습니다. 따라서 위에서 StringQ.enq를 사용할 때 원소가 문자열 타입이라는 것을 가정하고 "길동"이나 "춘향"을 원소로 전달하면 "협정위반"이 됩니다. OCaml은 이러한 상황을 엄밀히 확인하고 검증해 줍니다.

- 이를 해결하기 위해서는 아래와 같이:

```
module StringQ: Queue with type element = string
 = struct ... end
module StringQQ: Queue with type element = StringQ.queue
 = struct ... end
```

모듈 타입 Queue의 타입 element가 무엇인지를 보조로 표현해야 합니다 ("with type element = string" 부분).

그러면 이제는 외부에서 StringQ의 element타입으로 문자열을 사용할 수 있고, StringQQ의 element타입으로 StringQ.queue를 사용할 수 있게 됩니다.

□

Exercise 2 "집합큐도 큐"

같은 모듈 타입에 맞는 다른 정의들을 한 번 해 봅시다. 이제 위에 구현한 두 개의 큐 모듈을 "집합큐"를 구현하는 것으로 바꿉시다. 집합큐란 큐인데 중복된 원소들을 가지고 있지 않은 큐입니다. 집합큐는 넣으려는 원소가 이미 있으면 넣지 않아야 겠지요.

이때, 집합큐들은 위에 정의한 모듈 타입 Queue를 만족해야 합니다. 위의 모듈 StringQ와 StringQQ를 변형해서

```
module StringSetQ: Queue with type element = string
 = struct ... end
module StringSetQQ: Queue with type element = StringSetQ.queue
 = struct ... end
```

를 완성하세요. □

Exercise 3 "큐 모듈함수"

위의 Exercise1에서, 큐 모듈을 매번 따로 만들게 되면 거의 같은 정의를 반복해서 하게 되는 것을 겪었을 겁니다. 모듈함수 QueueMake를 하나 만들어서 다양한 큐를 QueueMake를 이용해서 공짜로 쉽게 만들 수 있도록 합시다.

모듈함수 QueueMake는 다음과 같이 정의 됩니다:

```
module QueueMake (Arg: ArgTy): Queue with type element = ...
  = struct ... end
```

인자로 전달되는 모듈 Arg은 모듈 타입 ArgTy과 어울려야 하고(최소한 그만큼의 정의가 있어야 하고), 만들어 내는 모듈은 Queue 타입과 어울려야 합니다. 그래서 다양한 큐가 아래와 같이 쉽게 만들어 지게끔:

```
module StringQ = QueueMake(...)
module StringQQ = QueueMake(...)
```

그래서 아래와 같이 사용할 수 있는 지를 확인하세요.

```
let csQ = StringQ.enq
          (StringQ.enq
           (StringQ.emptyq, "길동"),
           "춘향")
```

□

Exercise 4 "창살 디자인"

창살 무늬의 전체구조는 대게가 작은 기본 무늬들의 반복입니다. 기본 무늬는 정 사각형에 그려져있다고 합시다. 무늬를 디자인하는 작업은 기본 정사각형들을 4개로 짜집기해서 큰 정사각형의 모양을 만들고, 이것들 4개로 다시 보다 큰 정사격형을 만들고, 되었다 싶으면 디자인된 정 사각형들을 반복해서 창살에 짜넣는 방법을 취하겠지요.

그 작업 과정을 모듈함수(functor)로 표현할 수 있는데, 다음과 같습니다. 코드를 잘 읽어보고, 코드에서 "..." 부분을 메꾸어 보세요.

```
type design = TURTLE | WAVE | DRAGON  (* three design patterns *)
type orientation = NW | NE | SE | SW
type box = BOX of orientation * design | GLUED of box * box * box * box
module type FRAME =
  sig
   val box: box
   val rotate: box -> box (* rotate box M to 3 to W to E *)
   val pp: box -> int * int -> unit (* pretty printer *)
```

```
   val size: int
  end
module BasicFrame (Design: sig val design: design end): FRAME =
  struct
    exception NON_BASIC_BOX
    let box = BOX (NW, Design.design)    (* a box is defined *)
    let rotate = ...
    let pp b center = match b with
        BOX(NW,x) -> () (* dummy, fill it if you want *)
      | BOX(NE,x) -> () (* dummy, fill it if you want *)
      | BOX(SE,x) -> () (* dummy, fill it if you want *)
      | BOX(SW,x) -> () (* dummy, fill it if you want *)
      | _ -> raise NON_BASIC_BOX
    let size = 1
  end
module Rotate (Box: FRAME): FRAME =
  struct
   ...
  end
module Glue (Nw: FRAME) (Ne: FRAME) (Se: FRAME) (Sw: FRAME): FRAME =
  struct
   exception DIFFERENT_SIZED_BOXES
   ...
  end
```

자, 위의 코드를 이용한 다음의 디자인과정을 보세요:

```
module A = BasicFrame(struct let design = TURTLE end)
module B = BasicFrame(struct let design = WAVE end)
module A' = Rotate(A)
module A'' = Rotate(A')
module B' = Rotate(B)
module B'' = Rotate(B')
module A4 = Glue (A) (B) (A') (B')
module B4 = Glue (A) (A') (B) (B')
module A4' = Rotate(A4)
```

```
module B4' = Rotate(B4)
module C = Glue (A4) (B4) (A4') (B4')
let bluePrint = C.pp C.box
□
```