프로그래밍의 원리 2012 가을
실습 7
물건 중심, 값 중심 프로그래밍 다시 보기, 8-퀸 퍼즐

서울대학교 프로그래밍 연구실
강동옥, 최민아

2012년 10월 31일

이번 실습의 목적은

- 물건 중심 프로그래밍과 값 중심 프로그래밍을 연습해 양쪽 모두에 능숙해지도록 한다.
- 응용 문제를 풀어봄으로써 스스로 사고하는 능력을 기른다.

1. 스택(stack)을 값중심, 물건 중심으로 만들어 봅시다. `empty`는 빈 스택을 만들고, `push`은 스택에 맨 위에 원소를 넣고, `is-empty?`는 스택이 비어있는지 확인하며, `pop`은 스택의 맨 위의 원소를 꺼내 원소와 원소가 하나 없어진 스택을 돌려줍니다. 값중심 함수들의 타입은 이렇습니다.

`empty-stack-applicative` :	$stack$
`push-stack-applicative` :	$stack * element \rightarrow stack$
`is-empty?-stack-applicative` :	$stack \rightarrow bool$
`pop-stack-applicative` :	$stack \rightarrow element \times stack$

   물건 중심 함수들의 타입은 이렇습니다.

`empty-stack-imperative` :	$unit \rightarrow stack$
`push-stack-imperative` :	$stack * element \rightarrow unit$
`is-empty?-stack-imperative` :	$stack \rightarrow bool$
`pop-stack-imperative` :	$stack \rightarrow element$

   물건 중심 함수들의 뼈대 코드입니다.

```
(define (empty-stack-imperative) (cons 'stack 'bottom))
(define (push-stack-imperative s x)
  (let ((cell (cons x ())))
    (begin (set-cdr! cell (cdr s)) (set-cdr! s cell))
    ))
(define (is-empty?-stack-imperative s) <??>)
(define (pop-stack-imperative s)
  (if (is-empty?-stack-imperative s) (error "stack is empty")
      (let ((top <??>))
        (begin (set-cdr! s <??>) top)
```

```
        )))
```

테스트케이스입니다.

```
(define empty-stk-app empty-stack-applicative)
(define stk-app (push-stack-applicative empty-stk-app 5))
(is-empty?-stack-applicative stk-app)
#f
(define pair (pop-stack-applicative stk-app))
(define elmt (car pair))
elmt
5
(is-empty?-stack-applicative (cdr pair))
#t
(newline)
(define stk-imp (empty-stack-imperative))
(push-stack-imperative stk-imp 5)
(is-empty?-stack-imperative stk-app)
#f
(define elmt (pop-stack-imperative stk-imp))
elmt
5
(is-empty?-stack-imperative stk-imp)
#t
```

2. 위에서 만든 값중심 스택 2개를 사용해 값중심 큐(queue)를 만들어 봅시다. 스택 2개로 큐를 어떻게 만들 수 있을지 각자 생각해 구현해 봅시다.

$$
\begin{aligned}
\texttt{empty-queue}: &\quad \textit{queue} \\
\texttt{insert-queue}: &\quad \textit{queue} * \textit{element} \rightarrow \textit{queue} \\
\texttt{is-empty?-queue}: &\quad \textit{queue} \rightarrow \textit{bool} \\
\texttt{delete-queue}: &\quad \textit{queue} \rightarrow \textit{element} \times \textit{queue}
\end{aligned}
$$

테스트케이스입니다.

```
(define queue (insert-queue (insert-queue empty-queue 5) 7))
(define elm-queue (delete-queue queue))
(car elm-queue)
5
(define queue2 (cdr elm-queue))
(define elm-queue2 (delete-queue queue2))
(car elm-queue2)
7
(is-empty?-queue (cdr elm-queue2))
#t
```

3. (SICP ex. 2.42) "8-퀸 퍼즐(eight-queens puzzle)"은 체스판에 퀸 여덟을 놓되 서로 공격할 수 있는 자리에 오지 못하게 두는 방법이 무엇인지 찾아내는 수수께끼입니다. (다시 말해서, 두 퀸이 마주보지 않도록 가로줄, 세로줄, 대각선 위에 퀸 두개가 오면 안 됩니다.) 이 수수께끼를 푸는 한 가지 방법은, 세로줄마다 하나씩 퀸을 놓아 보는

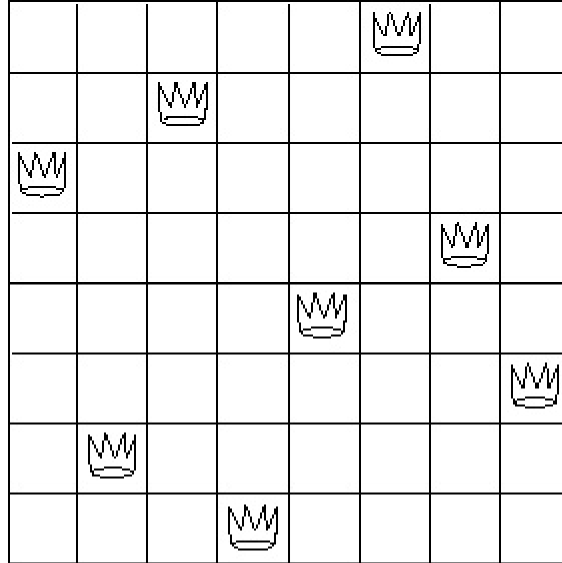

그림 1: 8-퀸 퍼즐을 푸는 한 가지 방법

것입니다. 따라서 퀸 $k-1$개를 제대로 두었다고 치면, k번째 퀸은 다른 퀸을 공격하지 못하는 자리에 와야 합니다.

이런 방식을 재귀적으로 정리해 보면 이렇습니다. 퀸 $k-1$개를 세로줄 $k-1$개에 두는 방법을 모두 찾아냈다고 칩시다. 그렇게 찾아낸 방법 하나하나에 대하여, k번째 세로줄의 모든 가로줄마다 k번째 퀸을 놓아봅니다. 이로부터 k번째 세로줄에 있는 퀸이 공격 받지 않는 자리 값만 골라냅니다. 이리하면 k개 세로줄에 퀸 k개를 안전하게 놓을 수 있는 방법을 몽땅 얻을 수 있습니다. 이런 과정을 계속 밟아 가면, 답 하나가 아니라 이 수수께끼를 푸는 모든 답을 얻어낼 수 있습니다.

이 풀이법을 queens 프로시저로 실현해 봅시다. queens는 퀸 n개를 $n \times n$ 체스판에 놓는다고 할 때, 얻을 수 있는 모든 답을 리스트로 묶어 냅니다. queens에 갇힌 프로시저 queen-cols는 처음 k개의 세로줄에 퀸을 놓을 수 있는 모든 방법을 리스트로 묶어냅니다.

```
(define (queens bs)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-b)
        (filter
         (lambda (p) (safe? p))
         (accumulate append
                     null
                     (map <??>
                          (queen-cols (- k 1)))))))
  (queen-cols bs))
```

이 프로시저에서

- empty-b는 공집합입니다.

- 프로시저 `safe?`는 체스판에서 퀸의 위치들의 집합을 받아, k번째 세로줄의 퀸이 나머지 다른 퀸에 대해 안전한지를 알아봅니다. (이 때, 나머지 퀸들은 벌써 안전한 자리를 잡고 있다 치고, 새로 놓을 퀸이 안전한지만 따져봅니다.)
- `accumulate` 함수는 리스트의 원소들과 두 번째 인자에 인자로 받은 함수를 적용합니다.

  코드의 빈 칸을 채워 봅시다.

사용할 수 있는 도움 함수는 다음과 같습니다.

- 집합에 새로운 가로, 세로 자리 값을 넣는 `adjoin-position` 함수를 사용합니다.

  ```
  (adjoin-position new-row k rest-of-queens)
  ```

  와 같이 사용하실 수 있습니다.
- 자연수 n을 받아 1부터 n까지의 리스트를 만드는 `enumerate-interval` 함수도 사용할 수 있습니다.