SNU 4190.210 프로그래밍 원리(Principles of Programming) Part III

Prof. Kwangkeun Yi

차례

다음

1 값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

2 프로그램의 이해: 환경과 메모리(environment & memory)

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

양쪽 프로그래밍 방식에 능숙해야

- 값: 변하지 않는다(value, immutable)

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

양쪽 프로그래밍 방식에 능숙해야

- 값: 변하지 않는다(value, immutable)
  - (+ 2 1)는 3을 의미; 2가 변해서 3이 되는 것이 아님.

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

양쪽 프로그래밍 방식에 능숙해야

- 값: 변하지 않는다(value, immutable)
  - `(+ 2 1)`는 3을 의미; 2가 변해서 3이 되는 것이 아님.
  - `(add-element 1 S)`는 $S \cup \{1\}$인 집합을 의미.
  - S에 1 이 첨가된, 변화된 S를 의미하지 않음.

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

양쪽 프로그래밍 방식에 능숙해야

- 값: 변하지 않는다(value, immutable)
  - `(+ 2 1)`는 3을 의미; 2가 변해서 3이 되는 것이 아님.
  - `(add-element 1 S)`는 $S \cup \{1\}$인 집합을 의미.
  - S에 1 이 첨가된, 변화된 S를 의미하지 않음.
  - 지금까지의 프로그래밍 방식(상위의. 수리논술 방식)

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

- 물건: 상태가 변한다(state, mutable)

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

- 물건: 상태가 변한다(state, mutable)
  - `(add-element S 1)`는 집합 `S`가 변해서 `1` 이 첨가된 새로운 집합이 됨.

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

- 물건: 상태가 변한다(state, mutable)
  - `(add-element S 1)`는 집합 `S`가 변해서 `1` 이 첨가된 새로운 집합이 됨.
  - 물건의 상태를 변화시키는, 이런 프로그램이 필요한 경우도 많다. (예: 1-데이타 n-구현을 돕던 "함수 테이블")

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

- 물건: 상태가 변한다(state, mutable)
  - `(add-element S 1)`는 집합 `S`가 변해서 `1` 이 첨가된 새로운 집합이 됨.
  - 물건의 상태를 변화시키는, 이런 프로그램이 필요한 경우도 많다. (예: 1-데이타 n-구현을 돕던 "함수 테이블")
  - 변하므로, 순서가 중요.

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

- 물건: 상태가 변한다(state, mutable)
  - `(add-element S 1)`는 집합 `S`가 변해서 `1` 이 첨가된 새로운 집합이 됨.
  - 물건의 상태를 변화시키는, 이런 프로그램이 필요한 경우도 많다. (예: 1-데이타 n-구현을 돕던 "함수 테이블")
  - 변하므로, 순서가 중요.
  - 물건의 상태를 변화시키는 명령형 프로그래밍 방식

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

예: `(append l r)` 함수

- 값중심(value, immutable): `l`과 `r`은 변하지 않음.

```
(define (append l r)
  (cond ((null? l) r)
        ((null? r) l)
        (else (cons (car l) (append (cdr l) r)))
        ))
```

- 물건중심(state, mutable): `l`이 변해서 `l@r`이 됨.

```
(define (append l r)
  (cond ((null? l) (begin (change l r) l))
        ((null? r) l)
        (else (begin (change (cdr l) (append (cdr l) r)) l))
        ))
```

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

프로그래밍 언어들은 대개

- 두가지 방식을 모두 구사할 수 있는 방안을 제공
- 단, 기본으로 지원하는 방식이 있고, 원한다면 다른 방식도 가능
  - Scheme: 값중심 > 물건중심
  - ML: 값중심 > 물건중심
  - Java: 물건중심 > 값중심
  - C: 물건중심 > 값중심
- 따라서, 데이타 속구현에서 두 방식중 하나를 선택해야

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

데이타 속구현을 두 방식중 하나로 선택.

- 겉(interface)으로 드러나는 기획의 차이

`empty` :	stk	vs	$unit \to stk$
`push` :	$stk * elmt \to stk$	vs	$stk * elmt \to unit$
`is-empty?` :	$stk \to bool$	vs	$stk \to bool$
`pop` :	$stk \to elmt \times stk$	vs	$stk \to elmt$

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

데이타 속구현을 두 방식중 하나로 선택.

- 속구현의 차이

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

데이타 속구현을 두 방식중 하나로 선택.

- 속구현의 차이

"applicative style"

```
(define empty ())
(define (push s x) (cons x s))
(define is-empty? null?)
(define (pop s) (if (is-empty? s) (error) s))
```

값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

데이타 속구현을 두 방식중 하나로 선택.

- 속구현의 차이

"applicative style"

```
(define empty ())
(define (push s x) (cons x s))
(define is-empty? null?)
(define (pop s) (if (is-empty? s) (error) s))
```

"imperative style"

```
(define empty (cons 0 0))
(define (push s x)
  (let ((cell (cons x nil)))
    (begin (set-cdr! cell (cdr s)) (set-cdr! s cell))
  ))
(define (is-empty? s) (= 0 (cdr s)))
(define (pop s)
  (if (is-empty? s) (error)
    (let ((top (cadr s)))
         (begin (set-cdr! s (cddr s)) (cons top s))
    )))
```

값중심 프로그래밍(applicative programming) 원리

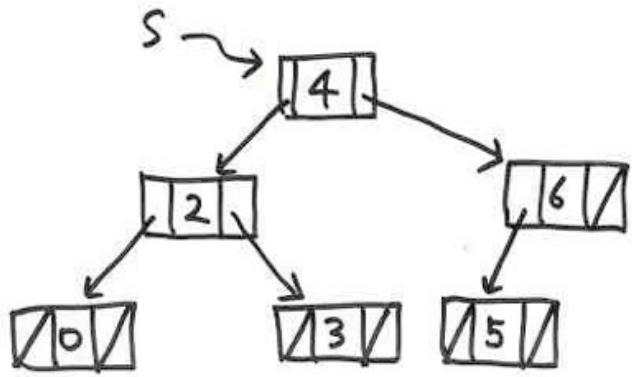
- 값은 변하지 않는 것
- 함수는 값을 받아 새로운 값을 만든다
- 값들은 변하지 않으므로 최대한 공유하도록 구현될 수 있다
  - See: 기계에서 구현되는 `(append l r)`, `(add-element S 1)` (예: $\mathrm{S} = \{0, 2, 3, 4, 5, 6\}$)

물건중심 프로그래밍(imperative programming) 원리

- 물건은 그 상태가 변하는 것
- 함수는 물건을 받아 기존의 물건을 변화시킬 수 있다
- 물건은 변하므로 공유하도록 구현되면 혼동스러울 수 있다
  - See: 기계에서 구현되는 `(append l r)`, `(add-element S 8)`
- 변하는 중간에 사용되므로, 순서가 중요

값중심과 물건중심 구현의 비용 비교

- $\mathtt{S} = \{0, 2, 3, 4, 5, 6\}$
- 구현: 이진 탐색 가지구조(binary search tree)



- 다음 함수의 구현: `(add-element S 9)`

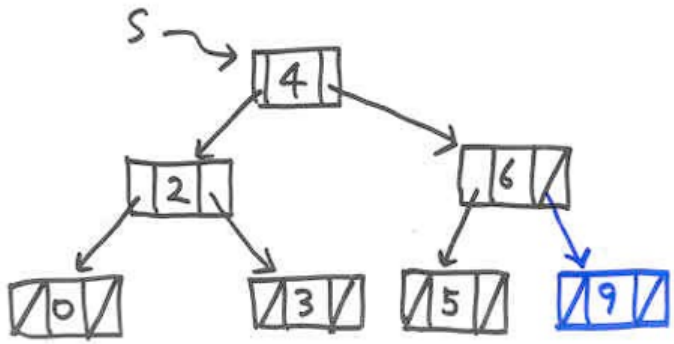
값중심과 물건중심 구현의 비용 비교

다음 함수의 구현: `(add-element S 9)`

값중심과 물건중심 구현의 비용 비교

다음 함수의 구현: `(add-element S 9)`

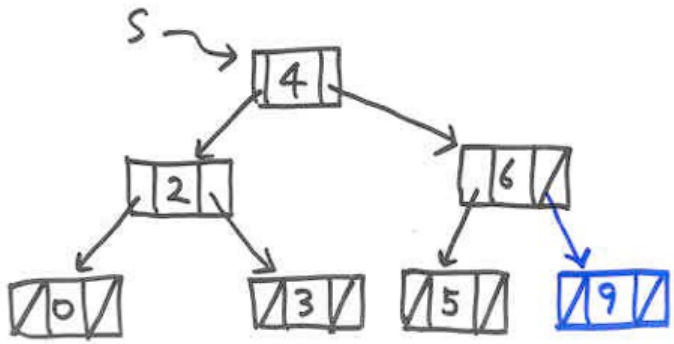
- 변화시키면서(imperative style)



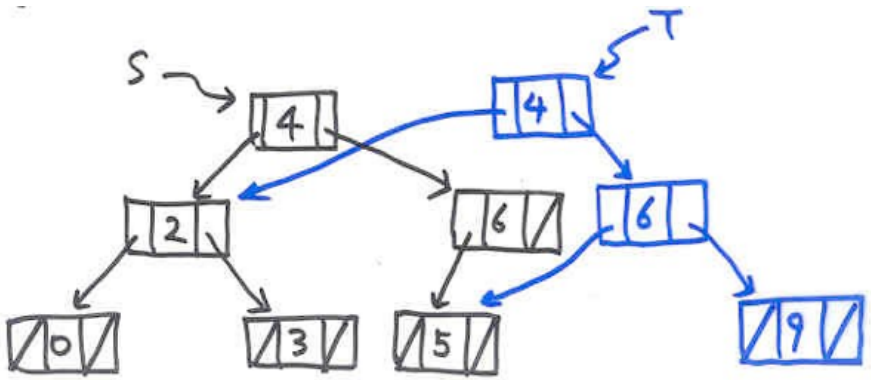
값중심과 물건중심 구현의 비용 비교

다음 함수의 구현: `(add-element S 9)`

- 변화시키면서(imperative style)



- 변화없이(applicative style)



값중심과 물건중심 구현의 비용 비교

원소를 넣는 경우

- 변화된 새것만 유지하면 되는 경우

	time	space
imperative	O(log N)	O(1)
applicative	O(log N)	O(1)

- 옛것과 새것 모두 유지해야 하는 경우

	time	space
imperative	O(N)	O(N)
applicative	O(log N)	O(log N)

다음

1 값중심 vs 물건중심 프로그래밍(applicative vs imperative programming)

2 프로그램의 이해: 환경과 메모리(environment & memory)

환경과 메모리(environment & memory)

프로그램 실행을 이해하는 데 필요한 두개의 실체

환경과 메모리(environment & memory)

프로그램 실행을 이해하는 데 필요한 두개의 실체

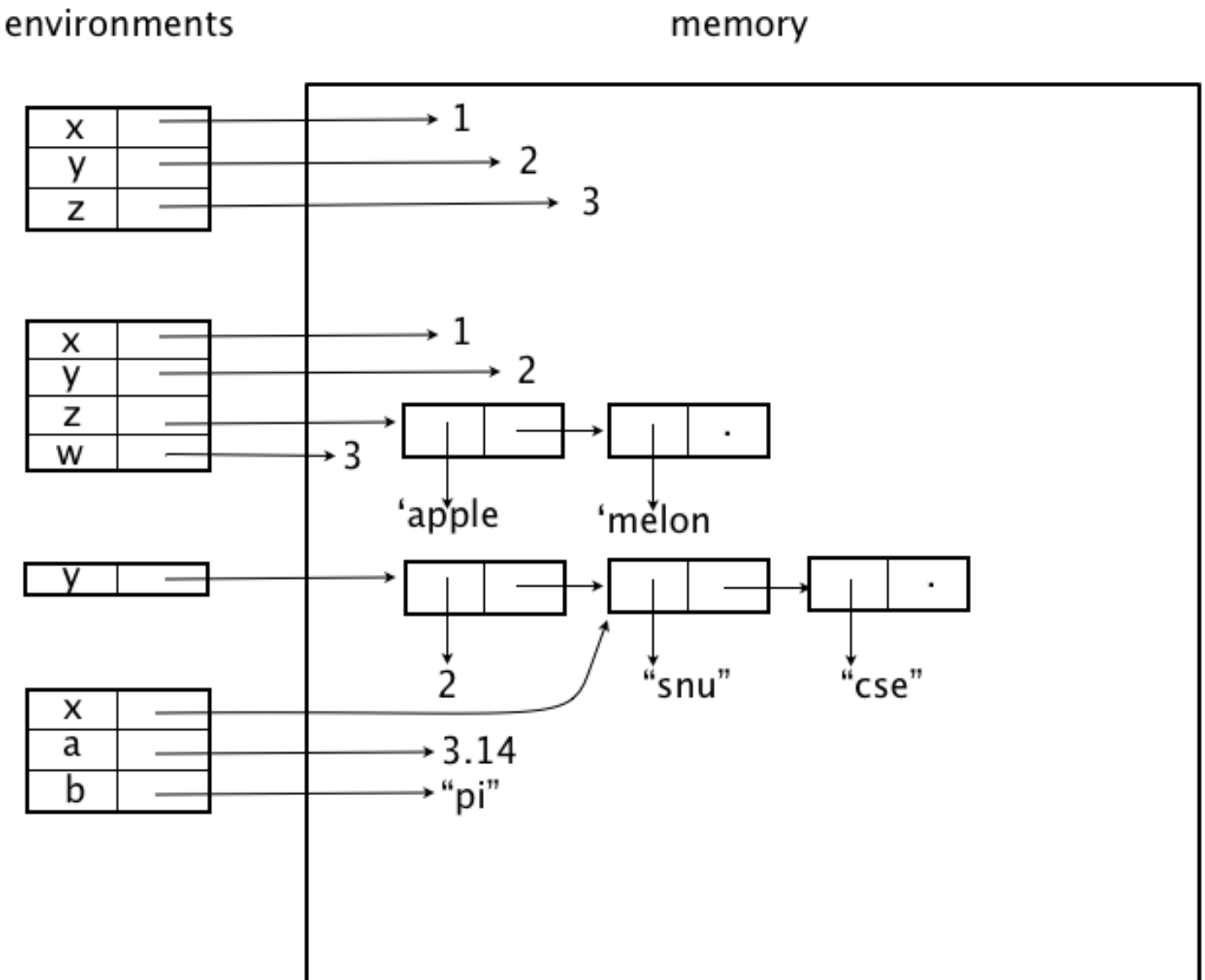
- 환경: 프로그램에서 정의된 이름들과 그 대상의 목록표
  - 환경은 여럿: 프로그램의 어디를 실행하냐에 따라 다른 환경이 사용됨
  - 유효범위에 따라 변화하는 이름의 대상을 파악하는 데 필요

환경과 메모리(environment & memory)

프로그램 실행을 이해하는 데 필요한 두개의 실체

- 환경: 프로그램에서 정의된 이름들과 그 대상의 목록표
  - 환경은 여럿: 프로그램의 어디를 실행하냐에 따라 다른 환경이 사용됨
  - 유효범위에 따라 변화하는 이름의 대상을 파악하는 데 필요
- 메모리: 이름이 지칭하는 그 대상(값이나 물건)을 구현하는 공간
  - 메모리는 하나: 메모리는 프로그램 시작때부터 끝날때까지 하나
  - 실행중에 변화하는 물건을 이해하는 데 필요. 구현된 공간에서 물건들이 변화.

환경과 메모리(environment & memory)

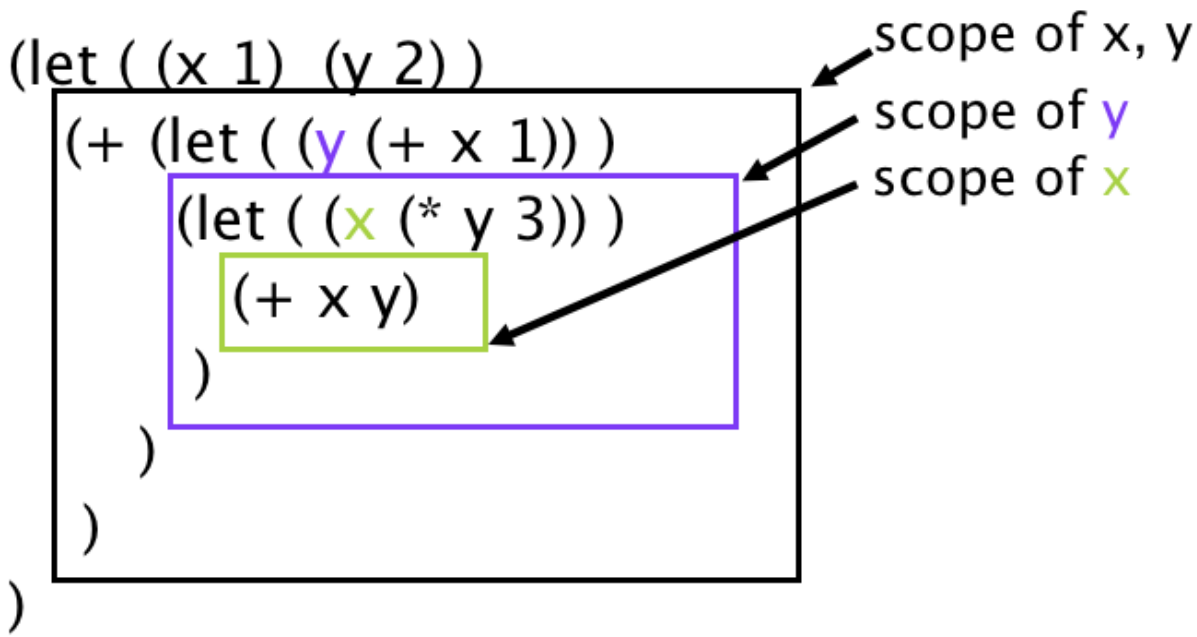


환경(environment)

- 환경은 이름들이 무엇을 지칭하는 지를 알려주는 테이블
  - 이름과 그 대상의 쌍(binding)들의 테이블
- 모든 프로그램식의 실행은 주어진 환경아래에서 진행된다
- 환경은 프로그램식을 바라보는 안경
  - `(+ x y)`의 실행결과는?
  - 다른 환경에서 다른 실행결과를 냄

환경(environment) 관리

- 환경 만들기: 이름이 지어지면
- 환경 참조하기: 이름이 나타나면
- 환경 폐기하기: 유효범위가 끝나면



새로운 환경이 도입되는 경우

이름짓는 경우(binding, declaration, definition)

- 식에서 이름짓기

E	$::=$	$\cdots$	예전것들
	$\mid$	$(\texttt{let}\ ((x\ E)^+)\ E)$	x의 정의
	$\mid$	$(\texttt{letrec}\ ((x\ E)^+)\ E)$	x의 재귀정의
	$\mid$	$(E\ E)$	함수호출시 함수인자가

- 프로그램에서 이름짓기

P	$::=$	E	계산식
	$\mid$	$(\texttt{define}\ x\ E)^*\ E$	이름정의 후 계산식

메모리의 물건을 변화시키는 경우

변화 시키는 명령문(mutation, imperative operations)

- Scheme: `set!`, `set-car!`, `set-cdr!`
- OCaml: 지정문(:=)
- Java, C, C++등: 모든 지정문(=)

함수와 환경

함수 = 함수 텍스트 정의와 함수가 정의될 때의 환경

함수와 환경

함수 = 함수 텍스트 정의와 함수가 정의될 때의 환경

```
(let ((y 1))
  (let ((udd (lambda (x) (+ x y))))
    (let ((y 10))
      (udd 8))))
```

환경모델 이해하기

환경모델 이해하기

```
(define x 1)
```

환경모델 이해하기

```
(define x 1)
(set! x (+ x 1))
```

환경모델 이해하기

```
(define x 1)
(set! x (+ x 1))
(* (let ((x (+ x 2)))
     (+ x 3))
   x)
```

환경모델 이해하기

```
(define x 1)
(set! x (+ x 1))
(* (let ((x (+ x 2)))
     (+ x 3))
   x)
(define f (lambda (n) (+ n x)))
```

환경모델 이해하기

```
(define x 1)
(set! x (+ x 1))
(* (let ((x (+ x 2)))
     (+ x 3))
   x)
(define f (lambda (n) (+ n x)))
(f 10)
```

환경모델 이해하기

```
(define x 1)
(set! x (+ x 1))
(* (let ((x (+ x 2)))
     (+ x 3))
   x)
(define f (lambda (n) (+ n x)))
(f 10)
(let ((x 100))
  (f 10))
```

환경모델 이해하기

```
(define (make-counter n)
  (lambda ()
    (begin (set! n (+ n 1)) n)
  ))
(define tic1 (make-counter 0))
```

환경모델 이해하기

```
(define (make-counter n)
  (lambda ()
    (begin (set! n (+ n 1)) n)
  ))
(define tic1 (make-counter 0))
(tic1)
(tic1)
```

환경모델 이해하기

```
(define (make-counter n)
  (lambda ()
    (begin (set! n (+ n 1)) n)
  ))
(define tic1 (make-counter 0))
(tic1)
(tic1)
(define tic2 (make-counter 0))
```

환경모델 이해하기

```
(define (make-counter n)
  (lambda ()
    (begin (set! n (+ n 1)) n)
  ))
(define tic1 (make-counter 0))
(tic1)
(tic1)
(define tic2 (make-counter 0))
(tic2)
```

환경모델 이해하기

```
(define (make-counter n)
  (lambda ()
    (begin (set! n (+ n 1)) n)
  ))
(define tic1 (make-counter 0))
(tic1)
(tic1)
(define tic2 (make-counter 0))
(tic2)
(tic1)
```

환경모델 이해하기

```
(define (make-withraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "insufficient fund"
    )))

(define withdraw (make-withdraw 100))
(withdraw 10)
(withdraw 20)
```

환경모델로 이해하기

```
(define empty (cons () ()))
(define (push s x)
  (let ((cell (cons x ())))
    (begin (set-cdr! cell (cdr s))
           (set-cdr! s cell))
```

환경모델로 이해하기

```
(define empty (cons () ()))
(define (push s x)
  (let ((cell (cons x ())))
    (begin (set-cdr! cell (cdr s))
           (set-cdr! s cell))
 (define stk empty)
(push stk 1)
(push stk 2)
```