Homework 2
CS420/KAIST Spring 1999
이 광근
Due: April 1, 24:00

1999년 3월 26일

Exercise 1 "귀납적(inductively)으로 생각하세요."
다음의 프로그램을 실행시키는 함수(interpreter) `intp`를 만들어 봅시다.

```
intp:  stm * store -> store
```

실행할 프로그램은 `stm` 타입의 값입니다.

```
type id = string
datatype binop = PLUS | TIMES
datatype stm = SET of id * exp
             | PRINT of exp list
             | SEQ of stm list
             | IF of exp * stm * stm
             | WHILE of exp * stm
     and exp = VAR of id
             | CONST of int
             | BINOP of exp * binop * exp
```

프로그램의 의미는 다음과 같습니다. 모든 명령문(stm)은 메모리를 변화시키거나 출력을 합니다. `SEQ`는 리스트에 있는 명령문들을 차례로(왼쪽에서 오른쪽으로) 수행합니다. `SET("x",e)`은 식 `e`의 값을 `x`라는 변수에 저장합니다. `PRINT`는 리스트에 있는 식들을 차례로 프린트합니다. 이때, 출력하는 값사이에 콤마를 찍습니다. `IF(e, s, s')`은 `e`이 값이 0 이 아니면 `s`를 수행하고 0 이면 `s'`을 수행합니다. `WHILE(e,s)`은 `e`를 수행해서 0 이 아니면 `s`를 수행하고 다시 `WHILE(e,s)`를 수행합니다. 만일 `e`가 0이면 반복을 멈춥니다.

모든 식들은 값을 계산합니다. `VAR "x"` 은 변수 `x`의 값입니다. `BINOP`는 두 식의 값을 가지고 주어진 binary operator를 적용한 값을 내놓습니다. □

Exercise 2 "큐는 딸랑 하나의 리스트일 필요는 없어요."

큐에 넣고(enqueue) 빼는(dequeue) 작업이 거의 한 스텝만으로 실현될 수 있지요. Types of each operations are:

```
empty:  queue
enqueue:  queue * elmt -> queue
dequeue:  queue -> elmt * queue
```

A queue $[a_1, \cdots, a_m, b_1, \cdots, b_n]$ (b_n is the head) is represented as a pair of two lists L and R:

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

After adding an element `x`, new queue becomes

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

After deleting an element, new queue becomes

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

When deleting, we sometimes have to reverse the L list and let it be the R list. Empty queue is $([], [])$.

Implement the funtor `QuickQueue`:

```
signature QUEUE =
  sig
    type queue and elmt
    val empty: queue
    val enqueue: queue * elmt -> queue
    val dequeue: queue -> elmt * queue
  end
functor QuickQueue(...): QUEUE =
  struct
    type queue = ...
    type elmt = ...
    ...
  end
```

Usage:

```
(* int list queue *)
structure ILQueue = QuickQueue(struct type elmt = int list end)
(* string pair queue *)
structure SPQueue = QuickQueue(struct type elmt = string * string end)
```

□

Exercise 3 "창살 무늬를 디자인하는 인간문화재 A 선생을 위해서."

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 정사 격형을 만들고, 되었다 싶으면 디자인된 정 사각형들을 반복해서 창살에 짜넣는 방법을 취하겠지요.[1]

그 작업 과정을 functor로 표현할 수 있는데, 다음과 같습니다. 코드를 잘 읽어 보고, 코드를 완성해 보세요.

```
datatype design = TURTLE | WAVE | DRAGON | KAIST (* four design patterns *)
datatype orientation = NW | NE | SE | SW
datatype box = BOX of orientation * design | GLUED of box * box * box * box
signature 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 *)
  end
functor BasicFrame(Design: sig val design: design end): FRAME =
  struct
    exception NON_BASIC_BOX
    val box = BOX (NW, Design.design)    (* a box is defined *)
    fun rotate = ...
    fun pp (BOX(NW,x)) center = ()
      | pp (BOX(NE,x)) center = ()
      | pp (BOX(SE,x)) center = ()
      | pp (BOX(SW,x)) center = ()
      | pp _ _ = raise NON_BASIC_BOX
  end
functor Rotate(Box: FRAME): FRAME =
  struct
   val box = ...
   val rotate = ...
   val pp = ...
  end
functor Glue(structure Nw: FRAME
             structure Ne: FRAME
             structure Se: FRAME
             structure Sw: FRAME): FRAME =
  struct
   ...
  end
```

위의 코드를 이용한 디자인과정의 한 예는 수업시간에!

□

[1]순전히 저의 추측인데, 실제로 이렇게 하지않을까요?

Exercise 4 "Dynamic-Scheme? Easy!"

Dynamic-Scheme 프로그램을 실행시키는 함수(interpreter) `dscheme`를 만들어 봅시다:

```
dscheme:  expr * env -> value
```

Dynamic-Scheme은 이름의 유효범위(scope)가 실행중에 결정되는 Scheme 입니다. 실행할 프로그램은 `expr` 타입의 값이고, `env`는 이름과 값의 짝을 가지고 있는 표라고 생각하면 됩니다.

```
type id = string
datatype expr = VALUE of value
              | VAR of string
              | IF of expr * expr * expr
              | BEGIN of expr list
              | APPLY of expr * expr list
     and value = INT of int
               | LIST of value list
               | CAR | CDR | CONS | NIL
               | IS_NUMBER | IS_SYMBOL | IS_LIST | IS_NULL
               | PRINT | PLUS | MINUS | MULT | DIV | EQUAL | LT
               | LAMBDA of string list * expr
```

위의 언어의 의미는 여러분이 알고 있는 Scheme의 경우를 가지고 유추하면 됩니다. 힌트는:

```
         (car 1 nil)  is  APPLY(VALUE CAR, [VALUE(INT 1), VALUE NIL])
(lambda (x) (+ x 1))  is  VALUE(LAMBDA([``x''],
                                       APPLY(VALUE PLUS,
                                             [VAR ``x'', VALUE(INT 1)]
                                            )
                                      ))
```

함수가 적용될 때 이름들의 유효범위가 실행전에 결정되는 것이 아니고 실행중에 결정되므로, 쉽게 구현할 수 있습니다.[2]

Dynamic-Scheme의 궁금점은 수없시간에!

□

[2] 초창기 Lisp이 이러했습니다.