Homework 4

SNU 4190.310, 2006 가을

이광근　박대준

Due: 10/30(Mon), 24:00

Exercise 1 (30pts) "설탕"

메모리 주소가 자유롭게 사용될 수 있는 K−에서는 주소전달 호출(call-by-reference)식

```
call f<x>
```

은 설탕(syntactic sugar)이 된다. 위의 설탕을

```
call f'(&x)
```

으로 녹이고, `f'`은 `f`의 정의에서 함수의 인자이름이 쓰이는 곳을 적절히 바꾸어 주면 될 것이다.

주어진 K− 프로그램 S를 받아서 위의 설탕을 녹인 프로그램 S'을 만드는 함수

```
dissolveCbr : program -> program
```

를 정의하라. `dissolveCbr`(S)는 S에서 설탕을 녹였지만 같은 일을 하는 프로그램이어야 한다:

$$\texttt{K.run (K.emptyMemory, K.emptyEnv, } S\texttt{)}$$
$$\equiv \texttt{ K.run (K.emptyMemory, K.emptyEnv, dissolveCbr } S\texttt{)}.$$

`program`은 숙제3의 `KMINUS` 꼴에 정의된 대로이다. □

Exercise 2 (30pts) "SM5"

K--는 K-에서 `for`문과 `while`문, 그리고 주소로 호출하기(call-by-reference)가 없는 언어이다.

SM5는 가상의 기계이다. "SM"은 "Stack Machine"을 뜻하고, "5"는 그 기계의 부품이 5개이기 때문이다:

$$(S, M, E, C, K)$$

S는 스택, M은 메모리, E는 환경, C는 명령어, K는 남은 할 일("continuation"이라고 부름)을 뜻하고 다음 집합들의 원소이다:

$$\begin{array}{rcrcl}
S & \in & \mathit{Stack} & = & \mathit{Svalue}\ \mathit{list} \\
M & \in & \mathit{Memory} & = & \mathit{Loc} \to \mathit{Value} \\
E & \in & \mathit{Environment} & = & (\mathit{Var} \times (\mathit{Loc} + \mathit{Proc}))\ \mathit{list} \\
C & \in & \mathit{Command} & = & \mathit{Cmd}\ \mathit{list} \\
K & \in & \mathit{Continuation} & = & (\mathit{Command} \times \mathit{Environment})\ \mathit{list}
\end{array}$$

$$\begin{array}{rcrcl}
v & \in & \mathit{Value} & = & \mathit{Integer} + \mathit{Bool} + \mathit{Unit} + \mathit{Record} + \mathit{Loc} \\
x & \in & \mathit{Var} & & \\
\langle b, o \rangle, l & \in & \mathit{Loc} & = & \mathit{Base} \times \mathit{Offset} \\
 & & \mathit{Offset} & = & \mathit{Integer} \\
z & \in & \mathit{Integer} & & \\
b & \in & \mathit{Bool} & & \\
r & \in & \mathit{Record} & = & (\mathit{Var} \times \mathit{Loc})\ \mathit{list} \\
w & \in & \mathit{Svalue} & = & \mathit{Value} + \mathit{Proc} + (\mathit{Var} \times \mathit{Loc}) \qquad \text{(* stackable values *)} \\
p & \in & \mathit{Proc} & = & \mathit{Var} \times \mathit{Command} \times \mathit{Environment} \\
 & & \mathit{Cmd} & = & \{\texttt{push}\ v\texttt{, push}\ x\texttt{, push(}x\texttt{,}C\texttt{),} \\
 & & & & \quad \texttt{pop, store, load, jtr(}C\texttt{,}C\texttt{),} \\
 & & & & \quad \texttt{malloc, box}\ z\texttt{, unbox}\ x\texttt{, bind}\ x\texttt{, unbind, get, put, call,} \\
 & & & & \quad \texttt{add, sub, mul, div, eq, less, not}\}
\end{array}$$

기계의 작동은 다음과 같이 기계의 상태가 변화하는 과정으로 정의할 수 있다:

$$(S, M, E, C, K) \Rightarrow (S', M', E', C', K')$$

언제 어떻게 위의 기계작동의 한 스텝($\Rightarrow$)이 일어나는 지는 다음과 같다:

$$
\begin{array}{lllll}
 & (S, & M, \ E, & \texttt{push}\ v :: C, \ K) & \\
\Rightarrow & (v :: S, & M, \ E, & C, \ K) & \\
 & & & & \\
 & (S, & M, \ E, & \texttt{push}\ x :: C, \ K) & \\
\Rightarrow & (w :: S, & M, \ E, & C, \ K) & \text{if } (x, w) \text{ is the first such entry in } E \\
 & & & & \\
 & (S, & M, \ E, & \texttt{push}\ (x, C') :: C, \ K) & \\
\Rightarrow & ((x, C', E) :: S, & M, \ E, & C, \ K) & \\
 & & & & \\
 & (w :: S, & M, \ E, & \texttt{pop} :: C, \ K) & \\
\Rightarrow & (S, & M, \ E, & C, \ K) & \\
 & & & & \\
 & (l :: v :: S, & M, \ E, & \texttt{store} :: C, \ K) & \\
\Rightarrow & (S, & M[v/l], \ E, & C, \ K) & \\
 & & & & \\
 & (l :: S, & M, \ E, & \texttt{load} :: C, \ K) & \\
\Rightarrow & (M(l) :: S, & M, \ E, & C, \ K) &
\end{array}
$$

$$
\begin{array}{lllllll}
 & (\mathit{true} :: S, & M, & E, & \texttt{jtr}(C_1, C_2) :: C, & K) & \\
\Rightarrow & (S, & M, & E, & C_1 :: C, & K) & \\
 & & & & & & \\
 & (\mathit{false} :: S, & M, & E, & \texttt{jtr}(C_1, C_2) :: C, & K) & \\
\Rightarrow & (S, & M, & E, & C_2 :: C, & K) & \\
 & & & & & & \\
 & (S, & M, & E, & \texttt{malloc} :: C, & K) & \\
\Rightarrow & (\langle b, 0\rangle :: S, & M, & E, & C, & K) & \text{new } b \\
 & & & & & & \\
 & (w_1 :: \cdots :: w_z :: S, & M, & E, & \texttt{box}\, z :: C, & K) & \\
\Rightarrow & ([w_1, \cdots, w_z] :: S, & M, & E, & C, & K) & \\
 & & & & & & \\
 & ([w_1, \cdots, w_z] :: S, & M, & E, & \texttt{unbox}\, x :: C, & K) & \\
\Rightarrow & (v_k :: S, & M, & E, & C, & K) & w_k = (x, v_k), 1 \leq k \leq z \\
 & & & & & & \\
 & (w :: S, & M, & E, & \texttt{bind}\, x :: C, & K) & \\
\Rightarrow & (S, & M, & (x, w) :: E, & C, & K) & \\
 & & & & & & \\
 & (S, & M, & (x, w) :: E, & \texttt{unbind} :: C, & K) & \\
\Rightarrow & ((x, w) :: S, & M, & E, & C, & K) & \\
 & & & & & & \\
 & (l :: v :: (x, C', E') :: S, & M, & E, & \texttt{call} :: C, & K) & \\
\Rightarrow & (S, & M[v/l], & (x, l) :: E', & C', & (C, E) :: K) & \\
 & & & & & & \\
 & (S, & M, & E, & \mathit{empty}, & (C, E') :: K) & \\
\Rightarrow & (S, & M, & E', & C, & K) &
\end{array}
$$

$$
\begin{array}{lllllll}
 & (S, & M, & E, & \texttt{get} :: C, & K) & \\
\Rightarrow & (z :: S, & M, & E, & C, & K) & \text{read } z \text{ from outside} \\
 & & & & & & \\
 & (z :: S, & M, & E, & \texttt{put} :: C, & K) & \\
\Rightarrow & (S, & M, & E, & C, & K) & \text{print } z \text{ and newline} \\
 & & & & & & \\
 & (v_2 :: v_1 :: S, & M, & E, & \texttt{add} :: C, & K) & \\
\Rightarrow & (\mathit{plus}(v_1, v_2) :: S, & M, & E, & C, & K) & \\
 & & & & & & \\
 & (v_2 :: v_1 :: S, & M, & E, & \texttt{sub} :: C, & K) & \\
\Rightarrow & (\mathit{minus}(v_1, v_2) :: S, & M, & E, & C, & K) & \\
 & & & & & & \\
 & (z_2 :: z_1 :: S, & M, & E, & \texttt{mul} :: C, & K) & \\
\Rightarrow & ((z_1 * z_2) :: S, & M, & E, & C, & K) & \text{similar for } \texttt{div} \\
 & & & & & & \\
 & (v_2 :: v_1 :: S, & M, & E, & \texttt{eq} :: C, & K) & \\
\Rightarrow & (\mathit{equal}(v_1, v_2) :: S, & M, & E, & C, & K) & \\
 & & & & & & \\
 & (v_2 :: v_1 :: S, & M, & E, & \texttt{less} :: C, & K) & \\
\Rightarrow & (\mathit{less}(v_1, v_2) :: S, & M, & E, & C, & K) & \\
 & & & & & & \\
 & (b :: S, & M, & E, & \texttt{not} :: C, & K) & \\
\Rightarrow & (\neg b :: S, & M, & E, & C, & K) &
\end{array}
$$

위에서 사용된 *plus*, *minus*, *equal*, *less*는 K-의 의미 정의 문서에 정의된 것과 동일하다.

SM5의 프로그램 C를 실행한다는 것은, C만 가지고 있는 빈 기계상태를 위에서 정의한 방식으로 변환해 간다는 뜻이다:

$$(\mathit{empty}, \mathit{empty}, \mathit{empty}, C, \mathit{empty}) \Rightarrow \cdots \Rightarrow \cdots$$

예를들어,

$$\texttt{push 1} :: \texttt{push 2} :: \texttt{add} :: \texttt{put}$$

는 K-- 프로그램 `write 1+2`과 같은 일을 하게 된다.

여러분이 할 것은, 잘 돌아가는 K-- 프로그램을 입력으로 받아서 같은 일

을 하는 SM5 프로그램으로 변환하는 함수

```
trans: K.program -> Sm5.command
```

를 작성하는 것이다.

`trans`가 제대로 정의되었는지는, K-- 프로그램 E에 대해서, `K.run(`E`)`와 `Sm5.run(trans(`E`))`을 실행해서 확인할 수 있을 것이다.

모듈 `Sm5`는 제공된다. 모듈 `K`와 K-의 파서는 이전에 제공된 것을 사용한다.

□

Exercise 3 (40pts) "재귀호출의 비용"

학생 왈: "저는 프로그램 짤 때 재귀함수로 짜지 않습니다. 함수 호출은 일반적으로 시간과 메모리를 너무 많이 잡아먹는데, 재귀 호출은 그게 너무 많이 반복되잖습니까. 그래서 피하고 있습니다."

이번 숙제에서는, 위 학생의 이야기가 무슨 뜻인지, SM5 프로그램의 실행과정을 관찰하면서 알아보고, 재귀함수호출이 가지는 비용을 줄이는 개선책을 찾아나서보자.

- 함수호출이 다른 명령에 비해 비용이 많이드는 이유:

  SM5의 실행과정을 보면 그 이유가 드러난다. 함수 `call` 명령어가 실행될 때 마다, K (continuation) 파트에 함수가 끝나고 계속 진행할 내용을 쌓아갔다:

$$\begin{array}{rlllll} & (l :: v :: (x, C', E') :: S, & M, & E, & \texttt{call} :: C, & K) \\ \Rightarrow & (S, & M[v/l], & (x, l) :: E', & C', & \underline{(C, E) :: K}) \end{array}$$

  그리고 K에 기록된 "계속할 일"은 함수가 끝나면 복원해 주었다:

$$\begin{array}{rllll} & (S, & M, & E, & \mathit{empty}, & (C, E') :: K) \\ \Rightarrow & (S, & M, & E', & C, & K) \end{array}$$

  즉, 함수의 호출은 호출이 끝나고 할 일을 K에 넣다 뺐다하는 과정이 필요하고, 함수 호출이 재귀적으로 일어나면 그러한 것들이 K에 계속해서 쌓이기 때문에 메모리나 자원을 많이 쓰는 셈인 것이리라. 예를 들어, factorial 함수같은 경우 "`n * fac(n-1)`"이라는 재귀호출 부분 때문에 K에 쌓이는 것은 변수 `n`의 크기만큼이 될 것이다.

- 언제 어떻게 K에 넣다 뺐다 하는 것을 줄일 수 있을까?

특이한 재귀함수의 호출인 경우 줄일 수 있다. 재귀함수 중에서 재귀호출이 끝나고 나서 앞으로 할 일이란 것은 다시 return하는 것 밖에는 없는 꼴이 있다. 예를 들어,

```
procedure f(x) = if x<0 then 1 else f(x-1)
```

에서, 재귀호출 `f(x-1)`이 끝난 후에 할 일은 아무것도 없다. 재귀호출 한 함수를 끝내는 것 이외에는. 이러한 재귀호출을 끝재귀호출(*tail-recursive call*)이라고 한다. 맨 마지막(tail)에 하는 일이 재귀호출밖에는 없다는 뜻이다. (한편 factorial은 끝재귀가 아니다. 재귀호출을 하고 나서 할일이 있다: 재귀호출 결과(`fac(n-1)`)에 `n`을 곱하는 일.)

이정도로 힌트는 마치기로 하고, 여러분이 할 일은 SM5에 최소의 명령어를 첨가해서 SM5x를 만들고, K--의 재귀호출을 SM5x로 번역할 때, 첨가한 새로운 SM5x 명령어를 이용해서 위에서 언급한 함수호출비용(K에 뭔가가 자꾸 쌓이는 현상)이 줄어들도록 하는 것이다.

제출할 것은 모듈 `Sm5x`과 프로그램과 `trans`함수이다. 코드에는 첨가된 명령어들의 정의를 코멘트로 잘 설명하도록 한다.

□