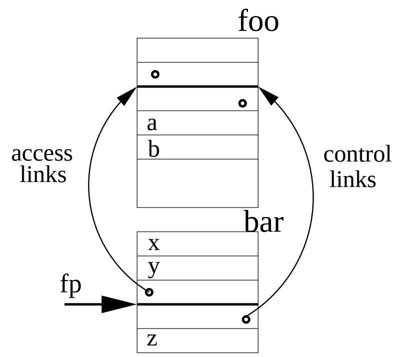
CS314 Spring 2014

Assignment 6

Due Friday, April 4, **before** class
Sample Solution

Problem 1 – Parameter Passing




```
program foo()
 {    a, b integer;
      procedure bar(integer x, integer y)
      {  z: integer;
                        <----- /* 0 */
         z = 5;                /* 1 */
         x = x + y + z;        /* 2 */
         y = 1;                /* 3 */
      }
      // statement body of foo
      a = 1;
      b = 2;
      call bar(a, b);
      print a, b; }
```

Use the RISC machine instructions `LOAD, STORE, LOADI, ADD` as used in the non-local data access example (lecture 13, pages 11 and 12) to show the code that needs to be generated for the body of procedure bar (statements (/*1*/ through /*3*/). Assume that

1. Register $r0$ contains the frame pointer (fp) value.
2. Formal parameter `x` is **call-by-reference**, and formal parameter `y` is **call-by-value**. Assume that bar's parameters `x` and `y` have been correctly initialized as part of the procedure call of bar.

3. Use the stack frame layout as shown above. The figure shows the runtime stack when the program execution reaches program point /*0*/ in procedure bar.

What values for `a` and `b` does the program print?

```
LOADI #5 => r1
LOADI #4 => r2      // offset of local variable z
ADD r0, r2 => r3    // address of z
STORE r1 => r3      // z = 5

LOADI #-12 => r4    // offset of call-by-reference parameter x
ADD r0, r4 => r5    // address of x
LOAD r5 => r6       // content of x is address of actual parameter
LOAD r6 => r7

LOADI #-8 => r8    // offset of call-by-value parameter y
ADD r0, r8 => r9   // address of y
LOAD r9 => r10

ADD r7, r10 => r11  // x + y

LOADI #4 => r12       // offset of local variable z
ADD r0, r12 => r13    // address of z
LOAD r13 => r14

ADD r11, r14 => r15  // x + y + z

LOADI #-12 => r16   // offset of call-by-reference parameter x
ADD r0, r16 => r17  // address of x
LOAD r17 => r18     // content of x is address of actual parameter
STORE r15 => r18    // x = x + y + z

LOADI #1 => 19
LOADI #-8 => r20     // offset of call-by-value parameter y
ADD r0, r20 => r21   // address of y
STORE r19 => r21     // y = 1
```

print a, b : 8, 2

Problem 2 – Parameter Passing

Assume that you don't know what particular parameter passing style a programming language is using. In order to find out, you are asked to write a short test program that will print a different output depending on whether a *call-by-value*, *call-by-reference*, or *call-by-value-result* parameter passing style is used. Your test program must have the following form:

```
program main()
 {    a integer;
      procedure foo(integer x)
      {
          // statement body of foo
      }

      // statement body of main
      a = 1;
      call foo(a);
      print a;
 }
```

The body of procedure *foo* must only contain assignment statements. For instance, you are not allowed to add any new variable declarations.

1. Write the body of procedure foo such that `print a` in the `main` program will print different values for the different parameter passing styles.

```
program main()
 {    a integer;
      procedure foo(integer x)
      {
          // statement body of foo
       S1:   x := x + 1;
       S2:   x := a + 1;
      }

      // statement body of main
      a = 1;
      call foo(a);
      print a;
 }
```

2. Give the output of your test program and explain why your solution works.

   call-by-value: prints 1 – both assignments modify memory location "x" within the frame of foo; value of "a" is copied into "x" when foo is called.

call-by-reference: prints 3 – both assignments change the value of "a"

call-by-value-result: prints 2 – statement S1 modifies memory location "x" within the frame of foo; statement S2 overwrites the same memory location with the same value, namely 2. At the write-back step, "a" is assigned 2.

Problem 3 – Scheme

Write Scheme programs that generate the following lists as output using only `cons` as the list building operator:

1. ```
   ;; '(a b (c d (e f (g))))

   (cons 'a
      (cons 'b
         (cons (cons 'c
            (cons 'd
               (cons (cons 'e
                  (cons 'f
                    (cons (cons 'g '()) '())))
             '())))
         '())))
   ```

2. ```
   ;; '(((((a) b c) d) (e f)) g)

   (cons
     (cons
       (cons
         (cons
           (cons 'a '()) (cons 'b (cons 'c '())))
         (cons 'd '()))
       (cons
         (cons 'e (cons 'f '())) '()))
      (cons 'g '()))
   ```

3. ```
   ;; '(a + 3) such that ((cadr '(a + 3)) 3 5)
   ;; evaluates to  8
   (cons 'a (cons + (cons 3 '())))
   ```
   ```

Problem 4 – Scheme

Write the following functions on lists in Scheme. The semantics of the functions is decribed through examples.

1.
```
(define flatten
  (lambda (l)
    (cond
      ((null? l) '())
      ((list? (car l)) (append (flatten (car l)) (flatten (cdr l))))
      (else (cons (car l) (flatten (cdr l)))))))

 ;; (flatten '(a ((b) (c d) (((e)))))) --> '(a b c d e)
```

2.
```
(define rev
   (lambda (l)
     (cond
       ((null? l) '())
       ((list? (car l)) (append (rev (cdr l)) (cons (rev (car l)) '())))
       (else (append (rev (cdr l)) (cons (car l) '()))))))

 ;; (rev '(a((b)(c d)(((e))))) --> '((((e)))(d c)(b))a)
 ;; Note: Do not use the Scheme build-in function "reverse".
```

3.
```
(define double
   (lambda (l)
     (cond
       ((null? l) '())
       ((list? (car l)) (cons (double (car l)) (double (cdr l))))
       (else (cons (car l) (cons (car l) (double (cdr l))))))))

 ;; (double '(a((b)(c d)(((e))))) --> '(a a((b b)(c c d d)(((e e)))))
```

4.
```
(define delete
   (lambda (atom l)
     (cond
       ((null? l) '())
       ((list? (car l)) (cons (delete atom (car l))  (delete atom (cdr l))))
       (else (if (eq? atom (car l))
          (delete atom (cdr l))
          (cons (car l) (delete atom (cdr l))))))))

 ;; (delete 'c '(a((b)(c d)(((e))))) --> (a((b)(d)(((e)))))
 ;; (delete 'f '(a((b)(c d)(((e))))) --> (a((b)(c d)(((e)))))
```