PL Midterm Solution

1 [Chap01, pp36~37]

Pro – Portable

Con – Inefficiency (or, Inflexible)

2 [Chap01, p7]

3 [Chap03, p12]

4 [Chap01, pp47,53]

5 Let n be the number of elements in the list

An imperative-style procedure simply modifies the list by erasing the last element. So, it takes $O(n)$ time and $O(1)$ space.

A functional-style procedure has to leave the list unchanged and copy the first $n-1$ elements to the resulting list. So, it takes $O(n)$ time and $O(n)$ space.

6 `$a=(1,2,3,4,5,6);`

Set the scalar variable $a to 6

`@a=(1,2,3,4,5,6);`

`@a` becomes an array containing six integers 1, 2, 3, 4, 5, 6.

`%a=(1,2,3,4,5,6);`

`%a` becomes a hash table containing three (key,value) pairs, namely, (1,2), (3,4), and (5,6).

7 a) [HW#2, Problem 1a]

?: is right-associative

, is left-associative

b) [HW#2, Problem 1b]

In a = b? c: d, ?: has a higher precedence than =.

But, in a? b: c = d, = has a higher precedence than ?:

c) *conditional-expression:*

logical-or-expression

~~*logical-or-expression* ? *expression* : *assignment-expression*~~

logical-or-expression ? *expression* : *conditional-expression*

assignment-expression:

conditional-expression

~~*logical-or-expression assignment-operator assignment-expression*~~

conditional-expression assignment-operator assignment-expression

expression:

assignment-expression

expression , *assignment-expression*

7 c) (Cont'd)

[The following explanations are optional.]

With this grammar, ?: always has a higher precedence than =.

In particular,

a = b? c: d means a = <u>b? c: d</u>

and

a? b: c = d means <u>a? b: c</u> = d.

Let's look at the derivations in detail.

First, a = b? c: d means a = <u>b? c: d</u>, because

expression

$\Rightarrow^*$ *conditional-expression assignment-operator assignment-expression*

$\Rightarrow^*$ a = *assignment-expression*

$\Rightarrow$ a = *conditional-expression*

$\Rightarrow^*$ a = b? c: d

On the other hand, a = b? c: d doesn't mean <u>a = b</u>? c: d, because

expression

$\Rightarrow^*$ *logical-or-expression*? *expression* : *conditional-expression*

$\Rightarrow^*$ *logical-or-expression*? c: d

Now, a = b can't be derived from *logical-or-expression* unless it is parenthesized.

Next, a? b: c = d means <u>a? b: c</u> = d, because

expression

$\Rightarrow^*$ *conditional-expression assignment-operator assignment-expression*

$\Rightarrow^*$ *conditional-expression* = d

$\Rightarrow^*$ a? b: c = d

On the other hand, a? b: c = d doesn't mean a? b: <u>c = d</u>, because

expression

$\Rightarrow^*$ *logical-or-expression*? *expression* : *conditional-expression*

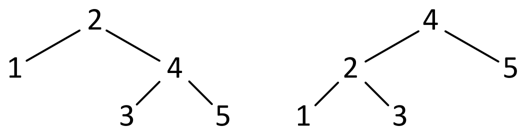
$\Rightarrow^*$ a? b: *conditional-expression*

Now, c = d can't be derived from *conditional-expression* unless it is parenthesized.

Moreover, the associativities of ?: and = remain unchanged. (Check it!)

8 a) First, recall (Chap03, p10) that the language P of all nested balanced parentheses is context-free, but not regular.
Next, observe that the language T of binary trees contains the sentences
((… ((0 0 0) 0 0) …) 0 0)
By removing the 0's, we obtain nested balanced parentheses:
((… (()) …))
Thus, P is essentially a sublanguage of T.
Since P isn't regular, it follows that T is not regular, too.

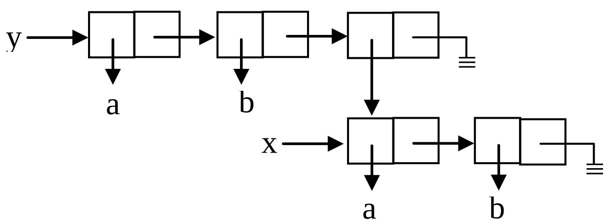
b) For example, 1 (2) 3 (4) 5 has two parse trees that corresponds to the following two binary trees, respectively.



c)

$T_1 \rightarrow (T_2\ D\ T_3)$	*predicate:* T_2.max < D.value <= T_3.min
	T_1.max = T_3.max
	T_1.min = T_2.min
$T \rightarrow D$	T.min = T.max = D.value
$D \rightarrow 0$	D.value = 0
$D \rightarrow 1$	D.value = 1
……	
$D \rightarrow 9$	D.value = 9

9 a)



b) 1 (a b (a b))
2 (cons (a b) a b)

10 a) compile (λx.λy.x)
= abstract x (compile (λy.x))
= abstract x (abstract y (compile x))
= abstract x (abstract y x)
= abstract x (K x)
= S (abstract x K) (abstract x x)
= S (K K) I

10 b) <u>S (S (K +) I) I (* 2 3)</u>

= <u>S (K +) I (* 2 3)</u> (I (* 2 3))

= <u>K + (* 2 3)</u> (I (* 2 3)) (I (* 2 3))

= + <u>(I (* 2 3))</u> <u>(I (* 2 3))</u>

= + <u>(* 2 3)</u> <u>(* 2 3)</u>

= <u>+ 6 6</u>

= 12

11 a) [HW#3, Problem 5a]

b) [HW#3, Problem 5b]

12 a) (lambda (v) (c (* n (f v))))

where c is the continuation of the function call (f n).

b) [Scheme, p36]

Such a technique can be used to escape from deep recursion so as to skip the remaining computations.

For example, to compute the product of a list, say (2 3 0 4 5 6), we may simply return 0 on seeing a 0 in the list by abandoning the continuation of multiplying 0 by 3 and 2.

13 a)

```
! iterative version
function sum(a)
implicit none
integer :: sum
integer, dimension(:) :: a
sum=0
do i = 1,ubound(a,1)
    sum=sum+a(i)
end do
end function sum
```

13 a) **! recursive version**

```
recursive function sum(a) result(r)
implicit none
integer :: r
integer, dimension(:) :: a
if (ubound(a,1)==1) then
   r=a(1)
else
   r=a(1)+sum(a(2:ubound(a,1)));
end if
end function sum
```

b) **# iterative version**

```
sub sum
{
      my $r=0;
      for (@_) { $r +=    $_; }
      return $r;
}
```

 # recursive version

```
sub sum
{
      if (@_==0) { return 0; }
      else { my $x=shift; return $x+sum(@_); }
}
```

c) **; recursive version**

```
(define sum
      (lambda (xs) (if (null? Xs) 0 (+ (car xs) (sum (cdr xs))))))
```

; iterative version

```
(define sum
   (lambda (xs)
      (let ((r 0))
         (let loop ()
            (cond ((null? xs) r)
                  (else (set! r (+ (car xs) r)) (set! xs (cdr xs)) (loop))))))
```