プログラミング言語論・テスト問題用紙

('02 年 7 月 18 日・ 8:50 〜 10:20)

解答上、その他の注意事項

I. 問題は、問 I〜VII までである。

II. 解答用紙の右上の欄に学籍番号・名前を記入すること。

III. 解答欄を間違えないよう注意すること。

IV. 選択式でない問で解答欄がマス目になっている場合は、1 字に 1 マスを用いること。特に空白にも必ず 1 マスを用いること

V. 解答中の文字 (特に a と d) がはっきりと区別できるよう注意すること。

VI. ノート・プリント・参考書などは持ち込み可である。プリントは 1 冊（やむを得ない場合は 2 冊）にまとめること。

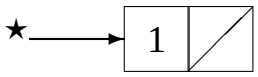
VII. テストの配点は 80 点である。合格はレポートの得点を加点して、100 点満点中 60 点以上とする。

このテストでの表記上の注意について:
(最初に必ず読んで下さい。)
Scheme では、(a$_1$ a$_2$ ...a$_n$) という括弧を使った式は、ユーザが入力するときは関数適用の意味に、処理系が出力するときはリストの意味になる。ユーザがリストを入力する時は「'」(クウォート記号) をつけて '(1 2 3 4) のように書く必要がある。これでは、計算の途中結果を示すなど時に不便なので、このテストではまぎらわしい時はリストを表す括弧はクウォート記号を使わずに大括弧 (square bracket、「[」と「]」) で表すことにする。
例: [1 2 3 4], [[1 2] [4 5]] など

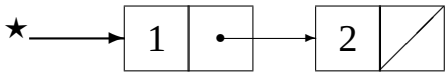
I. 箱矢印記法 (box-pointer notation) では、cons セルを 2 つのつながった箱 (box) で表す。左側の箱には car の内容が、右側の箱には cdr の内容が書かれる。

リストは、このような箱への矢印 (pointer) として表される。また、空リストは特別に斜線で表す。

例えば、car が 1 で cdr が空リストであるようなリストは、



[1 2] というリストは



の ⋆ 印のついている矢印で表される。

(1) [1 2 3] というリストと、

(2) [[1] [2] [3 4]] というリストを

箱矢印記法で表し、**その矢印 (各問ごとに 1 つのみ) に ⋆ 印をつけよ。**

II. 次のように関数 append を定義する。

```
(define (append xs ys)
   (if (null? xs)                                  ; この行の下線部を 式 1 とする
       ys                                          ; この行の下線部を 式 2 とする
       (cons (car xs) (append (cdr xs) ys))))      ; この行の下線部を 式 3 とする
```

例えば、(append [1 2] [4 5]) は次のように計算される。ここで下線部は、計算が行なわれる部分を示している。

```
(append [1 2] [4 5])
⇒ (cons 1 (append [2] [4 5]))          ; 式 1 が偽なので式 3 を選択する
⇒ (cons 1 (cons 2 (append [] [4 5])))  ; 式 1 が偽なので式 3 を選択する
⇒ (cons 1 (cons 2 [4 5]))              ; 式 1 が真なので式 2 を選択する
⇒ (cons 1 [2 4 5])                     ; cons の計算
⇒ [1 2 4 5]                            ; cons の計算
```

この例にならって、次のように定義された関数 zip

```
(define (zip xs xss)
   (if (or (null? xs) (null? xss))          ;  この行の下線部を 式 1 とする
       '()                                  ;  この行の下線部を 式 2 とする
       (cons (cons (car xs) (car xss))      ;  この 2 行の
             (zip (cdr xs) (cdr xss)))))    ;        下線部を 式 3 とする
```

について、(zip [1 4 7] [[2 3] [5 6]]) の評価（計算）の様子を書き下せ。リストを表す括弧には、上の append の例のように大括弧（square bracket）を使え。また、上の例の「cons の計算」もしくは「式★が真（または偽）なので式☆を選択」のようなコメントは解答には**必要ない**。

III. (take_upto n xs)（JavaScript の場合、take_upto(n, xs)）は**要素が昇順に並べられた**リスト xs の先頭の n 以下の要素のリストを返す。例えば、(take_upto 3 [2 3 4]) は [2 3] に、(take_upto 2 [4 6 8]) は、 [] になる。

関数 take_upto を次のように定義する。空欄を埋め take_upto の定義を完成させよ。

```
(define (take_upto n xs)
   (if [      (1)      ]
       '()
       [                (2)                ]))
```

以下の JavaScript 版の空欄を埋めても良い。

```
function take_upto(n, xs) {
  if ([     (1)     ]) {
    return nil;
  } else {
    return [                (2)                ];
  }
}
```

IV. 数のリストを数列とみなして、差分を作る関数 sabun と、初項と差分数列から元の数列を復元する関数 fukugen を定義する。 例えば、 (sabun [1 2 3 4]) は [1 1 1] に、(sabun [1 2 4 8 16]) は [1 2 4 8] に、(fukugen 1 [1 1 1]) は [1 2 3 4] になる。以下の Scheme, JavaScript のどちらかのプログラムの空欄を式で埋め sabun, fukugen の定義を完成させよ。

Scheme の場合:

```
(define (sabun xs)
   (if (null? (cdr xs))
       '()
       (cons [                (1)                ]
             [                (2)                ])))

(define (fukugen a xs)
  (cons a
      (if (null? xs) '()
          [                (3)                ])))
```

JavaScript の場合:

```
function sabun(xs) {
  if (isNull(cdr(xs))) {
    return nil;
  } else {
    return cons(        (1)        ,
                        (2)        );
  }
}

function fukugen (a, xs) {
  return
  cons(a, isNull(xs) ? nil :        (3)        );
}
```

V. 下の空欄を lambda 式（匿名関数）で埋めよ。

(1) 数列 $\{2 \cdot i - 1\}$ の第 n 項までの積 $\prod_{i=1}^{n} (2 \cdot i - 1)$ を計算する関数 foo の定義は

Scheme 版:

```
(define (foo n)
  (prod (iterate2 1 |      (1)      | n))
```

JavaScript 版:

```
function foo(n) {
  return prod(iterate2(1,|      (1)      |, n));
}
```

である。

(2) $\begin{cases} a_1 &= 1 \\ a_i &= (a_{i-1})^2 + 1 \quad (i \geq 2) \end{cases}$ という漸化式で定義される数列 $\{a_i\}$ の第 n 項までの和 $\sum_{i=1}^{n} a_i$ を計算する関数 bar の定義は

Scheme 版:

```
(define (bar n)
  (sum (iterate2 1 |      (2)      | n)))
```

または、JavaScript 版:

```
function bar(n) {
  sum(iterate2(1, |      (2)      |, n));
}
```

である。

なお sum, prod, iterate2 は次のように定義された関数である。

Scheme 版:

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

(define (prod xs)
  (if (null? xs)
     1
     (* (car xs) (prod (cdr xs)))))

(define (iterate2 a f n)
   (if (= n 0) '()
       (cons a (iterate2 (f a) f (- n 1)))))
```

JavaScript 版:

```
function sum(xs) {
  if (isNull(xs)) {
     return 0;
  } else {
     return car(xs)+sum(cdr(xs));
  }
}

function prod(xs) {
  if (isNull(xs)) {
     return 1;
  } else {
     return car(xs)*prod(cdr(xs));
  }
}

function iterate2(a, f, n) {
  if (n==0) {
    return nil;
  } else {
    return cons(a, iterate2(f(a), f, n-1));
  }
}
```

VI. Scheme や JavaScript は ML と事なり、型を推論しないが、それでも型の間違ったプログラムは実行時にエラーとなってしまう。それゆえプログラミングの際に型を意識する事は有用である。そこで、次のように定義された関数`iterate`:

Scheme 版:

```
(define (iterate a f p)
   (if (p a)
       '()
       (cons a (iterate (f a) f p))))
```

JavaScript 版:

```
function iterate(a, f, p) {
  if (p(a)) {
    return nil;
  } else {
    return cons(a, iterate(f(a), f, p));
  }
}
```

に対して、次の文章の空欄を埋めるのに最も適した語を指示された選択肢の中から選べ。ただし、α や β は型変数で、同じ型である必要が場合には、同じ型変数を、そうでなければ異なる型変数を用いる事にする。

`iterate` の第 1 引数（`a`）の型を α とすると、第 2 引数（`f`）の型は「 (1) を受け取って、 (2) を返す関数の型」であり、第 3 引数（`p`）の型は「 (3) を受け取って、真偽値型を返す関数の型」である。`iterate` の戻り値の型は (4) である。

選択肢

(A). 真偽値型　(B). 整数型　(C). α　(D). β
(E). 要素の型が真偽値型であるようなリスト型
(F). 要素の型が整数型であるようなリスト型
(G). 要素の型が α がであるようなリスト型
(H). 要素の型が β がであるようなリスト型

VII. リスト型を C の構造体として、次のように定義する。（ただし、リストの要素としては int 型のみを想定する。）

```
struct _list {
  int car;
  struct _list* cdr;
};

typedef struct _list* list;
```

また空リストは NULL で表す。

Scheme の関数 count_odd

```
(define (count_odd xs)
  (if (null? xs)
     0
     (if (odd? (car xs))
         (+ 1 (count_odd (cdr xs)))
         (count_odd (cdr xs)))))
```

あるいは JavaScript の関数 count_odd

```
function count_odd(xs) {
  if (isNull(xs)) {
    return 0;
  } else if (car(xs)%2==1) {
    return 1+count_odd(cdr(xs));
  } else {
    return count_odd(cdr(xs));
  }
}
```

とほぼ同等の動きをする（つまり、リスト中の奇数の個数を求める）C の関数cound_odd を再帰ではなく繰り返し（for 文）を使って定義する。空欄を式で埋め、count_odd の定義を完成させよ。（car, cdr にあたる C の関数は用意していないので、通常の構造体のメンバアクセスの記法を用いること。）

```
int count_odd(list xs) {
  int ret = 0;
  for( ;  (1)  ;  (2)  ) {
      (3)
  }
  return ret;
}
```

プログラミング言語論・テスト解答用紙('02 年 7 月 18 日)

学部	学籍番号		氏名	

I. (6 点 ×2)

(1).	
(2).	

II. (7 点)

```
(zip [1 4 7] [[2 3] [5 6]])
```

⇒

III. (3 点, 4 点)

(1).	
(2).	

IV. (4 点 ×3)

(1).	
(2).	
(3).	

V. (5 点 ×2)

(1).	
(2).	

VI. (4 点 ×4)

(1).		(2).		(3).		(4).	

VII. (5 点, 5 点, 6 点)

(1).	
(2).	
(3).	

授業・テストの感想

……

……

……

……

……

……

……

……

……