プログラム言語論（’06年度）・テスト問題用紙

（’07年2月6日（火）・ 8:50 〜 10:20）

解答上、その他の注意事項

I. 問題は、問 I〜VI まである。

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. 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。

V. 持ち込みは不可である。筆記用具・時計・学生証以外のものは、かばんの中などにしまうこと。

VI. 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。

I. （Backus-Naur 記法）
下の記号列の中で、次の BNF

$$N \ \rightarrow \ N\,0 \ | \ 1\,N\,1 \ | \ \varepsilon$$

の 非終端記号 N から生成されるものには○、生成されないものには ×をつけよ。

(1) 1 0 1　　(2) 0 1 0　　(3) 0 1 1　　(4) 1 1 1 1

II. （コンパイラのフェーズ）
次の (1)〜(3) の C 言語のプログラムにはそれぞれ誤りがある。コンパイラのどのフェーズで誤りが検出されるか？(あるいはされないか？) もっとも適当なものを下の選択肢 (A)〜(E) から選べ。

(1) （ブロックの “{”〜“}” の代わりに大括弧 “[”〜“]” を使った。）

```
#include <stdio.h>

int main(void) [
    int i;
    for (i=0; i<10; i++) [
        printf("Hello World!¥n");
    ]
]
```

(2) （正弦関数 double sin(double) に文字列を渡した。）

```
#include <stdio.h>
#include <math.h>

int main(void) {
    printf("sin(45 °) = %f¥n", sin("45 °"));
    return 0;
}
```

(3) （文字列リテラルの終わりを示す「"」を忘れた。）

```
#include <stdio.h>

int main(void) {
    printf("Hello! World¥n);
    return 0;
}
```

(1)〜(3) の選択肢

(A) 字句解析フェーズでエラーが検出される。

(B) 構文解析フェーズでエラーが検出される。

(C) 意味解析フェーズでエラーが検出される。

(D) コード生成フェーズでエラーが検出される。

(E) 実行時にエラーとなるか、まったくエラーにならない (が作成者の意図と異なる動作をする)。

III. （正規表現）

以下の表で「(ab|ba)*(a|ε)」と「(aba|bab)*(b|ε)」という正規表現について、それぞれ、ababaabab、 ababababa、abababbab という文字列と全体がマッチする時は○を全体がマッチしない時は ×をつけよ。

	ababaabab	ababababa	abababbab
(ab|ba)*(a|ε)	(1)	(2)	(3)
(aba|bab)*(b|ε)	(4)	(5)	(6)

IV. （演算子順位法）

次の BNF で表される文法を演算子順位法により構文解析する。

$$E \rightarrow \quad \textbf{id} \;\mid\; E \text{ “\#” } E \;\mid\; E \text{ “+” } E \;\mid\; E \text{ “*” } \;\mid\; \text{“(” } E \text{ “)”}$$

ただし、この文法は曖昧なので、優先順位と結合性について次のように決めておく。

> 「#」と「+」は、いずれも右結合で、「*」は「#」 よりも優先順位が高く、「#」は「+」よりも優先順位が高いものとする。つまり、「a # b *」は 「a # (b *)」と解釈され、「a + b # c」は「a + (b # c)」と解釈される。

以下の演算子順位行列の空欄を ⋖（シフト）、 ⋗（還元）、**err**（エラー）のいずれかで埋めよ。

左 \ 右	+	#	*	(	)	**id**	終
始	⋖	⋖	⋖	⋖	**err**	⋖	≐
+	(1)	(2)	⋖	⋖	⋗	⋖	⋗
#	(3)	(4)	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋗	**err**	⋗	**err**	⋗
(	⋖	⋖	⋖	⋖	≐	⋖	**err**
)	⋗	⋗	⋗	**err**	⋗	**err**	⋗
id	⋗	⋗	⋗	**err**	⋗	**err**	⋗

V. （再帰下降構文解析）

次のような BNF で表された文法に対して、再帰下降構文解析ルーチンを作成する。

$$
\begin{aligned}
S &\rightarrow \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{begin}\ L\ \textbf{end} \mid \textbf{print}\ E\ \text{“;”} \\
L &\rightarrow \varepsilon \mid S\ L \\
E &\rightarrow F \mid E\ \text{“+”}\ F \\
F &\rightarrow \textbf{id} \mid \text{“(”}\ E\ \text{“)”}
\end{aligned}
$$

ただし、「S」,「L」,「E」,「F」は非終端記号、「**if**」,「**then**」,「**else**」,「**begin**」,「**end**」,「**print**」,「**;**」,「+」,「**id**」,「(」,「)」は終端記号である。また、開始記号は「S」である。

(1) E の生成規則から左再帰を除去せよ。ただし補助的な非終端記号として E' を用いること。

(2) *First*(S) を求めよ。

(3) *Follow*(L) を求めよ。

(4) 〜 (6)

この文法に対して、入力が文法にしたがっていれば「正しい構文です。」間違っていれば「構文に誤りがあります。」と表示する構文解析プログラムを作成する。プログラム（次ページ）中の空欄 (4)〜(6) を埋めよ。ただし、非終端記号 S, L, E, E', F に対応する関数は、それぞれ `S`, `L`, `E`, `E1`, `F` である。

（**プログラムの補足説明:** プログラム中では、終端記号は、“**;**” のような 1 文字のものは、その字そのもの（の ASCII コード）、**if** などのキーワードは、C 言語のマクロ（例えば **if** の場合は `IF`）として表現している。

`yylex` 関数は、入力を読んで、次の終端記号を返す関数である。`token` という大域変数に、現在処理中の終端記号が代入される。`eat` 関数は、現在 `token` に入っている値が、引数として与えられた終端記号と等しいかどうか確かめ、等しければ次の終端記号を読み込む。）

再帰下降構文解析プログラム

```
/* これより上の部分は問題に直接関係がないので後掲 */

/* 関数プロトタイプ宣言 */
void S(void);
void L(void);
void E(void);
void E1(void);
void F(void);

void S(void) {
  if (token == IF) {
    eat(IF); E(); eat(THEN); S(); eat(ELSE); S();
  } else if (token == BEGIN) {
      (4)
  } else if (token == PRINT) {
    eat(PRINT); E(); eat(';');
  } else {
    printf("構文に誤りがあります。¥n"); exit(0); /* プログラムを終了 */
  }
}

void L(void) {
  if ( (5) ) {
    S(); L();
  } else if ( (6) ) {
    /* 何もしない */
  } else {
    printf("構文に誤りがあります。¥n"); exit(0); /* プログラムを終了 */
  }
}

/* 関数 E(), E1(), F() は直接問題に関係ないので省略する */

int main() {   /* main 関数 */
  token = yylex(); /* 最初のトークンを読む */
  S();
  printf("正しい構文です!¥n");
}
```

（参考） 上のプログラムで省略された最初の部分（問題には直接は関係ない。）

```
#include <stdio.h>
#include <stdlib.h>   /* exit() 用    */
#include <string.h>   /* strcmp() 用  */
#include <ctype.h>    /* isalpha() 用 */

/* 終端記号に対するマクロ */
#define IF      257     /* トークン if    */
#define THEN    258     /* トークン then  */
#define ELSE    259     /* トークン else  */
#define BEGIN   260     /* トークン begin */
#define END     261     /* トークン end   */
#define PRINT   262     /* トークン print */
#define NUM     263     /* トークン num   */
```

次ページに続く

```
int token;            /* 大域変数の宣言 */

int yylex(void) {     /* 簡易字句解析ルーチン */
  int c;
  char buf[256];

  do { /* 空白は読み飛ばす。 */
    c = getchar();
  } while (c == ' ' || c == '¥t' || c == '¥n');

  if (isalpha(c)) {  /* アルファベットだったら… */
    char* ptr = buf;
    ungetc(c, stdin);
    while (1) {
      c=getchar();
      if (!isalpha(c) && !isdigit(c)) break;
      *ptr++ = c;
    }
    *ptr = '¥0';
    ungetc(c, stdin);

    if (strcmp(buf, "if")==0) {        /* キーワードでないか */
      return IF;
    } else if (strcmp(buf, "then")==0) {
      return THEN;
    } else if (strcmp(buf, "else")==0) {
      return ELSE;
    } else if (strcmp(buf, "begin")==0) {
      return BEGIN;
    } else if (strcmp(buf, "end")==0) {
      return END;
    } else if (strcmp(buf, "print")==0) {
      return PRINT;
    } else {
      printf("構文に誤りがあります。¥n"); exit(0); /* プログラムを終了 */
    }
  } else if (isdigit (c)) {  /* 数字だったら… */
    double poi;
    ungetc(c, stdin); scanf("%lf", &poi);
    return NUM;
  } else {
    /* 上のどの条件にも合わなければ、文字をそのまま返す。*/
    return c;   /* '(', ')', '+', ';' など */
  }
}

void eat(int t) {       /* token（終端記号）を消費して、次の token を読む */
  if (token == t) {
    /* 現在のトークンを捨てて、次のトークンを読む */
    token = yylex();
    return;
  } else {
    printf("構文に誤りがあります。¥n"); exit(0); /* プログラムを終了 */
  }
}
```

VI. （LR 構文解析）

次の BNF で与えられる文法は曖昧であるが、優先順位を適当に指定することにより、LR 構文解析表を作成することができる。

$$
\begin{array}{lcll}
E & \rightarrow & \textbf{while}\ E\ \textbf{do}\ E & \cdots\ \text{I} \\
 & | & \textbf{id}\ \text{“=”}\ E & \cdots\ \text{II} \\
 & | & E\ \text{“+”}\ E & \cdots\ \text{III} \\
 & | & \textbf{id} & \cdots\ \text{IV}
\end{array}
$$

ただし、

- ⋯のあとの I, II などは生成規則の番号である。
- *E* は非終端記号、**while**, **do**, **id**, =, +は終端記号である。**id** はアルファベット 1 文字からなるトークンを表す。

下に示すのは `bison` が自動生成した LR 構文解析表である。

	while	**do**	**id**	=	+	$	*E*
⓪	shift ①		shift ②				goto ③
①	shift ①		shift ②				goto ④
②	reduce IV			shift ⑤	reduce IV		
③					shift ⑥	accept	
④		shift ⑦			shift ⑥		
⑤	shift ①		shift ②				goto ⑧
⑥	shift ①		shift ②				goto ⑨
⑦	shift ①		shift ②				goto ⑩
⑧	reduce II				shift ⑥	reduce II	
⑨	reduce III						
⑩	reduce I				shift ⑥	reduce I	

ここで、shift Ⓢは「シフトして状態Ⓢへ遷移」、 goto Ⓢは、「状態Ⓢへ遷移」、 reduce X は、「生成規則 X を使って還元」を表す。

次の入力に対して、↑の次（右）の記号を**シフトした直後**の（つまりシフトしたあと、還元がまだ起こっていない時の）スタックの状態はどのようになっているか? 下の選択肢から選べ。（左がスタックの底とする）

(1) **id=id+id**　　(2) **while id+id do id=id+id**

(1) の選択肢

(A). ⓪*E*③=⑤*E*⑧+⑥　　(B). ⓪*E*③=⑤**id**②+⑥

(C). ⓪**id**②=⑤*E*⑧+⑥　　(D). ⓪**id**②=⑤**id**②+⑥

(2) の選択肢

(A). ⓪**while**①*E*④**do**⑦　　(B). ⓪**while**①*E*④+⑥**id**②**do**⑦

(C). ⓪**while**①**id**②+⑥**id**②**do**⑦　　(D). ⓪**while**①*E*④+⑥*E*⑨**do**⑦

プログラム言語論（’06 年度）・テスト解答用紙 (’07 年 2 月 6 日)

学籍番号		氏名	

I. （Backus-Naur 記法） (3×4)

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

II. （コンパイラのフェーズ） (3×3)

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

III. （正規表現） (3×6)

(1).		(2).		(3).	
(4).		(5).		(6).	

IV. （演算子順位法） (4×4)

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

V. （再帰下降構文解析） (4, 4, 4, 3, 2, 2)

(1).

$E \rightarrow$

$E' \rightarrow$

(2).	{ }
(3).	{ }
(4).	
(5).	
(6).	

VI. （LR 構文解析） (3×2)

(1).		(2).	

テストの感想