言語処理プログラミング課題２

（１）課題名：プリティプリンタの作成
（２）演習期間：10/19 - 11/15
（３）レポート・プログラム提出期間：11/16 - 11/23

（４）課題

プログラミング言語 MPPL で書かれたプログラムを読み込み, LL(1)構文解析法により構文解析を行い，構文エラーがなければ入力されたプログラムをプリティプリントした結果を出力し，構文エラーがあればそのエラーの情報（エラーの箇所，内容等）を少なくとも一つ出力するプログラムを作成する．例えば，作成するプログラム名を pp，MPPL で書かれたプログラムのファイル名を foo.mpl とするとき，コマンドラインからのコマンドを

pp foo.mpl

とすれば(foo.mpl のみが引数)，foo.mpl の内容のプリティプリントを出力するプログラムを作成する．

入力：MPPL で書かれたプログラムのファイル名．コマンドラインから与える．MPPL のマイクロ構文は課題１の通りである．マクロ構文は次の通り（左辺の括弧の中は非終端記号の英語表記であり参考のために付加してある)．終端記号は字句(token)である．なお，「//」以降は構文に関する制約や注意である．

プログラム(program) ::= "program" "名前" ";" ブロック "."
ブロック(block) ::= { 変数宣言部 | 副プログラム宣言 } 複合文
変数宣言部(variable declaration) ::= "var" 変数名の並び ":" 型 ";" { 変数名の並び ":" 型 ";" }
変数名の並び(variable names) ::= 変数名 { "," 変数名 }
変数名(variable name) ::= "名前"
型(type) ::= 標準型 | 配列型
標準型(standard type) ::= "integer" | "boolean" | "char"
配列型(array type) ::= "array" "[" "符号なし整数" "]" "of" 標準型
副プログラム宣言(subprogram declaration) ::=
"procedure" 手続き名 [仮引数部] ";" [変数宣言部] 複合文 ";"
手続き名(procedure name) ::= "名前"
仮引数部(formal parameters) ::= "(" 変数名の並び ":" 型 { ";" 変数名の並び ":" 型 } ")"
複合文(compound statement) ::= "begin" 文 { ";" 文 } "end"
文(statement) ::= 代入文 | 分岐文 | 繰り返し文 | 脱出文 | 手続き呼び出し文 | 戻り文 | 入力文 | 出力文 | 複合文 | 空文
分岐文(condition statement) ::= "if" 式 "then" 文 ["else" 文]
// どの"if"に対応するか曖昧な"else"は候補の内で最も近い"if"に対応するとする．
繰り返し文(iteration statement) ::= "while" 式 "do" 文
脱出文(exit statement) ::= "break"

　　// この脱出文は少なくとも一つの繰り返し文に含まれていなくてはならない
手続き呼び出し文(call statement) ::= "call" 手続き名 ["(" 式の並び ")"]
式の並び(expressions) ::= 式 { "," 式 }
戻り文(return statement) ::= "return"
代入文(assignment statement) ::= 左辺部 ":=" 式
左辺部(left part) ::= 変数
変数(variabble) ::= 変数名 ["[" 式 "]"]
式(expression) ::= 単純式 { 関係演算子　単純式 }
　　// 関係演算子は左に結合的である.
単純式(simple expression) ::= ["+" | "-"] 項 { 加法演算子　項 }
　　// 加法演算子は左に結合的である.
項(term) ::= 因子 { 乗法演算子　因子 }
　　// 乗法演算子は左に結合的である.
因子(factor) ::= 変数 | 定数 | "(" 式 ")" | "not" 因子 | 標準型 "(" 式 ")"
定数(constant) ::= "符号なし整数" | "false" | "true" | "文字列"
乗法演算子(multiplicative operator) ::= "*" | "div" | "and"
加法演算子(additive operator) ::= "+" | "-" | "or"
関係演算子(relational operator) ::= "=" | "<>" | "<" | "<=" | ">" | ">="
入力文(input statement) ::= ("read" | "readln") ["(" 変数 { "," 変数 } ")"]
出力文(output statement) ::= ("write" | "writeln") ["(" 出力指定 { "," 出力指定 } ")"]
出力指定(output format) ::= 式 [":" "符号なし整数"] | "文字列"
　　// この"文字列"の長さ（文字数）は 1 以外である.
　　// 長さが 1 の場合は式から生成される定数の一つである"文字列"とする.
空文(empty statement) ::= ε

［注意］構文において，用いられている記号の意味は次の通りである.
　" "　　終端記号であることを示す
　::=　　左辺の被終端記号が右辺で定義されることを示す
　|　　左側と右側のどちらかであることを示す
　{ }　内部を 0 回以上繰り返すことを表す
　[ ]　内部を省略してもよい(0 回か 1 回)ことを示す
　( )　優先順位を変更する．通常は「|」がもっとも弱いが，「( )」により，その内部が優先される

出力：入力のプログラム中に構文的な誤りがなければ，そのプログラムのプリティプリント（下記参照）を標準出力へ出力せよ．もし，構文的な誤りがあれば，それを最初に検出した時点で，入力ファイルでの検出した行の番号（つまり，入力ファイルの何行目に誤りがあったか）と誤りの内容（演算子が必要なところで演算子がない，など）を標準出力へ出力せよ．エラー検出の前にその部分までのプリティプリントを出力してもかまわない.

［プリティプリント］

プログラムのプリティプリントとは，見やすく段付けして印刷されたプログラムリストである．プログラムの見やすさには主観的な要素が多く，どのようなリストがもっとも見やすいかは一概に言えないが，本課題においては，たとえば，次の条件を満たすものとする．

・行頭を除いて複数の空白やタブは連続しない．行頭を除いてタブは現れない．行末には空白やタブは現れない．すなわち，特に指定されていない限り，行中の字句と字句の間は一つの空白だけがある．
・前項の指示に関わらず，式中で字句間の空白を省いても字句の認識が変わらず，より見やすくなると思われるときは，空白を省いても良い．また，より見やすくなると思われるときは，","，";"や"."の直前の空白も省いて良い．
・字句";"の次は必ず改行される．ただし，仮引数中の";"については見やすさによって改行しなくてよい．
・最初の字句"program"は段付けされない．つまり，1 カラム目（行頭）から表示される．
・変数宣言部は行頭から 1 段段付けされる．
・副プログラム宣言(キーワード)procedure で始まる行は行頭から 1 段段付けされる．
・副プログラム宣言内の一番外側の begin", "end"は行頭から 1 段段付けされる．
・それ以外の一番外側の"begin", "end"は段付けされない．
・対応する"begin"， "end"が段付けされるときは同じ量だけ段付けされる．直前の字句に引き続いて"end"を印刷すると，対応する"begin"より段付け量が多くなるときは改行して同じにする．
・対応する"begin", "end"の間の文は，その"begin", "end"より少なくとも 1 段多く段付けされる．
・分岐文の"else"の前では必ず改行し，その段付けの量は対応する"if"と同じである．
・分岐文，繰り返し文中の文が複合文でなく，"then", "else", "do"の次で改行するときは，その改行後の文は"if"よりも 1 段多く段付けされる．
・分岐文，繰り返し文中の文が複合文のときは，その先頭の"begin"の前で改行する．
・以上に指定のない点については，見やすさに基づいて字句を配置すること．
・注釈は削除する．その結果，構文解析結果が変わるときには空白を一つ入れる．

見やすいとする条件を明示するならば，必ずしも，この条件でなくてもよいが，少なくとも次の条件を満足しなくてはならない．

・構文的に対応する"begin"と"end"の間の文は，そのすぐ外側の部分よりも少なくとも 1 段多く段付けされる．それらの文がさらに "begin"と"end"に挟まれていない限り，それらの文の段付け量は同じである．
・副プログラム宣言の一番段付け量の少ない部分は，そのすぐ外側の部分よりも 1 段多く段付けされる
・上記の 2 条件を満足し，かつ，見やすさを損なわない限り，不要な空白，タブ，改行を削除する．ここで，「不要」というのは，その空白等がなくても構文解析結果が変わらないという意味である．
・注釈は削除する．その結果，構文解析結果が変わるときには空白を一つ入れる．

なお，段付け 1 段分の量は 4 カラム（空白 4 文字分）とする．

［まとめ］

直感的には，プリティプリントとは，注釈及び無駄な空白やタブがなく（字句と字句の間には一つ空白があってよい)，副プログラム宣言部や複合文，ブロック内部はそのすぐ外よりも一段段付けがされているようなプログラムリストである．

［プリティプリントの例(1)］

/* プリティプリントする前のリスト */

```
program sample11;var n,sum,data:integer;begin
writeln （('input the number of data')　;readln（　n　）;sum:=0 ;while　　n>0do
begin　readln(data);sum:=sum+data;n:=n-1end;writeln('Sum of data = ',sum) end.
```

/* プリティプリンタの出力 */

```
program sample11;
    var n, sum, data : integer;
begin
    writeln('input the number of data');
    readln(n);
    sum := 0;
    while n > 0 do
    begin
        readln(data);
        sum := sum + data;
        n := n - 1
    end;
    writeln('Sum of data = ', sum)
end.
```

［プリティプリントの例(2)］

/* プリティプリントする前のリスト */

```
program sample11pp;
procedure kazuyomikomi(n : integer); begin　writeln('input the number of data'); readln(n)　end;
var sum : integer; procedure wakakidasi; begin　writeln('Sum of data = ', sum)　end; var data : integer;
procedure goukei(n, s : integer); var data : integer; begin
s := 0; while n > 0 do begin readln(data); s := s + data; n := n - 1 end
end; var n : integer; begin call kazuyomikomi(n); call goukei(n * 2, sum); call wakakidasi
                                                                                    end.
```

/* プリティプリンタの出力 */

```
program sample11pp;
    procedure kazuyomikomi（ n : integer ）;
    begin
        writeln（ 'input the number of data' ）;
        readln（ n ）
    end;
    var
        sum : integer;
    procedure wakakidasi;
    begin
        writeln（ 'Sum of data = ' , sum ）
    end;
    var
        data : integer;
    procedure goukei（ n , s : integer ）;
        var
```

```
            data : integer;
    begin
        s := 0;
        while n > 0 do
        begin
            readln( data );
            s := s + data;
            n := n - 1
        end
    end;
    var
        n : integer;
begin
    call kazuyomikomi( n );
    call goukei( n * 2 , sum );
    call wakakidasi
end.
```

LL(1)構文解析系の作り方（課題２相当）

前期のコンパイラの講義で，説明したとおり，次のような手順で作れます．

（１）EBNF 記法で書かれた文法を用意する．
これは課題にあります．

（２）EBNF の規則の左辺がすべて異なるようにする．
もし，同じ左辺を持つ規則があれば，右辺を「|」で結んだ規則に置き換えて，一つにする．たとえば，
A ::= α
A ::= β
という規則があれば，
A ::=（α）|（β）
に置き換え，各非終端記号に対して，それを左辺に持つ規則を一つにする．

与えられた規則には左辺が同じものはないはずです．

（３）各規則に対して（つまり，各非終端記号に対して），構文解析処理関数を一つずつ作る．そのとき，
・終端記号に対しては，それが次の字句であることを確認する
・非終端記号に対しては，その非終端記号に対応する関数を呼ぶ
・「..... | .....」は switch 文か if 文に置き換える
・「{ ..... }」は while 文に置き換える
・「[.....]」は if 文になる
ここでループの中に入るかどうか，if などの選択肢のどこへ行くのかは，それぞれの選択肢の **FIRST 集合**を計算して，次の字句がそれに属している方へ処理が進むようにプログラムします．ここで，それぞれの FIRST 集合が共通部分を持てば，どちらへ行けばいいのか決定できないので，LL(1)構文解析ができな

いことになります.
　たとえば,
　　プログラム ::= "program" "名前" ";" ブロック "."
に対しては,

```
int parse_program() {
  if(token != TPROGRAM) return(error("Keyword 'program' is not found"));
  token = scan();
  if(token != TNAME) return(error("Program name is not found"));
  token = scan();
  if(token != TSEMI) return(error("Semicolon is not found"));
  token = scan();
  if(block() == ERROR) return(ERROR);
  if(token != TDOT) return(error("Period is not found at the end of program"));
  token = scan();
  return(NORMAL);
}
```

とすればいいし,
　　項 ::= 因子 { 乗法演算子　因子 }
に対しては,

```
int kou() {
  if(inshi() == ERROR) return(ERROR);
  while(token == TSTAR || token == TAND || token == TDIV) {
      /* 「乗法演算子 因子」の FIRST 集合の要素が，TSTAR, TAND, TDIV なので，この条件となる */
    if(jouhouenzannshi() == ERROR) return(ERROR);
    if(inshi() == ERROR) return(ERROR);
  }
  return(NORMAL);
}
```

とすればよいわけです．このように関数をすべての非終端記号について（生成規則について）作ればよいわけです.
　もちろん，最初に,

```
#define NORMAL 0
#define ERROR 1
int token;
```

と宣言しておく必要があります．また，ここで使っている関数 error()はエラーメッセージを出力して，戻り値として，ERROR を返す関数であるとしています．たとえば,

```
int error(char *mes) {
        printf("¥n ERROR: %s¥n", mes);
        end_scan();         /* 入力ファイルを閉じる */
        return(ERROR);
}
```

のようにすればよいわけです.

構文解析系を最初に呼び出すためには，スキャナを初期化した後，

```
token = scan();
parse_program();
```

とします.

なお，課題に与えられている EBNF 記法で書かれた構文は，意味をわかりやすくするために冗長な非終端記号が数多く導入されています．たとえば，規則,

項 ::= 因子 ｛ 乗法演算子　因子 ｝

には，非終端記号「乗法演算子」がありますが，この非終端記号はここにしか（つまり，1 カ所しか）ありません.「乗法演算子」の規則は,

乗法演算子 ::= "*" | "div" | "and"

ですので，この二つをまとめて,

項 ::= 因子 ｛（"*" | "div" | "and"）　因子 ｝

としても，何の問題もありません．このように，右辺の 1 カ所にしか現れない非終端記号は，規則の代入によってなくすことができます．なくしてしまえば，作成する関数の数が減り，関数を超えた情報のやりとりも減るので，一つの規則が大きくなりすぎない限りにおいて，プログラムが簡単になります.

以上は構文解析をして，構文エラーのチェックだけをする方法です.

最後に文の処理について簡単にコメントします．文が代入文か○○文か△△文かの判定は，最初のトークンでほとんどわかります．最初のトークンが名前であれば代入文ですし，"if"であれば分岐文です．このように，空文を除いてすべての文の最初のトークンは決まっています．従って，それら以外のトークンが文の先頭にあるときには，そこに空文があるものとしなさい．理論的には，Follow 集合を計算して，空文がある場合とエラーの場合を分ける必要がありますが，実際には､エラーの場合には後でエラーを検出できますので，これで問題ありません.

プリティプリンタについて

上記の LL(1)構文解析系をプリティプリンタにするには，適切な位置に printf 文を加えるだけです．スキャナを呼び出す毎にスキャンした字句（トークン）（トークン番号ではなく，プログラムリスト上での文字列）を printf 文で出力すれば，余計な空白やコメントを削除することができます．このとき，次のような字句の配列があれば便利かもしれませんね.

```
char *token_str[] = {
        "",         /* 対応なし */
```

```
        "",         /* TNAME */
        "program",
        "var",
        "array",
                /* 以下，略 */
    }
```

もちろん，名前，文字列，符号なし整数のときはその実体を出力する必要があります.

ただ，これだけでは字句が隙間なく詰まって出力されてしまいますので，その字句が行頭でなければ，その字句を出力する前に空白文字を一つ出力する，行頭であれば，段付け量だけ出力する，適切なタイミングで改行('¥n')を出力することを加える必要があります.

課題３以降への対応について

型のチェックまでしよう（課題３相当）とすれば，各関数の戻り値が ERROR や NORMAL だけではすみません．たとえば，上の関数 kou()で型のチェックをしようとすれば，被演算子を担当する inshi()からその被演算子の型を返してもらう必要があります. また, 演算子が具体的に何かを知る必要もありますね. 下位の関数から必要な情報をもらって kou()は型のチェックを行うわけです.

コード生成（課題４相当）は，以上の関数群にコード生成を行う出力命令を加えていくだけでできるはずです.

以上の他に，型チェック，コード生成に共通して必要なものは，記号表です．注意して記号表を設計しましょう.