

水野修(@omzn)

課題2の説明





課題2 - 構文解析



課題名



プリティプリンタの作成



演習期間



2024-10-21 ~ 2024-11-17









レポート・プログラム提出期間



2024-11-18 ~ 2024-11-25










課題内容

-  プログラミング言語 MPPL で書かれたプログラムを読み込む.
-  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する.
-  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1つ出力する.
-  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



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.
-  マクロ構文は図 3.1 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.



MPPLプログラムの例

```
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.
```



マクロ構文 (図3.1)


```
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"は候補の中で最も近い"if"に対応するとする。
```


👤 C言語と異なり，文の末尾に";"が付くとは限らない。

👤 elseの対応については次ページを参照



あいまいなelse


 たとえば、右のようなプログラムにおいてはelseは上のifにも下のifにも対応可能である。


 "if" 式 "then" 文 ["else" 文]


 解釈1

 文 → 「if y = 0 then a := 0 else a := 1」

 解釈2

 文 → 「if y = 0 then a := 0」

 "else" 文 → 「else a:=1」


 本課題では解釈1を採用する


```
if x = 0 then
if y = 0 then
a := 0
else
a := 1
```



マクロ構文 (図3.1)

```
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 (variable) ::= 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"
```

 脱出文が繰り返し文に含まれているかどうかはこの段階で判定する.

 式の「左に結合的」は制約として定める.



マクロ構文 (図3.1)

🌳 「出力指定」においては、まず「式」を評価するが、「式」が「"文字列"」のみであった場合には、その長さが1であれば「式」を採用し、そうでなければ「式」ではないと判定し、「出力指定」の「"文字列"」であると判定する。

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

```
writeln('c':8)
c は長さ1なので式
```




```
writeln('')
'' は長さ1なので式
```

```
writeln('abc')
abc は長さ3なので"文字列"
```

```
writeln('abc':8)
構文エラー
```









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）を標準エラー出力へ出力せよ。
-  エラー検出の前にその部分までのプリティプリントを標準入力に出力してもかまわない。



出力

プリティプリント

-  (1) 段付の1段は空白4文字とする.
-  (2) 行頭を除いて複数の空白は連続しない. 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1つの空白だけがある.
-  (3) 前項の指示に関わらず, ";", "."の直前には空白を入れない.
-  (4) 字句 ";"の次は必ず改行される. ただし, 仮引数中の ";"については改行しない.
-  (5) 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.
-  (6) 変数宣言部 "var"は行頭から1段段付けされる. また, 変数の並びは改行し, "var"からさらに1段段付けする.

```
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.
```



出力



プリティプリント



(7) 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.



(8) 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.



(9) 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.



(10) 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.

(7)

```
program sample11pp;
```

(8)

```
procedure kazuyomikomi ( n : integer );
```

```
begin
```

```
    writeln ( 'input the number of data' );
```

```
    readln ( n )
```

(8)

```
end;
```

```
var
```

```
    sum : integer;
```

```
    sum := 0;
```

```
while n > 0 do
```

```
begin
```

```
    readln ( data );
```

```
    sum := sum + data;
```

```
    n := n - 1
```

```
end;
```

(9)

(10)



出力



プリティプリント



(11) 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.



(12) 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", "while"よりも1段多く段付けされる.



(13) 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"の前で改行し，段付けする.



(14) 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.

(12)

(11)

```
if sum = 0 then
  writeln ( 'Sum is zero' )
else
  writeln ( 'Sum is non zero' )
```

(13)

```
sum := 0;
while n > 0 do
  begin
    readln ( data );
    sum := sum + data;
    n := n - 1
  end;
writeln ( 'Sum of data = ' , sum )
end.
```



LL(1) 構文解析系の作り方

- 👤 EBNF 記法で書かれた文法を用意する (図3.1)
- 👤 EBNF の規則の左辺がすべて異なるようにする (対応済み)
- 👤 構文解析処理関数を各規則に対して1つずつ作る
 - 👤 終端記号に対しては, それが次に来るべき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(規則)に対応する関数を呼ぶ.
 - 👤 「`... | ...`」は, switch文かif文で場合分け
 - 👤 「`{ ... }`」はwhile文で実装する
 - 👤 「`[...]`」はif文で実装する
 - 👤 ifなどの選択枝のどこへ行くのかは, 選択枝のFIRST集合を計算して, 次の字句が属している方へ処理が進む. FIRST集合が共通集合を持つ場合は, LL(1)構文解析ができない. (その場合, 特別な処理などで無理矢理解決することもある)



構文解析処理関数の実例

プログラム (program) ::= "program" "名前" ";" ブロック "."

```
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 (parse_block() == ERROR) return(ERROR);  
    if (token != TDOT) return(error("Period is not found at the end of program  
"));  
    token = scan();  
    return(NORMAL);  
}
```

🧑 parse_program() は一番始めに呼ぶ関数となる。全ての処理はここから再帰的に行われる。



構文解析処理関数の実例

```
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.
```

```
int parse_term() {  
    if(parse_factor() == 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(parse_multiple_opr() == ERROR) return(ERROR);  
        if(parse_factor() == ERROR) return(ERROR);  
    }  
    return(NORMAL);  
}
```

🏠 FIRST集合はほとんどの場合, 自明である.



プリティプリンタの作り方



LL(1)構文解析系を作る.







構文解析だけを行い, 構文解析エラーを検出できるもの.



構文解析処理関数の適切な場所に, 読み取った字句を適切な段付や改行を付加しながらprintf()する処理を挿入していく.



テスト

-  プリティプリンタの出力仕様には曖昧な点が多いため、こちらが想定した出力でなくても仕様を満たす可能性は十分ある.
-  dockerで提供する課題2用のテスト"02test/pp_test.py"は、仕様を満たすものの一例であり、これにPASSしないからといって、作ったプログラムが仕様を満たしていないとは限らない.
-  02test/pp_mm_test.pyは、自身が出力したプリティプリント結果を再度自身のプログラムへの入力とし、その結果が同一かどうかを判定している(冪等性判定).
-  正しいプログラムを入力した場合は、出力も正しいプログラムになるべき. プリティプリントはプログラムの動作・構造に影響は及ぼさない.