プログラミング言語レポート課題

us162039　梶田悠

2019

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| (1) | (2) | (3) | (4) | (5) |
| ◎ | ◎ | ◎ | ◎ |  |
| (6) |  | | | |

2019

目次

[課題内容 2](#_Toc30200179)

[ソースコード 3](#_Toc30200180)

[使い方 11](#_Toc30200181)

[全体を通して共通する文法 11](#_Toc30200182)

[if文 11](#_Toc30200183)

[文字列 12](#_Toc30200184)

[実行結果 12](#_Toc30200185)

[解決アプローチ 13](#_Toc30200186)

[(1) 引数を２つ以上扱える数学的関数を取り扱えるようにする。 13](#_Toc30200187)

[(2) 実行文を:で区切って、１行に複数の文を記述する 16](#_Toc30200188)

[(3) if 文を使えるようにする。 17](#_Toc30200189)

[真偽値の導入 17](#_Toc30200190)

[if文の導入 18](#_Toc30200191)

[(4) 変数に文字列を代入できるようにする。 20](#_Toc30200192)

[考察 21](#_Toc30200193)

[関数の入れ子 21](#_Toc30200194)

[if文の問題点 21](#_Toc30200195)

[テーブルの可変長化 22](#_Toc30200196)

# 課題内容

　講義中の題材として扱っていた電卓を、以下の点について改良せよ。

(1) 引数を２つ以上扱える数学的関数を取り扱えるようにする。

(2) 実行文を:で区切って、１行に複数の文を記述する（全体で１つの複文となる）

(3) 次のような if 文を使えるようにする。 if( 条件式 ) 実行文

(4) 変数に文字列を代入できるようにする。

(5) 文字を扱う関数を取り扱えるようにする。

(6) その他、この電卓に追加すると良いと思う機能を考え、実現してみよ。

# ソースコード

コードは[github](https://github.com/HarukaKajita/flexbison_calculator)にアップロードしてあるので、レポートのコードが読みづらい場合は適宜参照して下さい。

ソースコード全文を以下に掲載する。

コード 1 report.h

|  |
| --- |
| #ifndef REPORTHEADER  #define REPORTHEADER  #define SYMBOLNUM 20  typedef enum symbolType{          undefined,          numeric,          boolean,          characters,          function  } t\_symbolType;  struct symbol{          char\* name;          t\_symbolType type;          //-------------------          double (\*funcptr)();          double value;          char\* charPtr;  } symbolTable[SYMBOLNUM];  struct symbol\* lookSymbol(char\* name);  #endif |

コード 2 report.l

|  |
| --- |
| %{    #include "y.tab.h"    #include "report.h"    #include <math.h>    #include <stdio.h>    #include <string.h>  %}  %%  if {    return IF;  }  endif {    return ENDIF;  }  true {    yylval.bval = 1;    return BOOL;  }  false {    yylval.bval = 0;    return BOOL;  }  "<=" {return LEEQ;}  ">=" {return GREQ;}  "==" {return EQ;}  "!=" {return NOTEQ;}  "<" {return LESS;}  ">" {return GREAT;}  \"[^\"]\*\" {    size\_t size = strlen(yytext)-2;//""の分小さく    const char\* charsHead = &yytext[1];    char\* characters = (char\*)strndup(charsHead, size);    yylval.charPtr = characters;    return CHARACTERS;  }  ([0-9]+|([0-9]\*\.[0-9]+)([eE][-+]?[0-9]+)?) {    yylval.dval = atof(yytext);    return NUMBER;  }  [ \t] ;  [a-zA-Z][a-zA-Z0-9]\*  {    struct symbol\* symbolPtr = lookSymbol(yytext);    yylval.symbolPtr = symbolPtr;    switch(symbolPtr->type){      case numeric:    return NUMERICNAME;      case boolean:    return BOOLEANNAME;      case characters: return CHARACTERSNAME;      case function:   return FUNCNAME;      default: return NAME;    }  }  "$" {return 0;}  \n |  . return yytext[0];  %%  void yyerror(char\* errmsg){    //stderrはエラー出力    fprintf(stderr, "ERROR: %s at %s\n", errmsg, yytext);  } |

コード 3 report.y

|  |
| --- |
| %{  #include <stdio.h>  #include <stdlib.h>  #include "report.h"  #include <string.h>  #include <math.h>  extern void yyerror(char\* errmsg);  //関数が扱える引数の上限数  #define MAXARGNUM 3  //引数の一時格納用のバッファ  double argList[MAXARGNUM];  //現在詰められている引数の数  short argNum = 0;  int statement\_state = 1;  %}  %union{    double dval;    struct symbol\* symbolPtr;    int bval;    char\* charPtr;  }  %token <symbolPtr> NAME  %token <symbolPtr> BOOLEANNAME  %token <symbolPtr> NUMERICNAME  %token <symbolPtr> CHARACTERSNAME  %token <symbolPtr> FUNCNAME  %token <dval> NUMBER  %token <bval> BOOL  %token <charPtr> CHARACTERS  %token IF ENDIF  %left LESS GREAT LEEQ GREQ EQ NOTEQ  %left '+' '-'  %left '\*' '/'  %nonassoc UMINUS  %type <dval> expression  %type <bval> logical\_expression  %%  statement\_list:    statement                {}  | statement\_list statement {}  | statement\_list '\n'      {statement\_state = 1;}  ;  statement:    NAME           '=' expression ':'          {$1->value = $3; $1->type = numeric;}  | NUMERICNAME    '=' expression ':'          {$1->value = $3; $1->type = numeric;}  | NAME           '=' logical\_expression ':'  {$1->value = $3; $1->type = boolean;}  | BOOLEANNAME    '=' logical\_expression ':'  {$1->value = $3; $1->type = boolean;}  | NAME           '=' CHARACTERS ':'          {$1->charPtr = $3; $1->type = characters;}  | CHARACTERSNAME '=' CHARACTERS ':'          {$1->charPtr = $3; $1->type = characters;}  | CHARACTERSNAME ':'                         {printf("= %s\n", $1->charPtr);}  | expression ':'                             {printf("= %g\n", $1);}  | logical\_expression ':'                     {                                                  if($1 == 1) printf("statement : true\n");                                                  else printf("statement : false\n");                                               }  | ifstatement                              {;}  | endifstatement                           {;}  ;  expression\_list:    expression                     {argList[argNum++] = $1;}  | expression\_list ',' expression {argList[argNum++] = $3;}  ;  endifstatement:    ENDIF { statement\_state = 1;}  ;  ifstatement:    IF '(' logical\_expression ')'    {      if($3 == 1) statement\_state = 1; else statement\_state = 0;    }  ;  logical\_expression:    BOOL                           {$$ = $1;}  | BOOLEANNAME                    {$$ = $1->value;}  | expression LESS expression     {$$ = $1 < $3;}  | expression GREAT expression    {$$ = $1 > $3;}  | expression LEEQ expression     {$$ = $1 <= $3;}  | expression GREQ expression     {$$ = $1 >= $3;}  | expression EQ expression       {$$ = $1 == $3;}  | expression NOTEQ expression    {$$ = $1 != $3;}  ;  expression:    expression '+' expression   { $$ = $1 + $3;}  | expression '-' expression   { $$ = $1 - $3;}  | expression '\*' expression   { $$ = $1 \* $3;}  | expression '/' expression   { if($3 == 0.0) yyerror("Devide by zero"); else $$ = $1 / $3;}  | '-' expression %prec UMINUS { $$ = -$2;}  | '(' expression ')'          { $$ = $2;}  | NUMBER                      { $$ = $1;}  | NUMERICNAME                        { $$ = $1->value;}  | FUNCNAME '(' expression\_list ')'{                                      if($1 -> funcptr){                                        //MAXARGNUMがいくつかに依ってここの引数の記述は変わる。                                        if(statement\_state==1){                                          $$ = ($1->funcptr)(argList[0], argList[1], argList[2]);                                          argNum = 0;                                        }                                      }else{                                        printf("%s not a function.\n", $1->name);                                      }                                    }  ;  %%  /\*変数名nameの変数をテーブルから取得（無ければ追加）\*/  struct symbol\* lookSymbol(char\* name){    struct symbol\* ptr;    for(ptr = symbolTable; ptr < &symbolTable[SYMBOLNUM]; ptr++){      if(ptr->name && !strcmp(ptr->name, name)) return ptr;      /\*見つからずに末尾まで到達したら追加\*/      if(!ptr->name){        ptr->name = strdup(name);        ptr->type = undefined;        return ptr;      }    }    yyerror("Too many symbols");    exit(1);  }  void addfunc(char\* name, double (\*func)()){    struct symbol\* ptr = lookSymbol(name);    ptr->funcptr = func;    ptr->type = function;  }  double triangle(double a, double h);  double trapezoid(double a, double b, double h);  int main(){    extern double sqrt(), exp(), log(), sin(), cos(), pow();    double triangle(), trapezoid();    addfunc("sqrt", sqrt);    addfunc("exp",  exp);    addfunc("log",  log);    addfunc("sin",  sin);    addfunc("cos",  cos);    addfunc("pow",  pow);    addfunc("triangle",  triangle);    addfunc("trapezoid",  trapezoid);    yyparse();    return 0;  }  double triangle(double a, double h){    return a\*h/2.0;  }  double trapezoid(double a, double b, double h){    return (a+b)\*h/2.0;  } |

# 使い方

Ch3-05系との差異に集中して使い方を解説する。

## 全体を通して共通する文法

文末には必ず : が必要。

if文  
if文の文法は以下。

|  |
| --- |
| if(真偽値) 文or複文 endif |

ifからendifがCにおけるif文のブロックに相当する。endifを過ぎれば真偽値に依存しない。endifは書かなくても良いがその場合改行時にendifが書かれている場合と同等の処理になる。

## 文字列

文字列リテラルは以下のように””で囲む必要がある。

|  |
| --- |
| “some words |

# 実行結果

|  |
| --- |
| $ ./report.exe  a=10:b=2:  p=pow(a,b):  p:  = 100  boolValue=true:  str="initStr":  if(boolValue)str="secondStr":  str:  = secondStr  boolValue=1>2:  if(boolValue)str="thirdStr":  str:  = secondStr |

# 解決アプローチ

## 引数を２つ以上扱える数学的関数を取り扱えるようにする。

「引数を２つ以上」とあるが、現実的な数であれば引数が何個でも扱えるように対応させるのがベストだと考えたのでひとまずそれを目指して考えてみた。しかし関数を呼び出す際には必ず例えば以下のような関数ポインタからコールするしかない。

$$ = ($1->funcptr)(/\*引数\*/);

したがって、/\*引数\*/の部分に固定個数の引数を表す変数を記述するほかないという結論に至った。この部分を、実行する関数が必要とする引数の数に応じて可変に変わるような仕組みにしたかったが、$1に必要とする引数の数を表す変数があったとしても上記の記述の部分を変えることは出来ないので諦めた。アセンブリ言語のコード生成まで実装するなら、引数に対応するレジスタ領域に連続するように引数を代入するループ処理を作れると考えたが、コード生成まで対応するのは学習・実装時間的に現実的ではないと思ったのでこれも諦めた。

ここまでで扱える引数の数に上限を設けることは決定した。この上限の数を後から変えたいという時の為になるべく簡単に上限の数の変更に対応できる仕組みを考えることに集中した。グローバルな変数として引数を格納するargList配列をよういした。この配列の要素数が扱える引数の上限になるということである。

//関数が扱える引数の上限数

#define MAXARGNUM 3

//引数の一時格納用のバッファ

double argList[MAXARGNUM];

//現在詰められている引数の数

short argNum = 0;

関数をコールする処理は以下の用になる。

$$ = ($1->funcptr)(argList[0], argList[1], argList[2]);

つまり、扱える引数の上限の数を変えたい場合はMAXARGNUMの数とコールするコードを変えれば変えられるという状態にできた。コールする部分のコードもプリプロセスで自動にMAXARGNUMに応じて記述がされるようにすれば、MAXARGNUMの修正のみで扱える引数の上限を修正できるようになると考えたが、これは可読性が著しく低くなる事を懸念してやらなかった。

argListに実引数を詰めていく処理は以下のようなコードで実現している。

expression\_list:

  expression {argList[argNum++] = $1;}

| expression\_list ',' expression {argList[argNum++] = $3;}

;

expression\_list記号はexpressionをカンマ区切りで並べたトークン列を意味している。argNum変数に現在配列に詰められている実引数の数を表しているので、一つ詰めるごとに自動的に代入する際のインデックスがずれていく。なのでこの部分については扱える引数の上限に依存しない状態にできた。

関数のコールの文法規則はexpression\_listの導入によって以下のようになった。

expression:

/\*省略\*/

| FUNCNAME '(' expression\_list ')'

{

  if($1 -> funcptr){

    //MAXARGNUMがいくつかに依ってここの引数の記述は変わる。

    if(statement\_state==1){

      $$ = ($1->funcptr)(argList[0], argList[1], argList[2]);

      argNum = 0;

    }

  }else{

    printf("%s not a function.\n", $1->name);

  }

}

関数をコールした時点でargNumを0にリセットして実引数をリセットしてる。

これまでと同様にaddFuncで関数を登録すれば課題内容にある関数を実行できるようにした。

extern double sqrt(), exp(), log(), sin(), cos(), pow();

  double triangle(), trapezoid();

  addfunc("sqrt", sqrt);

  addfunc("exp",  exp);

  addfunc("log",  log);

  addfunc("sin",  sin);

  addfunc("cos",  cos);

  addfunc("pow",  pow);

  addfunc("triangle",  triangle);

  addfunc("trapezoid",  trapezoid);

## 実行文を:で区切って、１行に複数の文を記述する

複文に対応するにあたって、文の末尾には必ず:を記述しなければならないという仕様にした。複文対応をする前の文法規則を踏襲すると、入力された文字列の末尾の文は:がついていてもいいしついていなくてもいいという事になる。この文法規則だとyaccのコードが両方の状態に対応する為に冗長になる。さらにユーザーにとってもメリットもほとんどないので末尾の文でも:は必須とした。冗長な文法規則をなくした分。文法規則の定義はシンプルに以下のようになった。ifstatemenとendifstatementについては次節で説明する。

statement:

  NAME           '=' expression ':'        {$1->value = $3; $1->type = numeric;}

| NUMERICNAME    '=' expression ':'        {$1->value = $3; $1->type = numeric;}

| NAME           '=' logical\_expression ':'{$1->value = $3; $1->type = boolean;}

| BOOLEANNAME    '=' logical\_expression ':'{$1->value = $3; $1->type = boolean;}

| NAME           '=' CHARACTERS ':'        {$1->charPtr = $3; $1->type = characters;}

| CHARACTERSNAME '=' CHARACTERS ':'        {$1->charPtr = $3; $1->type = characters;}

| CHARACTERSNAME ':'                       {printf("= %s\n", $1->charPtr);}

| expression ':'                          　{printf("= %g\n", $1);}

| logical\_expression ':'                {

                                          if(statement\_state==1){

                                            if($1 == 1) printf("statement : true\n");

                                           else printf("statement : false\n");

                                          }

                                        }

| ifstatement                           {;}

| endifstatement                        {;}

;

## if 文を使えるようにする。

### 真偽値の導入

if文を導入するにあたって、真偽値も扱うことになるので真偽値に関係する部分もまとめて解説する。まずトークン定義に以下の記述を追加した。特筆することはなく予約語と関係演算子の類を定義している。endifについては後述する。

if {

  return IF;

}

endif {

  return ENDIF;

}

true {

  yylval.bval = 1;

  return BOOL;

}

false {

  yylval.bval = 0;

  return BOOL;

}

"<=" {return LEEQ;}

">=" {return GREQ;}

"==" {return EQ;}

"!=" {return NOTEQ;}

"<" {return LESS;}

">" {return GREAT;}

真偽値を表現するシンボルであるlogical\_expressionは以下のように定義した。

logical\_expression:

  BOOL                           {$$ = $1;}

| BOOLEANNAME                    {$$ = $1->value;}

| expression LESS expression     {$$ = $1 < $3;}

| expression GREAT expression    {$$ = $1 > $3;}

| expression LEEQ expression     {$$ = $1 <= $3;}

| expression GREQ expression     {$$ = $1 >= $3;}

| expression EQ expression       {$$ = $1 == $3;}

| expression NOTEQ expression    {$$ = $1 != $3;}

;

これらの文法規則を上から順に説明すると、予約語として定義したtrue/falseがBOOLトークンとしてyaccに入力されそのままlogical\_expressionになる。元のプログラム（ch3-0-5系）では変数をNAMEトークンで統一していたが、真偽値や文字列リテラルが導入されたので変数を型で分けた。　真偽値を値に持つ変数をBOOLNAMEトークンとして扱えるようにしてあるので、既に真偽値の変数があればこれもlogical\_expressionになる。他は数値同士の大小比較などの関係演算子による演算に対応している。

### if文の導入

まず課題内容の記述にあるif文の例では文法規則は以下のようになると考えた。

IF (logical\_expression) statement

しかし、この文法規則が成り立った時点においては既にstatementが成り立っているので、statementを構成する文法のアクションが実行されてしまっている。つまり、真偽値がfalseになる場合には処理を実行しないという仕組みを実行し得ないと気づいた。最終的には次のような文法規則でifstatementを定義した。

ifstatement:

  IF '(' logical\_expression ')'

  {

    if(statement\_state==1){

      if($3 == 1) statement\_state = 1; else statement\_state = 0;

    }

  }

;

文法規則にstatementを含めないことによって、statementを構成する文法のアクションが実行される前にif文の真偽値に依存した処理を実行できるようにした。statement\_stateはグローバルな変数として定義されていて、ほぼすべてのアクションはstatement\_stateが1である場合のみアクションを実行するように書き換えた。つまりif文に後続する処理は全て、直前のif文のlogical\_expressionの値によって処理するorしないで分岐することになる。このままだとlogical\_exoressionがfalseであるif文が一度登場するとstatement\_stateの値をtureに戻すすべがなくなるので、Cのプリプロセスを踏襲してendifにあたる文法を導入した。

endifstatement:

  ENDIF { statement\_state = 1;}

;

単純にendifと入力すればstatement\_stateを1にリセットできる。これによって直前のif文による影響を受けない状態に戻すことができる。If(logical\_expression) からendifまでの記述がif文のブロックにあたるということになる。

## (4) 変数に文字列を代入できるようにする。

文字列リテラルを表現するトークンをCHARACTERSとして以下のように定義した。

\"[^\"]\*\" {

  size\_t size = strlen(yytext)-2;//""の分小さく

  const char\* charsHead = &yytext[1];

  char\* characters = (char\*)strndup(charsHead, size);

  yylval.charPtr = characters;

  return CHARACTERS;

}

“some words”のように”で囲まれた範囲が文字列としてyylvalに含まれる。

yaccには以下の記述を追加した。

statement:

…省略…

| NAME           '=' CHARACTERS ':'      　{ $1->charPtr = $3; $1->type = characters;}

| CHARACTERSNAME '=' CHARACTERS ':'       { $1->charPtr = $3;}

| CHARACTERSNAME ':'                      { printf("= %s\n", $1->charPtr);}

…省略…

;

文字列を変数に代入する文法と、文字列を出力する文法が追加されている。CHARACTERSNAMEは文字列が代入されている変数を意味している。

# 考察

作ったプログラムの問題点やその解決アプローチなどを考察として記述する。

## 関数の入れ子

実装した関数の仕組みだとfunc1(1, func2(), 0)のような関数の返り値をそのまま引数に使用する記述に対応できないという問題がある。argListと同等の役割を担う構造体を作成して、その構造体のlinked listを実装すればこの問題に対応できる。

## if文の問題点

実装したif文の仕組みだと、statement\_stateを使っているために各アクションにif(statement\_state){}の記述をせざるを得ない。コードを読む側としては非常に冗長でアクションの使い方としても不適切に思える。

他にもifとendifでブロックを再現しているが、Cにおけるif(b1){ if(b2)... ; }のようなif文の入れ子にも対応できない。これもstatement\_staeをスタック化してブロックに入るたびにpushしてブロックから出る度にpopしてブロックに応じて適切なstatement\_stateを参照する仕組みにすれば解決する。

## テーブルの可変長化

シンボルテーブルも静的な固定長配列として定義されているが、実際にはいくつの変数や関数が入力されるか分からないので可変長な配列にするべきである。Lihnked listにすればこの問題も解決する。