J5実験 第2回

第1回で，基本的なLexとYaccの使い方を学びました．今回はそれを発展させて，

- 電卓
- 単純なプログラミング言語のインタプリタ（解釈実行器）

を作りましょう．

忘れてしまったところは，前回の授業資料 1.ipynb を参照してください．

電卓（1）

簡易的な電卓

前回は最後に，以下のような Yacc のプログラムを作成しました．

```
%token NUM;
%%
expr : NUM              { printf("NUM: %s\n", yytext); }
     | expr '+' NUM     { printf("ADD: %s\n", yytext); }
     ;
```

構文解析して，構文規則に当てはまったらLexの該当した値を出力するプログラムでしたね． 各構文規則が認識できているので，これを値として計算するのは簡単です．

```
%token NUM;
%%
expr : NUM              { value = atoi(yytext); }
     | expr '+' NUM     { value = value + atoi(yytext); }
     ;
```

各構文規則の右側の `{}` で囲まれたアクションの部分に注目してください． 数が出現したときに， `value` というCプログラム上の変数に覚えておきます．Lexが読み込んだ `yytext` はあくまで文字列としての数字の並びですので，Cの関数 `atoi(char*)` を使って，整数に変換しています．足し算があるごとに， `value` の値を更新していけば，電卓になるわけです． atoiの仕様

それでは，実際に実行してみましょう．

In [ ]:
```
/* yacc ex2-1.yacc */
%token NUM;
%%
expr : NUM              { value = atoi(yytext); }
     | expr '+' NUM     { value = value + atoi(yytext); }
     ;
```

In [ ]:
```
/* lex ex2-1.lex */
digit     [0-9]
white     [\n\t ]
%%
{digit}+  { return NUM; }
"+"       { return '+'; }
{white}   { ; }
```

In [ ]:
```
/* c ex2-1.c */
#include <stdio.h>
extern char *yytext;

int value;

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  printf("%d\n", value);
  return 0;
}
```

In [ ]:
```
/* a.out */
1+2+3
```

ちゃんと足し算の結果が出ていることが分かります．

以下のように，第1回の問題5のように，構文規則を変えるとどうなるでしょうか．

In [ ]:
```
/* yacc p1-5.yacc */
%token NUM;
%%
expr : NUM              { value = atoi(yytext); }
     | NUM '+' expr     { value = value + atoi(yytext); }
     ;
```

（Cのコードを再コンパイルするのを忘れないように）

In [ ]:
```
/* a.out */
1+2+3
```

`atoi()` は整数に変換できないときに `0` を返します．ですので，前回の出席課題の問題5で考えてもらったように，この構文で `yytext` を使ってしまうと，どんな式を与えても `value` は `0` になってしまいます．そのため，上のような実行結果になったのでした．

このような構文は，左辺の `expr` が右辺の右側で再帰的に出現しているので，***右再帰*** の構文という風に呼ばれます． ここでは思った電卓の計算結果が出ませんでしたが，右再帰の構文が一概に悪いというわけではなく，ここで `yytext` を使っていたのが良くなかったということになりますので気をつけましょう（この問題を解決する属性値という仕組みについては後述）．ここで，Yaccから生成された構文解析器のつもりになってみると，右再帰の構文の場合には，入力の文字列（上の例だと `1+2+3` ）を読んでいくときに，読み込み済みの箇所の結果をスタックのような構造で覚えておかないといけなくなってしまいます．

ついでに，以下のような構文はどうすればよいでしょう．

```
E ::= E + E | NUM
```

この構文に対して，右辺の `E` を区別して書いて `E ::= E_a + E_b | NUM` とすると， `1+2+3` は，

```
E_a → 1 + 2, E_b → 3
```

と解析することもできますし，

```
E_a → 1, E_b → 2 + 3
```

と解析する事もできてしまいます． そのため，上のような `E ::= E + E | NUM` という構文は***あいまいな構文***と言われます．

以上のように，構文規則を書くにしても，色々な書き方があるわけで，気をつけて書かないといけないという事です．

Yaccが生成する構文解析器

ここまでちゃんと意識してこなかったですが，Yaccは「左（L）から順に入力を読みとっていき，右端導出（R）を行う」解析手法（***LR構文解析***）を採用した構文解析器を生成します． 右端導出とは，「右端の非終端記号から順に構文規則を適用していく方法」です． より厳密には，Yaccから生成される構文解析器は，***LALR構文解析***（***Lookahead LR法***）を使って構文解析をしています．Lookaheadというのは「入力文字を先読みする」という事です．

上で出てきたあいまいな構文に対して，YaccがLR構文解析をしようとしたときに，

- `shift/reduce conflict`
- `reduce/reduce conflict`

が起こる可能性があります． （前回「Lex と Yacc のデバッグ」の節で出てきた，ShiftとReduceの話です）

`shift/reduce conflict` とは、構文規則が Shift（つまり，もっと読み込んでさらに長い非終端記号に Reduce（還元）できるようにする）か、Reduce（そこで非終端記号にしてしまう）かの2つの解釈ができてしまうということを意味しています． `reduce/reduce conflict` とは，同時に Reduce できる構文規則が複数あるということです（Yaccの場合には先に表われたほうを優先するが，望ましい構文ではない）．

ここでは，詳しい説明は書きませんが（詳しいことを調べるのは，レポート課題としましょう！），我々が書いた構文規則にどうマッチするか確認する `y.tab.c` のプログラムがどのように入力文字を処理しているか（つまりLALR構文解析がどう実装できるか）を知っておくことも重要だと思います． （右再帰と左再帰で，どちらがYaccにとって良いかというのも分かるはず．）

電卓（2）

先の簡易的な電卓では，足し算の規則にマッチングしてCプログラム上の変数 `value` に計算結果を保存していました．それだけだと不便な事もあるので，入力された式1行ごとに計算するように変えてみましょう．

```
%token NUM;
%%
exprlist:
        | exprlist expr '\n'    { printf("%d\n", value); }
        ;

expr    :
       | NUM                    { value = atoi(yytext); }
       | expr '+' NUM           { value = value + atoi(yytext); }
       ;
```

この構文は「式の並び（ `exprlist` ）とは，空っぽであるか，または式の並びの後に式と改行が続いたもの」と読むことができます（空っぽの行があることに注意しましょう）． これで，各行毎に `value` の値を出力すれば，計算結果の値を表示することができます．

実際に実行してみましょう．

In [ ]:
```
/* lex ex2-2.lex */
digit        [0-9]
white        [\t ]
%%
{digit}+     { return NUM; }
"+"          { return '+'; }
"\n"         { return '\n'; }
{white}      { ; }
```

前の例と比べて， `white` から `\n` を抜いていることに注目してください．その代わりに `\n` のパターンが追加されています．

In [ ]:
```
/* yacc ex2-2.yacc */
%token NUM;
%%
exprlist:
        | exprlist expr '\n'    { printf("%d\n", value); }
        ;

expr    :
        | NUM                   { value = atoi(yytext); }
        | expr '+' NUM          { value = value + atoi(yytext); }
        ;
```

In [ ]:
```
/* c ex2-2.c */
#include <stdio.h>
extern char *yytext;

int value;

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

In [ ]:
```
/* a.out */
1 + 2 + 3
3 + 82
9 + 2
```

記号の属性値

`yytext` の値を `value` に保存してしまうと不都合があるということを前述しました．これをCプログラム内で適宜保存する先を変えて配列やスタックのような構造を使って解決することも可能ではあるのですが，Yaccの「各非終端記号に**属性値**をつける機能」を使ってみましょう．ここでは，簡単のために整数値が1個だけということにしておきましょう．

たとえば，括弧に対応した電卓の構文を考えると，以下のように属性値を使ったアクションを書くことができます．

```
%token NUM;
%%
exprlist:
        | exprlist expr '\n'    { printf("%d\n", $2); }
        ;

expr    : prim                  { $$ = $1; }
        | expr '+' prim         { $$ = $1 + $3; }
        | expr '*' prim         { $$ = $1 * $3; }
        ;
prim    : NUM                   { $$ = atoi(yytext); }
        | '(' expr ')'          { $$ = $2; }
        ;
```

ここで `$$` は「左辺の非終端記号の属性値」を表し，`$n` は「左辺の `n` 番目の記号の属性値」を表します（`n` は1から始まる）．つまり `exprlist` の構文において，2つ目の構文規則での `expr` は2番目の記号なので `$2` を `printf` することで計算結果を表示できます．また，足し算の結果も `'+'` の左右の `expr` と `prim` の値を足し算して式の値（`$$`）とすれば良いということです． それでは，実際に実行してみましょう．

In [ ]:
```
/* yacc ex2-3.yacc */
%token NUM;
%%
exprlist:
        | exprlist expr '\n'    { printf("%d\n", $2); }
        ;

expr    : prim                  { $$ = $1; }
        | expr '+' prim         { $$ = $1 + $3; }
        | expr '*' prim         { $$ = $1 * $3; }
        ;
prim    : NUM                   { $$ = atoi(yytext); }
        | '(' expr ')'          { $$ = $2; }
        ;
```

Lexは `*` と `(` と `)` を使えるようにすれば，ほぼそのままです．

In [ ]:
```
/* lex ex2-3.lex */
digit        [0-9]
white        [\t ]
%%
{digit}+     { return NUM; }
[+*()]       { return yytext[0]; }
"\n"         { return '\n'; }
{white}      { ; }
```

記号1文字に一度にマッチするパターンを書きました．前回のLexのパターンの書き方を思い出してください．マッチさせたい記号であれば，その文字が `yytext` の最初に入るのでそれを返せばまとめて扱えます．

Cプログラムからは `value` が不要になります．

In [ ]:
```
/* c ex2-3.c */
#include <stdio.h>
extern char *yytext;

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

括弧が使えているか試してみましょう．

In [ ]:
```
/* a.out */
(1+2)*(3+1)
1+2*3+1
```

2つ目の結果は想像していたものと違うはずです． 今日の練習問題でそれを直してみましょう．

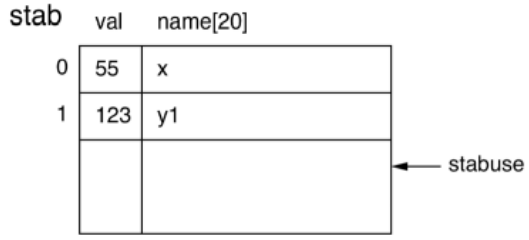
名前と記号表

電卓（2）までで計算はできるようになったので，「変数」を導入してみましょう．

```
z = 1 + 2
```

のようにして，計算結果を変数に覚えておけるような機能を電卓につけてみるということです．

簡単化のために，各変数名の長さは20が上限で，変数は100個までとして，以下のようなデータ構造で考えてみましょう，



この図では，`55`という値を`x`に対応づけ，`123`を`y1`に対応づけていることになります．これをC言語で実装すると，以下のようなコードになります．

```c
struct stab {
  int val;
  char name[20];
} stab[100];
int stabuse = 0;
```

表全体は`stab`という名前で，`val`と`name`の領域を保持しています．それを`struct stab`型が100個並んだ配列としています．どこまで使っているかというのを`stabuse`という変数に入れておくことにします．

この変数表に対する処理を行う，変数を探す関数`lookup`を以下のように定義しておきましょう．

```c
// 引数にとる文字列が表にあれば見つかった位置を返して，なければ追加する関数
int lookup(char *s) {
  int i;
  // 表の使っているところを順に見ていって
  for (i = 0; i < stabuse; ++i) {
    if (strcmp(stab[i].name, s) == 0) {
      // 見つかった場合はその位置を返す
      return i;
    }
  }
  if (stabuse >= 99) {
    // 変数の上限の個数を超えてしまった
    printf("table overflow\n");
    exit(1);
  }

  // 表になかったので追加する
  strcpy(stab[stabuse].name, s);
  return stabuse++;  // 追加した位置を返す
}
```

入力の各行が代入文として，代入された値を表示するような仕様だとすると，以下のようなYaccのプログラムになります．

```
   %token NUM;
   %token IDENT;
   %%
   stlist :
          | stlist stat '\n'
          ;
   stat   : var '=' expr      { stab[$1].val = $3; printf("%d\n", $3); }
          ;
   expr   : prim              { $$ = $1; }
          | expr '+' prim     { $$ = $1 + $3; }
          | expr '-' prim     { $$ = $1 - $3; }
          ;
   prim   : NUM               { $$ = atoi(yytext); }
          | var               { $$ = stab[$1].val; }
          | '(' expr ')'      { $$ = $2; }
          ;
   var    : IDENT             { $$ = lookup(yytext); }
          ;
```

- 代入文は，式の値（`expr`の属性値`$3`）を左辺の変数値（`var`の属性値`$1`に対応する記号表の場所の`val`欄）に入れ`printf`で表示します．
- 式に現れる変数は，その値（正確にはその変数に対応する記号表の場所（varの属性値`$1`）の`val`欄）を値とします．
  - その際には，まずさきほどCプログラムで定義した`lookup`を使って位置を求めておくことになります．

`IDENT`を使えるようにするために，Lexの定義も見直しましょう．

```
alpha   [a-zA-Z]
digit   [0-9]
white   [\t ]
%%
{alpha}({alpha}|{digit})*   { return IDENT; }
{digit}+                    { return NUM; }
[\n+\-()=]                  { return yytext[0]; }
{white}                     { ; }
```

ここまで理解できたでしょうか．理解できたら，ここまでのプログラム群を実行して動作を確認してみましょう．

In [ ]:
```
/* lex ex2-4.lex */
alpha   [a-zA-Z]
digit   [0-9]
white   [\t ]
%%
{alpha}({alpha}|{digit})*   { return IDENT; }
{digit}+                    { return NUM; }
[\n+\-()=]                  { return yytext[0]; }
{white}                     { ; }
```

In [ ]:
```
/* yacc ex2-4.yacc */
%token NUM;
%token IDENT;
%%
stlist :
       | stlist stat '\n'
       ;
stat   : var '=' expr      { stab[$1].val = $3; printf("%d\n", $3); }
       ;
expr   : prim              { $$ = $1; }
       | expr '+' prim     { $$ = $1 + $3; }
       | expr '-' prim     { $$ = $1 - $3; }
       ;
prim   : NUM               { $$ = atoi(yytext); }
       | var               { $$ = stab[$1].val; }
       | '(' expr ')'      { $$ = $2; }
       ;
var    : IDENT             { $$ = lookup(yytext); }
       ;
```

In [ ]:
```
/* c ex2-4.c */
#include <stdio.h>
struct stab {
  int val;
  char name[20];
} stab[100];
int stabuse = 0;

extern char *yytext;

// 引数にとる文字列が表にあれば見つかった位置を返して，なければ追加する関数
int lookup(char *s) {
  int i;
  // 表の使っているところを順に見ていって
  for (i = 0; i < stabuse; ++i) {
    if (strcmp(stab[i].name, s) == 0) {
      // 見つかった場合はその位置を返す
      return i;
    }
  }
  if (stabuse >= 99) {
      // 変数の上限の個数を超えてしまった
      printf("table overflow,\n");
      exit(1);
    }

  // 表になかったので追加する
  strcpy(stab[stabuse].name, s);
  return stabuse++;  // 追加した位置を返す
}
```

```
#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

In [ ]:
```
/* a.out */
z = 1 + 4
x = z - 2 + z
```

単純なプログラミング言語のインタプリタ（1）

代入文を持った電卓ができたので，それをプログラミング言語っぽくユーザーの入力を受け取れるようにしてみましょう．

In [ ]:
```
/* yacc ex2-5.yacc */
%token NUM;
%token IDENT;
%token READ;
%token PRINT;
%%
prog   : IDENT '{' stlist '}'       { return $3; }
       ;
stlist :
       | stlist stat
       ;
stat   : var '=' expr ';'           { stab[$1].val = $3; }
       | READ var ';'               { scanf("%d", &stab[$2].val); }
       | PRINT expr ';'             { printf("%d\n", $2); }
       ;
expr   : prim              { $$ = $1; }
       | expr '+' prim     { $$ = $1 + $3; }
       | expr '-' prim     { $$ = $1 - $3; }
       ;
prim   : NUM               { $$ = atoi(yytext); }
       | var               { $$ = stab[$1].val; }
       | '(' expr ')'      { $$ = $2; }
       ;
var    : IDENT             { $$ = lookup(yytext); }
       ;
```

`read 変数` と `print 変数` という形式でユーザー（プログラムを使う人）が変数を画面に入出力できるようになります．（ただし，今の場合にはプログラムを書く人と使う人が同じになってしまうことは，後で実行してみれば分かります．） また，各文を行ではなくて `;` で区切るようにもしてあります．

Lexのプログラムは以下のようになります．

In [ ]:
```
/* lex ex2-5.lex */
alpha   [a-zA-Z]
digit   [0-9]
white   [\n\t ]
%%
read                        { return READ; }
print                       { return PRINT; }
{alpha}({alpha}|{digit})*   { return IDENT; }
{digit}+                    { return NUM; }
[+\-=();{}]                 { return yytext[0]; }
{white}                     { ; }
```

Cプログラムはそのままで良いですね．

In [ ]:
```
/* c ex2-5.c */
#include <stdio.h>
struct stab {
  int val;
  char name[20];
} stab[100];
int stabuse = 0;

extern char *yytext;

// 引数にとる文字列が表にあれば見つかった位置を返して，なければ追加する関数
int lookup(char *s) {
  int i;
  // 表の使っているところを順に見ていって
  for (i = 0; i < stabuse; ++i) {
    if (strcmp(stab[i].name, s) == 0) {
      // 見つかった場合はその位置を返す
      return i;
    }
```

```
  }
  if (stabuse >= 99) {
    // 変数の上限の個数を超えてしまった
    printf("table overflow,\n");
    exit(1);
  }

  // 表になかったので追加する
  strcpy(stab[stabuse].name, s);
  return stabuse++;  // 追加した位置を返す
}

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

それでは，この自前のプログラミング言語のプログラムを書いて実行してみましょう．

In [ ]:
```
/* a.out */
main {
  x = 1 + 2;
  print x;
}
```

In [ ]:
```
/* a.out */
main {
  read x;
10
  y = x + 1;
  print y;
}
```

`read` 文を打ち込んだ直後に，その入力を打たないといけないというのが普通のプログラミング言語とは異なりますが，プログラムを解釈してすぐに実行するようなインタプリタを作ることができました．

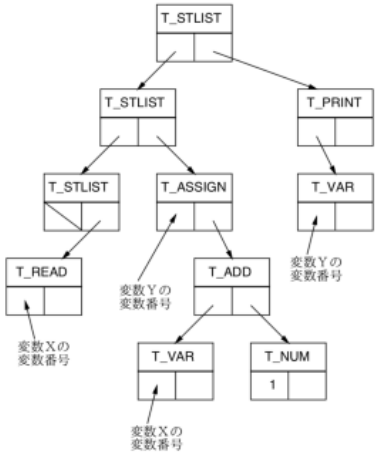
単純なプログラミング言語のインタプリタ（2）

上の例でプログラムを解釈してすぐに実行してしまうのは，Yaccのアクションでいきなり計算を実行してしまうからです． 普通のプログラミング言語処理系であれば，まずプログラムの構造に対応した中間形式を作成し，それを元に実行します． コンパイラであれば，実行する代わりに別のアセンブリなどのコードを生成します．（コード生成は第3回の内容です．）

ここでは，字句・構文解析した結果を木構造（抽象構文木）として表現して，実行するようにしてみましょう． たとえば，上の実行例

```
main {
  read x;
  y = x + 1;
  print y;
}
```

に対応した抽象構文木は，以下のようにできます．



`T_STLIST` や `T_PRINT` は，そのノード（節）がどの種類かを示すためのタグです．たとえば，`T_ASSIGN` の左の子に変数番号が直接入るようにここでは決めてしまいましたが，ノードをどう表現するかは自由に決められます． どういう表現するようにしたにしろ，この抽象構文木に，上の実行例のプログラムの情報が全て含まれていることに注意してください．

最終的には，

1. 抽象構文木を作って，
2. 抽象構文木をたどって行って実行する

という2段階になるわけですが，まずは抽象構文木が正しくできているか確認しなくてはいけないので，2の代わりに作った抽象構文木を表示するようにしてみましょう．

抽象構文木のデータ構造は以下のようにC言語で定義できます． 木構造はこれまでの授業で習ってきましたね．

```
typedef struct Node {
    int node_type;
    struct Node* left;
    struct Node* right;
} Node;
```

木構造を操作する操作も，普通に定義すればよいです．

```
// 新しいノードを作る関数
Node* createNode(int node_type, Node* left, Node* right) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->node_type = node_type;
    newNode->left = left;
    newNode->right = right;
    return newNode;
}
```

また，さきほどの抽象構文木の図にあった，それぞれのノードを示すnode_typeを以下のように定義しましょう．

```
#define T_STLIST 1
#define T_ASSIGN 2
#define T_READ   3
#define T_PRINT  4
#define T_ADD    5
#define T_SUB    6
#define T_NUM    7
#define T_VAR    8
```

それでは，前節「単純なプログラミング言語のインタプリタ（1）」のYaccのアクション部分で木構造を組み立てるようにしてみましょう．

In [ ]:

```
/* yacc ex2-6.yacc */
%union {
  Node* np;
  int i;
}
%type <np> stlist stat expr prim var
%token NUM;
%token IDENT;
%token READ;
%token PRINT;
%%
prog   : IDENT '{' stlist '}'       { traverse_tree($3); return 0; }
       ;
stlist :                            { $$ = NULL; }
       | stlist stat                { $$ = createNode(T_STLIST, $1, $2); }
       ;
stat   : var '=' expr ';'           { $$ = createNode(T_ASSIGN, $1, $3); }
       | READ var ';'               { $$ = createNode(T_READ, $2, NULL); }
       | PRINT expr ';'             { $$ = createNode(T_PRINT, $2, NULL); }
       ;
expr   : prim              { $$ = $1; }
       | expr '+' prim     { $$ = createNode(T_ADD, $1, $3); }
       | expr '-' prim     { $$ = createNode(T_SUB, $1, $3); }
       ;
prim   : NUM               { $$ = createNode(T_NUM, atoi(yytext), NULL); }
       | var               { $$ = createNode(T_VAR, $1, NULL); }
       | '(' expr ')'      { $$ = $2; }
       ;
var    : IDENT             { $$ = lookup(yytext); }
       ;
```

まずは，各構文規則のアクションに注目してください． 数値の計算や値の入出力をしていた具体的な処理の代わりに，上でCプログラムに用意しておいた `createNode` でノードを作成しています．属性値を使って木構造が作れているのが分かると思います． たとえば， `stat : var '=' expr ';'` という構文規則であれば， `stat` の属性値 `$$` に `createNode` で新たにノードを作成しています．そのときの引数は `node_type` として `T_ASSIGN` ，左の子に `$1` （つまり `var` の属性値，なので変数），右の子に `$3` （つまり `expr` の属性値，なので変数に入れる式）を指定してノードを作成することになります．図中の `y = x + 1;` に該当する箇所の木がこれで作成できるわけです． ここまで， `traverse_tree()` は説明していなかったですが，上に書いた「作った抽象構文木を表示する」関数になります．木構造を再帰的に表示するだけですので，難しくないCプログラムです．それを含めた以下のCプログラム全体で確認してください． 次に，Yaccの定義の先頭に `%union` と `%type` という箇所が増えていることに気付いたと思います． これは， `createNode` が `Node*` 型を返すために必要になった記述です．これ以前のYaccの例を思い出してほしいのですが， `$$` の型はこれまで気にしておらず，すべて `int` 型でした（前に戻って見直してみてください）．この部分は，Cプログラムになっていますので，Yaccが自動生成してくれる際にはちゃんと型を考えないといけません． つまり，これまではYaccのデフォルトの `int` 型であったので， 気にしなくてよかったものの，この例からは `Node*` 型として扱えないと，y.tab.cが生成されたときに不都合が起きてしまうということです．そのための記述が， `%union` の箇所で， `$$` はCの共用体で `Node*` か `int` ですという風に読みます（Cの共用体について分からなければ自分で調べてみること） ．次の `%type` は，非終端記号の型を定義しているもので，ここに挙げた `stlist` などは，今回 `Node*` 型を返すことにしたと分かると思います．

ここまで，コードの量も増えて，説明も複雑になってきましたが，順番に理解したら，実際に動かしてみましょう． 実行するときは Lex もちゃんと対応したものであることを忘れないように．

In [ ]:
```
/* lex ex2-6.lex */
alpha   [a-zA-Z]
digit   [0-9]
white   [\n\t ]
%%
read                        { return READ; }
print                       { return PRINT; }
{alpha}({alpha}|{digit})*   { return IDENT; }
{digit}+                    { return NUM; }
[+\-=();{}]                 { return yytext[0]; }
{white}                     { ; }
```

In [ ]:
```
/* c ex2-6.c */
#include <stdio.h>
struct stab {
  int val;
  char name[20];
} stab[100];
int stabuse = 0;

extern char *yytext;

#define T_STLIST 1
#define T_ASSIGN 2
#define T_READ   3
#define T_PRINT  4
#define T_ADD    5
#define T_SUB    6
#define T_NUM    7
#define T_VAR    8

typedef struct Node {
    int node_type;
    struct Node* left;
    struct Node* right;
} Node;


// 引数にとる文字列が表にあれば見つかった位置を返して，なければ追加する関数
int lookup(char *s) {
  int i;
  // 表の使っているところを順に見ていって
  for (i = 0; i < stabuse; ++i) {
    if (strcmp(stab[i].name, s) == 0) {
      // 見つかった場合はその位置を返す
      return i;
    }
  }
  if (stabuse >= 99) {
    // 変数の上限の個数を超えてしまった
    printf("table overflow,\n");
    exit(1);
  }

  // 表になかったので追加する
  strcpy(stab[stabuse].name, s);
  return stabuse++;  // 追加した位置を返す
}

// 新しいノードを作る関数
Node* createNode(int node_type, Node* left, Node* right) {
    Node* newNode = (Node*)malloc(sizeof(Node));
```

```
    newNode->node_type = node_type;
    newNode->left = left;
    newNode->right = right;
    return newNode;
}

// ノードをたどる関数
void traverse_tree(Node* node) {
  if (node == NULL) {
    return;
  }
  switch (node->node_type) {
    case T_STLIST:
      if (node->left != NULL) { traverse_tree(node->left); printf("; "); }
      traverse_tree(node->right);
      break;
    case T_ASSIGN:
      printf("v%d = ", node->left);
      traverse_tree(node->right);
      printf("\n");
      break;
    case T_READ:
      printf("read v%d\n", node->left);
      break;
    case T_PRINT:
      printf("print ");
      traverse_tree(node->left);
      printf("\n");
      break;
    case T_ADD:
      printf("(");
      traverse_tree(node->left);
      printf(" + ");
      traverse_tree(node->right);
      printf(")");
      break;
    case T_SUB:
      printf("(");
      traverse_tree(node->left);
      printf(" - ");
      traverse_tree(node->right);
      printf(")");
      break;
    case T_NUM:
      printf("%d", node->left);
      break;
    case T_VAR:
      printf("v%d", node->left);
      break;
  }
}

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

In [ ]:
```
/* a.out */
main {
  read x;
  y = x + 1;
  print y;
}
```

変数は変数番号（変数表のインデックス）が出ていて，図に示した木構造が（ちょっと木には見難いですが）ちゃんと出力されています．

単純なプログラミング言語のインタプリタ（3）

抽象構文木を使ってプログラムを実行する

抽象構文木が作成できるようになったので，抽象構文木をたどって実行するように変更してインタプリタを完成させましょう．

LexとYaccの定義はそのままで，Cの `traverse_tree` の代わりに実行すればよいですね．

In [ ]:
```
/* yacc ex2-7.yacc */
%union {
  Node* np;
```

```
  int i;
}
%type <np> stlist stat expr prim var
%token NUM;
%token IDENT;
%token READ;
%token PRINT;
%%
prog   : IDENT '{' stlist '}'       { dotree($3); return 0; }
       ;
stlist :                            { $$ = NULL; }
       | stlist stat                { $$ = createNode(T_STLIST, $1, $2); }
       ;
stat   : var '=' expr ';'           { $$ = createNode(T_ASSIGN, $1, $3); }
       | READ var ';'               { $$ = createNode(T_READ, $2, NULL); }
       | PRINT expr ';'             { $$ = createNode(T_PRINT, $2, NULL); }
       ;
expr   : prim              { $$ = $1; }
       | expr '+' prim     { $$ = createNode(T_ADD, $1, $3); }
       | expr '-' prim     { $$ = createNode(T_SUB, $1, $3); }
       ;
prim   : NUM               { $$ = createNode(T_NUM, atoi(yytext), NULL); }
       | var               { $$ = createNode(T_VAR, $1, NULL); }
       | '(' expr ')'      { $$ = $2; }
       ;
var    : IDENT             { $$ = lookup(yytext); }
       ;
```

In [ ]:

```
/* c ex2-7.c */
#include <stdio.h>
struct stab {
  int val;
  char name[20];
} stab[100];
int stabuse = 0;

extern char *yytext;

#define T_STLIST 1
#define T_ASSIGN 2
#define T_READ   3
#define T_PRINT  4
#define T_ADD    5
#define T_SUB    6
#define T_NUM    7
#define T_VAR    8

typedef struct Node {
    int node_type;
    struct Node* left;
    struct Node* right;
} Node;


// 引数にとる文字列が表にあれば見つかった位置を返して，なければ追加する関数
int lookup(char *s) {
  int i;
  // 表の使っているところを順に見ていって
  for (i = 0; i < stabuse; ++i) {
    if (strcmp(stab[i].name, s) == 0) {
      // 見つかった場合はその位置を返す
      return i;
    }
  }
  if (stabuse >= 99) {
    // 変数の上限の個数を超えてしまった
    printf("table overflow,\n");
    exit(1);
  }

  // 表になかったので追加する
  strcpy(stab[stabuse].name, s);
  return stabuse++;  // 追加した位置を返す
}

// 新しいノードを作る関数
Node* createNode(int node_type, Node* left, Node* right) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->node_type = node_type;
    newNode->left = left;
    newNode->right = right;
    return newNode;
}
```

```
// ノードをたどる関数
int dotree(Node* node) {
  if (node == NULL) {
    return 0;
  }
  switch (node->node_type) {
    case T_STLIST:
      if (node->left != NULL) { dotree(node->left); }
      dotree(node->right);
      break;
    case T_ASSIGN:
      stab[(int)(node->left)].val = dotree(node->right);
      break;
    case T_READ:
      scanf("%d", &stab[(int)(node->left)].val);
      break;
    case T_PRINT:
      printf("%d\n", dotree(node->left));
      break;
    case T_ADD:
      return dotree(node->left) + dotree(node->right);
    case T_SUB:
      return dotree(node->left) - dotree(node->right);
    case T_NUM:
      return node->left;
    case T_VAR:
      return stab[(int)(node->left)].val;
  }
}

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

In [ ]:
```
/* a.out */
main {
  read x;
  y = x + 1;
  print y;
}
20
```

全部プログラムを打ち終わった後で， `read` の入力を受けとって計算していることが分かります．

while文の追加

よりプログラミング言語らしくするために， `while` 文のループを追加してみましょう．

まず，Yaccを変更します． `stat` に `while` 文を追加します．

```
       |  WHILE '(' cond ')' stat       { $$ = createNode(T_WHILE, $3, $5); }
```

条件 `cond` が成り立っている間は `stat` を実行するという意味を考えましょう．条件 `cond` は以下のように定義できます．

```
   cond  : expr '<' expr          { $$ = createNode(T_LT, $1, $3); }
         | expr '>' expr          { $$ = createNode(T_GT, $1, $3); }
```

`stat` の定義を見直すと，1文を表わすだけなので，この構文定義では `while` 文のループ本体に1文しか書けないです． 無限ループを回避してループを実行できるようにするため， `変数++` の機能を追加して，まずは `print x++` のようなプログラムが動くようにしてみましょう． `prim` の定義に以下を追加します．

```
          | var PLUSPLUS          { $$ = createNode(T_PP, $1, NULL); }
```

これで準備ができましたので， 実行してみましょう． （どこが変更されているか，ちゃんと理解して実行すること．）

In [ ]:
```
/* yacc ex2-8.yacc */
%union {
  Node* np;
  int i;
```

```
}
%type <np> stlist stat expr prim var cond
%token NUM;
%token IDENT;
%token READ;
%token PRINT;
%token WHILE;
%token PLUSPLUS;
%%
prog   : IDENT '{' stlist '}'       { dotree($3); return 0; }
       ;
stlist :                            { $$ = NULL; }
       | stlist stat                { $$ = createNode(T_STLIST, $1, $2); }
       ;
stat   : var '=' expr ';'           { $$ = createNode(T_ASSIGN, $1, $3); }
       | READ var ';'               { $$ = createNode(T_READ, $2, NULL); }
       | PRINT expr ';'             { $$ = createNode(T_PRINT, $2, NULL); }
       | WHILE '(' cond ')' stat    { $$ = createNode(T_WHILE, $3, $5); }
       ;
expr   : prim              { $$ = $1; }
       | expr '+' prim     { $$ = createNode(T_ADD, $1, $3); }
       | expr '-' prim     { $$ = createNode(T_SUB, $1, $3); }
       ;
prim   : NUM               { $$ = createNode(T_NUM, atoi(yytext), NULL); }
       | var               { $$ = createNode(T_VAR, $1, NULL); }
       | '(' expr ')'      { $$ = $2; }
       | var PLUSPLUS      { $$ = createNode(T_PP, $1, NULL); }
       ;
var    : IDENT             { $$ = lookup(yytext); }
       ;
cond   : expr '<' expr     { $$ = createNode(T_LT, $1, $3); }
       | expr '>' expr     { $$ = createNode(T_GT, $1, $3); }
       ;
```

Lexの定義もどこが変わったか理解した上で実行しましょう.

In [ ]:
```
/* lex ex2-8.lex */
alpha   [a-zA-Z]
digit   [0-9]
white   [\n\t ]
%%
read                        { return READ; }
print                       { return PRINT; }
while                       { return WHILE; }
"++"                        { return PLUSPLUS; }
{alpha}({alpha}|{digit})*   { return IDENT; }
{digit}+                    { return NUM; }
[+\-=();{}<>]               { return yytext[0]; }
{white}                     { ; }
```

Cのプログラムにも変更点があります.

In [ ]:
```
/* c ex2-8.c */
#include <stdio.h>
struct stab {
  int val;
  char name[20];
} stab[100];
int stabuse = 0;

extern char *yytext;

#define T_STLIST 1
#define T_ASSIGN 2
#define T_READ   3
#define T_PRINT  4
#define T_ADD    5
#define T_SUB    6
#define T_NUM    7
#define T_VAR    8
#define T_WHILE  9
#define T_LT     10
#define T_GT     11
#define T_PP     12

typedef struct Node {
    int node_type;
    struct Node* left;
    struct Node* right;
} Node;


// 引数にとる文字列が表にあれば見つかった位置を返して，なければ追加する関数
```

```
int lookup(char *s) {
  int i;
  // 表の使っているところを順に見ていって
  for (i = 0; i < stabuse; ++i) {
    if (strcmp(stab[i].name, s) == 0) {
      // 見つかった場合はその位置を返す
      return i;
    }
  }
  if (stabuse >= 99) {
    // 変数の上限の個数を超えてしまった
    printf("table overflow,\n");
    exit(1);
  }

  // 表になかったので追加する
  strcpy(stab[stabuse].name, s);
  return stabuse++;  // 追加した位置を返す
}

// 新しいノードを作る関数
Node* createNode(int node_type, Node* left, Node* right) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->node_type = node_type;
    newNode->left = left;
    newNode->right = right;
    return newNode;
}

// ノードをたどる関数
int dotree(Node* node) {
  if (node == NULL) {
    return 0;
  }
  switch (node->node_type) {
    case T_STLIST:
      if (node->left != NULL) { dotree(node->left); }
      dotree(node->right);
      break;
    case T_ASSIGN:
      stab[(int)(node->left)].val = dotree(node->right);
      break;
    case T_READ:
      scanf("%d", &stab[(int)(node->left)].val);
      break;
    case T_PRINT:
      printf("%d\n", dotree(node->left));
      break;
    case T_ADD:
      return dotree(node->left) + dotree(node->right);
    case T_SUB:
      return dotree(node->left) - dotree(node->right);
    case T_NUM:
      return node->left;
    case T_VAR:
      return stab[(int)(node->left)].val;
    case T_WHILE:
      while(dotree(node->left)) {
        dotree(node->right);
      }
      break;
    case T_LT:
      return dotree(node->left) < dotree(node->right);
    case T_GT:
      return dotree(node->left) > dotree(node->right);
    case T_PP:
      int v = stab[(int)(dotree(node->left))].val;
      stab[(int)(node->left)].val = v + 1;
      return v;
  }
}

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

まず，無限ループになるのがいやなので，`++`の動作を確認しましょう．

In [ ]:
```
/* a.out */
main {
```

```
  x = 2;
  print x++;
  print x;
}
```

それでは， `while` 文を実行してみましょう．

In [ ]:
```
/* a.out */
main {
  x = 0;
  while (x < 5)
    print x++;
}
```

複合文を表わす構文規則が今の文法にはないため， `while` 文のループ本体に1文しか書けないので，これぐらいのプログラムしか書けません． 複合文を追加するのは，今日の練習問題としましょう．

まとめ

- 電卓
- 単純なプログラミング言語のインタプリタ（解釈実行器）

を作り，LexとYaccで自分で考えた構文を持つプログラムを構文解析し，どう実行できるかを見てきました． これで，LALR構文解析など，レポート課題に残した部分はありますが，LexとYaccの基本的な機能はほぼ説明したこととなります．

次回は，これを発展させて，

- 簡単な英文処理プログラム
- 単純なプログラミング言語のコンパイラ

を作成してみましょう．

練習問題

成績の評価時に考慮します．

必須ではないですが，後からの提出は認めないので，授業時間内+α（13:00〜19:00）の間に提出してください．

順番にやる必要はなく，できるものから取り組んでください．

提出方法

以下の問題について，この2.ipynbファイルを編集し，完成した自分が編集した .ipynb ファイルをGoogle Classroomの該当課題に提出してください．（もしこのノートブックをファイルに保存したいときには，メニューの「file」→「Download」で .ipynb ファイルをダウンロード出来るので，それを提出してください．）

- 適宜コメントを入れてください．
- 提出する際のファイル名は，「 **`学籍番号.ipynb`** 」としてください．

問題1

電卓（1）を拡張して，四則演算など他の計算もできるように拡張してください． 前回の出席課題で追加した符号付き整数・実数・指数表現などにも対応している時には加点します．

In [ ]:
```
/* lex p2-1.lex */
ここに書く
```

In [ ]:
```
/* yacc p2-1.yacc */
ここに書く
```

In [ ]:
```
/* c p2-1.c */
ここに書く
```

In [ ]:
```
/* a.out */
自分の書いた電卓のプログラムが正しく動作をしていることを確認できる入力を考えて書く
```

問題2

電卓（2）の属性値を使った最後の結果で，`1+2*3+1`の計算結果が正しくありませんでしたYaccの構文を変更して，演算子の順位をちゃんと考慮して計算できる電卓にしてください．lexとcはそのままで良いはず．

- ヒント：今の定義では，式（`expr`)と因子（`prim`）しかないので，その間に項（`term`，つまり因子を1個以上乗除算でつないだもの）という構文規則を導入してみましょう．

In [ ]:
```
/* lex ex2-3.lex */
digit        [0-9]
white        [\t ]
%%
{digit}+     { return NUM; }
[+*()]       { return yytext[0]; }
"\n"         { return '\n'; }
{white}      { ; }
```

In [ ]:
```
/* yacc p2-2.yacc */
ここに書く
```

In [ ]:
```
/* c ex2-3.c */
#include <stdio.h>
extern char *yytext;

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

In [ ]:
```
/* a.out */
演算子の順序が正しいことを確認できるような入力例を考えて書く
```

問題3

電卓（2）の変数に値を設定するのは代入文でした．C言語のように式の途中に`=`をかける代入演算子にするには，どのように変更すれば良いでしょうか．

In [ ]:
```
/* lex ex2-3.lex */
digit        [0-9]
white        [\t ]
%%
{digit}+     { return NUM; }
[+*()]       { return yytext[0]; }
"\n"         { return '\n'; }
{white}      { ; }
```

In [ ]:
```
/* yacc p2-3.yacc */
%token NUM;
%token IDENT;
%%
stlist :
       | stlist stat '\n'
       ;
stat   : var '=' expr      { stab[$1].val = $3; printf("%d\n", $3); }
       ;
expr   : prim              { $$ = $1; }
       | expr '+' prim     { $$ = $1 + $3; }
       | expr '*' prim     { $$ = $1 + $3; }
       ;
prim   : NUM               { $$ = atoi(yytext); }
       | var               { $$ = stab[$1].val; }
       | '(' expr ')'      { $$ = $2; }
       ;
var    : IDENT             { $$ = lookup(yytext); }
       ;
```

In [ ]:
```
/* c ex2-3.c */
#include <stdio.h>
extern char *yytext;

#include "y.tab.c"
#include "lex.yy.c"

int main() {
  yyparse();
  return 0;
}
```

問題4

`while`文のループ本体で複数の文が書けるように，複合文の構文規則を追加してください．

- ヒント: C言語のように `{}` で囲った文を用意すればよい．
- ヒント: 複数の文は既に `stlist` と定義されているので，それを `{}` で囲った構文規則があればよい．

In [ ]:
```
/* lex p2-4.lex */
ここに書く
```

In [ ]:
```
/* yacc p2-4.yacc */
ここに書く
```

In [ ]:
```
/* c p2-4.c */
ここに書く
```

In [ ]:
```
/* a.out */
while文の本体で複数の文が動いている例を考えてここに書く
```

問題5

`cond`で使えるように，`<=`，`>=`，`==`を追加してください．

- ヒント: Lexに演算子のためのパターンを追加する必要があります（`++`が参考になるかも）
- ヒント: Yaccに規則を追加する必要があります（`cond`の構文規則に追加）
- ヒント: Cの`dotree`の`switch`に処理を追加する必要があります．

In [ ]:
```
/* lex p2-5.lex */
ここに書く
```

In [ ]:
```
/* yacc p2-5.yacc */
ここに書く
```

In [ ]:
```
/* c p2-5.c */
ここに書く
```

In [ ]:
```
/* a.out */
ちゃんと動いている例を考えてここに書く
```

問題6

`cond`で使えるように，`!`（否定），`&&`（論理積），`||`（論理和）を追加してください．

In [ ]:
```
/* lex p2-6.lex */
ここに書く
```

In [ ]:
```
/* yacc p2-6.yacc */
ここに書く
```

In [ ]:
```
/* c p2-6.c */
ここに書く
```

In [ ]:
```
/* a.out */
ちゃんと動いている例を考えてここに書く
```

問題7

今回出てきた，Lex，Yacc，Cを自由に拡張して，作ったプログラミング言語に機能を追加してください．

以下は，レポート課題にする予定なので，ここでやらないでもよい（今やっておいて，レポートに使い回すのはOK）

- `if`文（`else`を含む）
- `switch`文
- サブルーチン
- 関数
- 配列

In [ ]:
```
/* lex p2-7.lex */
ここに書く
```

In [ ]:
```
/* yacc p2-7.yacc */
ここに書く
```

In [ ]:
```
/* c p2-7.c */
ここに書く
```

In [ ]:
```
/* a.out */
ちゃんと動いている例を考えてここに書く
```