J5実験 第1回

テキストを読み込むプログラム

入学してからこれまでの演習・実験で，ユーザーからの入力を受けとるプログラムを作ってきたと思います． J5実験では，**Lex** と **Yacc** というツールを利用して，「構文」に従った入力を受けとるプログラムを作成する方法について学びます．

構文とは

言語処理系論では，文脈自由文法で記述された以下のような構文の例が出てきたと思います．

```
E ::= E_1 + E_2
    | f ( E_1, ..., E_k )
    | N
S ::= E_1 = E_2
    | S_1 S_2
    | while E_1 : S_1
```

この構文の例は，

`E` とは， `E` が `+` でつながれているか， `f` の後ろに括弧 `()` で囲われて `,` で区切られている `E` が並んでいるか， `N` である．

と `E::=` から始まる箇所は読めます．これがプログラミング言語の構文だと考えると，たとえば `f(1, 3)` のようなよくあるプログラミング言語の関数呼び出しを対象とする構文だと分かります．

または，以下のような四則演算の数式を表現する構文も考えられます．

```
因子 ::= 数値 | 変数 | '(' 式 ')'
項   ::= 因子 | 項 '*' 因子 | 項 '/' 因子
式   ::= 項 | 式 '+' 項 | 式 '-' 項
```

これらのような構文を用意し，数値や変数がどういったパターンの文字の並びを言うかさえ定義すれば，

- ユーザーから入力されたテキストが構文に合っているか確かめられて（たとえば，入力 `6+3*5` は上の構文にマッチしていると分かって， `4-(2` はマッチしていないと分かる．），
- 合っていれば抽象構文木が作れる（上の `6+3*5` が `式+項` → `項+項*因子` → `因子+因子*因子` → `数値+数値*数値` と解析できる（ちょっといいかげんな書き方だけれど））

と，自分でパーサ（パーザ）[注1] を1からコツコツと書かなくて済むとうれしいですよね．

Lex と Yacc はそのようなツールで，字句のパターンと構文規則を与えれば，テキストを読み込むプログラムを自動生成してくれます． つまり，J5実験では，**テキストを読み込むプログラムを作るプログラム**を作るわけです．

注1) C言語でパーサを書くとしたら，簡単な構文だったらscanfで「スペースで区切られた数字」などというように読み込めるかもしれませんが，「(と)で囲われていて，if文のような構文があって，...」というような複雑な構文を自分でgetchar() で1文字読みながら構文に合っているか確認するプログラムを書くのはしんどそうというのはなんとなく分かるのではないでしょうか．

テキストを読み込むプログラムを作るプログラム

それぞれ，Lexは **字句解析**，Yaccは**構文解析**を行うプログラムを生成してくれるツールです．

Lexは**正規表現**で字句のパターンを記述して，Yaccは **BNF記法**（**Backus–Naurform**） により構文規則を記述します． つまりJ5実験では，

1. Lexの書き方
2. Yaccでの構文の書き方
3. C言語のプログラムからそれを利用する方法
4. （コード生成）

を学ぶということになります． LexやYaccの新たな記法を学ばなくてはいけないということで，しんどいと思う人もいるかもしれませんが，新たにプログラミング言語を覚えるのに比べたら，記法は直感的であり，動的な動きを考える必要は少なく，はるかに楽です． がんばってやっていきましょう．

Lex

まずは，簡単な例を使って，Lexでのパターンの書き方を見てみましょう．

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

`\n\t` では `\t` の後ろに空白スペースがあることに注意． `NUM` はNumber（数），`IDENT` はIdentifier（識別子）というつもりで付けている名前です．

Lexの記法

Lexのプログラムは，`%%` で上下に区切られており，上が**定義**，下が**パターン**を表しています． つまり，

```
定義1
定義2
...
%%
パターン1　　アクション1
パターン2　　アクション2
...
```

という形をしています．順番に定義の記法，パターンの記法を見ていきましょう．

定義の記法

定義は，パターンで置き換えて使うことができるようにパターンを書いておくものです（変数のようなもの）．つまり，上の例だと，

- `digit` は `[0-9]` に置き換えられる
- `alpha` は `[a-zA-Z]` に置き換えられる
- `white` は `[\n\t ]` に置き換えられる

と読むことができます．つまり，定義を書くことは必須ではなくて，上のLexプログラムは，

```
lex
%%
[0-9]+                  { return NUM; }
[a-zA-Z][a-zA-Z0-9]*    { return IDENT; }
[\n\t ]                 { ; }
```

とも書けるわけです．（良く見た人は，`({alpha}|{digit})*` のところが単純な置き換えではないことに気付いたかもしれません．後述のパターンの記法（正規表現）で分かると思います．）

パターンの記法

パターンは，正規表現を使って記述します．正規表現は，以下の記法を持っています．

パターン	意味	例
`[` 文字… 文字 `]`	`[]` 内の文字のどれか（`[a-z]` のように書いて，`a` から `z` までの全部を表わせる．）	`[a0]` `a` がマッチ
`[^` 文字… 文字 `]`	`[]` 内の文字以外の文字のどれか	`[^x]` `z` がマッチ
`.`	任意の1文字	
文字	その文字．ただし，`[` や `.` などパターンで使う記号は下のエスケープが必要	
`\` 文字	その文字．`[` などのエスケープに使う	`\[`
`"` 文字 `"`	上の `\` 文字 と同じ	`"["`
`{` 定義名 `}`	定義の参照	`{alpha}`

パターン	意味	例
α β	αに続いてβ	`[abc][xyz]` `az` にマッチ
α `|` β	αまたはβ	`([0-2]|[a-c])` `0` にマッチ
`^` α	行の先頭にあるα	`^[+-][0-9]` `+2` にマッチ
α `$`	行の末尾にあるα	`[a-z]Z$` `cZ` にマッチ
α `*`	αの0回以上の繰り返し	`a[a-z]*` `abcdef` にマッチ，`a` にもマッチ
α `+`	αの1回以上の繰り返し	`a[a-z]+` `abcdef` にマッチ，`a` にはマッチしない
α `?`	0個または1個のα	`[+-]?[0-9]+` `+123` にマッチ，`123` にもマッチ
α `/` β	α，ただし直後にβがある	`abc/d` `abcd` にマッチ（ただしマッチの対象は `abc` ），`abc` にはマッチしない
`(` α `)`	αを `()` でくくれる	

αとβには任意のパターンを意味しています．

なので，上の定義で出てきた例はそれぞれ，

- `[0-9]` は，0から9までの字のどれか1文字（つまり数字1文字）
- `[a-zA-Z]` は，aからz，AからZの字どれか1文字（つまりアルファベット1文字）
- `[\n\t ]` は，`\n` か `\t` か ` ` （つまり改行かタブか空白1文字）

と読むことができます．また，定義を展開して，

- `{alpha}+` は，アルファベットが1個以上並んでいるもの
- `{alpha}({alpha}|{digit})*` はアルファベット1文字の後に，アルファベットか数字が0文字以上続く

という正規表現が書かれていたということが分かります． また，上に挙げた例で，`{alpha}({alpha}|{digit})* { return IDENT; }` の定義を展開したときに，`[a-zA-Z][a-zA-Z0-9]*` というパターンにしていたことも何故だったか分かりましたね．

正規表現は，Lexに限定しない一般的なパターンの書き方ですので，ウェブ上で検索するとより詳しい情報が調べられます．

各パターンは正規表現で記述するというのが分かりました．それでは，パターン間で複数のパターンに入力がマッチしてしまう場合はどうなるでしょうか．Lexでは，

1. **マッチする長さが長い方が優先する**
2. **複数のパターンにマッチする場合には，上に書いてある方が優先する**

といるルールがあります（1が優先）．ですので，たとえば，

```
%%
begin                   { return KBEGIN; }
```

```
    [a-zA-Z][a-zA-Z0-9]*    { return IDENT; }
```

というLexの記述があったときに，`begin`という入力に対して`IDENT`としてマッチするわけではなくて，2. のルールから`KBEGIN`としてマッチします．また，`begining`という入力に対しては`begin`と`ing`に分かれてマッチするのではなく，`IDENT`としてマッチします（上のルール1つ目が優先）．

アクションの記法

アクションのところには，パターンにマッチしたときにどういうコードを動かすか，**C言語で書きます**．

Lexの実行

下のコードセルを実行してみましょう．授業スライドにあるように，セルを選択した状態で三角形を押してください．

授業スライドで「重要」と説明したように，1行目はこのJ5実験のために準備したノートブックのコマンドなので，Lexの本来の構文には含まれないことに注意してください．コンテナ内に`ex1-1.lex`が作成されます．

`[Lex] flex generates lex.yy.c successfully`となっていれば，実行は成功（Lexでのパターンの書き方は間違っていなかった）ということになり，`lex.yy.c`が作成されます．
`lex.yy.c`は**上書きされるので注意**．

In [ ]:
```
/* lex ex1-1.lex */
digit [0-9]
alpha [a-zA-Z]
white [\n\t ]
%%
{digit}+                  { return NUM; }
{alpha}({alpha}|{digit})* { return IDENT; }
{white}                   { ; }
```

間違っていると，どうなるかも見てみましょう．6行目で`}`を入れ忘れています．

In [ ]:
```
/* lex ex1-2.lex */
digit [0-9]
alpha [a-zA-Z]
white [\n\t ]
%%
{digit+                  { return NUM; }
{alpha}({alpha}|{digit})* { return IDENT; }
{white}                   { ; }
```

これは，ノートブックに書かれたコードを`ex1-2.lex`に保存した上で，flexコマンドを実行したときのエラーメッセージがそのまま出力されています．8行目で`}`が合わないということが判明するので，8行目で`parse error`とメッセージが出ますが，間違えは6行目にあるので注意してください．（ノートブックのコマンド行（1行目）は除いて行を数える）

字句解析の結果を利用する

yylex()

Lexは字句解析をするコード片を含んだ `lex.yy.c` を自動生成してくれます．その内容は例えば，以下のようになっています．

▶ lex.yy.c の中身

普段は，この `lex.yy.c` の中身を見る必要はありませんし，細かい動作を知る必要もありません．このソースコードで定義されている `yylex()` というC言語の関数が，アクションに書いたとおりの結果を返してくれる（returnしてくれる）と思えばよいです．

yylex()を利用するCコード

それでは，`yylex()` を利用して，字句解析結果を利用するプログラムを見てみましょう．

```
#define NUM 1
#define IDENT 2
#include "lex.yy.c"
int main() {
    int code;
    while((code=yylex())) {
        switch(code) {
            case NUM: printf("num: %s\n", yytext); break;
            case IDENT: printf("id: %s\n", yytext); break;
        }
    }
    return 0;
}
```

`NUM` と `IDENT` は，上で実行した `ex1-1.lex` で動作の箇所で `return` していた値です．`lex.yy.c` を `include` することで `yylex()` を利用でき，字句解析のパターンにあてはまったときの動作に記述した処理が行なわれて結果が返ってくるので，`switch-case` 文で分岐して，字句解析の結果を利用できるわけです．このとき，Lexのパターンに一致した文字列が `yytext` という配列に保存される約束になっています． つまり，このCプログラムは，入力を順番に見ていき（`while` 文），パターンにマッチする箇所で `num:` や `id:` という文字列と一緒にマッチした文字列を出力します．

さっそく実行しましょう．

さきほど，エラーのあるLexを試してしまいましたので，一応，もう一度 `lex ex1-1.lex` を実行して，`lex.yy.c` を作っておきます．

In [ ]:
```
/* lex ex1-1.lex */
digit [0-9]
alpha [a-zA-Z]
white [\n\t ]
%%
{digit}+                    { return NUM; }
{alpha}({alpha}|{digit})* { return IDENT; }
{white}                     { ; }
```

続けて，上で説明したCコードをコンパイルします．（Lexのときと同様に，1行目はこのJ5実験のために準備したノートブックの記法）

In [ ]:
```
/* c ex1-1.c */
#define NUM 1
#define IDENT 2
#include "lex.yy.c"
int main() {
    int code;
    while((code=yylex())) {
        switch(code) {
            case NUM: printf("num: %s\n", yytext); break;
            case IDENT: printf("id: %s\n", yytext); break;
        }
    }
    return 0;
}
```

`[C] gcc generates a.out successfully` と出力されれば，Cコンパイルができて，`a.out` ファイルがコンテナ内に生成されています．エラーの場合にはgccコマンドのエラーがそのまま表示されます．

それでは，a.outを実行してみましょう．このノートブックでは，`/* a.out */` と書いたセルに，a.outに与える標準入力を書いてください．

In [ ]:
```
/* a.out */
123
abc
abc123
123abc
```

実行結果から，どの入力がどの出力に相当しているか分かるでしょうか．`IDENT` を返すパターンでは，アルファベットから始まらないといけないという風になっていたことに注目しないといけないですね．なので，`abc123` は `id:abc123` という結果になるのに対し，`123abc` は `num:123` と `id:abc` となるわけです．自分でも以下のセルに入力を与えて実行してみましょう．

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

演算子とキーワード

以上で，Lexの基本的な記法と使い方が分かったと思います．少し細かいことですが，マッチした文字を `NUM` や `IDENT` のようにC言語で対応を取るとき，すべてを定義するのは少し面倒ですよね．今後，この実験で出てくるようなプログラミング言語を作ろうと思うと，たとえば，`+` や `*` のような記号を**演算子**として使いたくなると思います．そういうときには，ASCII文字の演算子であれば，文字コードをそのまま使えばよいということになります．ASCIIの文字コードは0〜127の数値で，このWikipediaのサイト のように定義されています．つまり，`+` は10進で43（0x2b）ですし，`*` は42（0x2a）をそのまま使えばよいので，

```
    "+"    { return '+'; }
```

というようなパターンとアクションを書けばよいわけです（C言語側でも `'+'` という1文字で対応を取れる）．当然ですが，`NUM` や `IDENT` を先の例ではそれぞれ `1` と `2` で `define` してあるときに他のASCIIコードと被るように数値を割り当てては区別ができなくなってしまうのでいけません．

2文字の演算子を考えると，

```
    "=="    { return EQUAL; }
```

が正しくて，

```
    "=="    { return "=="; }
```

のように間違えがちなので注意してください（上の `+` の例でも1文字なので `'` で囲われてC言語の1文字になっています）．

ここまで分かれば，今後もしプログラムのキーワードが出てきたときには，

```
    if      { return KIF; }
    while   { return KWHILE; }
```

のように定義できることが分かるかと思います．

Lex でのコメント

Lexのプログラムにもコメントを入れることができます．今バックエンドで使っているflexでは，C言語のコメントと同様の記法（ `/*` と `*/` で囲う）が使えます． `%%` より上の「定義」の箇所はそのまま置き換えられるので注意．

適宜コメントを入れましょう．

In [ ]:
```
/* lex ex1-3.lex */
digit [0-9]
alpha [a-zA-Z]
white [\n\t ]
%%
{digit}+                  { return NUM; }          /* 数字が1つ以上並んでいる */
{alpha}({alpha}|{digit})* { return IDENT; }        /* 識別子はアルファベットから始
{white}                   { ; }                    /* 空白のときは何もしない */
```

Lexの限界

Lexは，正規表現を用いてパターンを記述するので，**入れ子構造になったときに，括弧の対応を取る**といったことを表現するには表現力が足りません．つまり，`(a, b, (c, d))` のような構造のときに

は，`(` がどれだけ深くなったかを数えられないため，正しい括弧閉じで対応が取られているか表現できないということです．(詳しくは正規表現についてウェブなどで調べてみましょう．)

そのため，Lexだけを使って，このJ5実験の目的であるプログラミング言語のような構文規則に従った入力を受けとるプログラムを作成できないということになります． 次章で Yacc を使って，その問題を解決しましょう．

Yacc と BNF

BNFとは

BNFとは，以下のような形式で書かれた構文の規則を並べたものです．

```
記号1 ::= 記号の並び
記号2 ::= 記号の並び
...
```

（この授業資料では，各行を**構文規則**と言うことにします．）

ここで，「記号（シンボル）」とは名前のようなもので，

- 終端記号 : `::=` の左辺に表われない．
- 非終端記号 : `::=` の左辺に表われる．

の2種類があります．おおまかに言うと，非終端記号は他の規則を使って右辺に置き換えて考えることができるということです． 構文の例を見てみましょう．

```
expr ::= NUM
expr ::= expr '+' NUM
```

ここでは，2つの構文規則があります． `NUM` や `'+'` が終端記号， `expr` が非終端記号ということが分かると思います． この構文をどう読むかというと，

「`NUM` が1個のものは `expr` であり（1つ目の構文規則），また `expr` の後に `'+'` と `NUM` が続くものも `expr` である（2つ目の構文規則）．」

ということになります． `expr` が左辺に2回表われるので，省略して，

```
expr ::= NUM
       |  expr '+' NUM
```

もしくは，1行にまとめて，

```
expr ::= NUM | expr '+' NUM
```

と書けます．

このような構文は，「+でつながれたNUMの並び」と読むことができます． **左辺を適当な右辺で置き換えていく**ことで，実際に試してみることができます． たとえば， `1+2+3` という文字列であれば（NUMは整定数だとする），

```
expr
→ expr + NUM
→ expr + NUM + NUM
→ NUM + NUM + NUM
```

と順番に `expr` に構文規則を適用していくことができますね． つまり，Lexのときのように，ある入力文字列が，この構文にマッチしているかを確認できることになります．

BNF と Yacc

Yaccは，前節の BNF を使って，構文解析してくれるツールです．さっそく Yacc で上の BNF を記述してみましょう．

```
%token NUM;
%%
expr : NUM
     | expr '+' NUM
     ;
```

Yaccの記述もLexと似ていて， `%%` より前に定義，後にパターン（構文規則）を書きます． この例の定義部では，「 `NUM` が終端記号です．」と定義しています． `'+'` のように `'` で囲まれた1文字の記号以外の終端記号は，定義しておく約束になっています．

構文規則は， `::=` の代りに `:` を使っていること，構文規則の最後に `;` を書くこと以外は，前節の BNF と違いはありません． さっそく，以下の Yacc のコードを実行してみましょう．

In [ ]:
```
/* yacc ex1-1.yacc */
%token NUM;
%%
expr : NUM
     | expr '+' NUM
     ;
```

`[Yacc] bison generates y.tab.c successfully` と出力されていれば，正しく実行できています．Lexのときは `lex.yy.c` でしたが，Yaccでは `y.tab.c` が生成されています． なので，Lexのときと同様にC言語のmainを作ればよいことになるのですが， **Yaccは `yylex()` から入力を読みとるようにできています． つまり，Lexと合わせて使えばよいということです．**

対応するLexのプログラムは以下のようになるので，実行して `lex.yy.c` を生成しましょう．

In [ ]:
```
/* lex ex1-4.lex */
digit     [0-9]
white     [\n\t ]
%%
{digit}+  { return NUM; }
"+"       { return '+'; }
{white}   { ; }
```

両者を呼び出すCのプログラムは以下のようになります．

In [ ]:
```
/* c ex1-2.c */
#include <stdio.h>
extern char *yytext;

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

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

このmainは，何もせずに構文にマッチしたかだけを見ています．つまり，構文に適した正しい入力をしたときは**何もしない**で，間違っていると**エラーのメッセージ**が出ます．それを確認するために，生成されたa.outを実行してみよう．

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

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

Yaccのアクション

第1回の最後に，Yaccにもアクションが書けることを見て終わりにしましょう． 上の例では，正しいとき（構文規則にマッチしたとき）には何もしないで，間違っているときにはエラーメッセージが出ました．当然，構文規則にマッチしたときに処理がしたくなるわけです． そのためには，Lexのときのように，構文規則それぞれにアクションを書いておきます．

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

Lexでは，パターンにマッチした文字列が `yytext` に入っているので（不安な人は 前のほうの節を利用するCコード) に戻る），それを表示するアクションを追加してみました．LexとCはそのままで，実行しなおしてみましょう．

In [ ]:
```
/* lex ex1-4.lex */
digit     [0-9]
white     [\n\t ]
%%
{digit}+  { return NUM; }
"+"       { return '+'; }
{white}   { ; }
```

In [ ]:
```
/* yacc ex1-2.yacc */
%token NUM;
%%
expr : NUM              { printf("NUM: %s\n", yytext); }
     | expr '+' NUM     { printf("ADD: %s\n", yytext); }
     ;
```

In [ ]:
```
/* c ex1-2.c */
#include <stdio.h>
extern char *yytext;

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

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

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

上で説明したように構文規則にマッチしていることが分かります．

Yacc でのコメント

Yacc のプログラムにもコメントをC言語のコメントと同様の記法（ `/*` と `*/` で囲う）で入れることができます．．

適宜コメントを入れましょう．

Lex と Yacc のデバッグ

`a.out` を実行したときに，どう構文規則にマッチしたかを知りたいときがあります．

まず，Lexの場合には， `-d` オプションを付けてください．（本来の `flex` コマンドの時には `% lex -d ファイル名` となりますが，このノートブックのコマンドでは最後に `-d` を付けます．）

In [ ]:
```
/* lex ex1-4.lex -d */
digit [0-9]
alpha [a-zA-Z]
white [\n\t ]
%%
{digit}+                  { return NUM; }
```

```
{alpha}({alpha}|{digit})* { return IDENT; }
{white}                   { ; }
```

In [ ]:
```
/* c ex1-1.c */
#define NUM 1
#define IDENT 2
#include "lex.yy.c"
int main() {
    int code;
    while((code=yylex())) {
        switch(code) {
            case NUM: printf("num: %s\n", yytext); break;
            case IDENT: printf("id: %s\n", yytext); break;
        }
    }
    return 0;
}
```

In [ ]:
```
/* a.out */
123
abc
abc123
123abc
```

テキストがどうマッチしたかが詳細に出力されます．

次に，Yaccの場合には，Yaccのプログラムをコンパイルするときに `--debug` オプションを付けます． この授業のノートブックでは，1行目のコマンド行に `yacc ファイル名 オプション` と書いておくことで，Yacc（Bison）のオプションを指定できるようにしてありますので，以下のように書きます．

In [ ]:
```
/* lex ex1-4.lex */
digit     [0-9]
white     [\n\t ]
%%
{digit}+  { return NUM; }
"+"       { return '+'; }
{white}   { ; }
```

In [ ]:
```
/* yacc ex1-2.yacc --debug */
%token NUM;
%%
expr : NUM              { printf("NUM: %s\n", yytext); }
     | expr '+' NUM     { printf("ADD: %s\n", yytext); }
     ;
```

さらに，Cのプログラムで， `yydebug` 変数の値を `0` 以外にします．

In [ ]:
```
/* c ex1-3.c */
#include <stdio.h>
extern char *yytext;

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

int main() {
  yydebug = 1;              /* この行がデバッグ表示のために必要 */
  yyparse();
```

```
  return 0;
}
```

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

少し長いですが，テキストを読んでいってどのように解析が進んだかが出てきます．「`Shifting token`」や「`Reducing stack by`」というところに注目してください．（行番号は，コマンド行（`/* a.out */`）を抜かして数えてください）． たとえば，

```
   Reducing stack by rule 1 (line 3):
```

がありますが，これは `expr : NUM` の規則を適用したことを意味しています．

詳しくは，Yacc が生成する構文解析器がどのようなアルゴリズムかが分かっていないとちゃんと読めないのですが，

- ***Shift（シフト，遷移）***：入力を進める．
- ***Reduce（還元）***：構文規則を適用する．

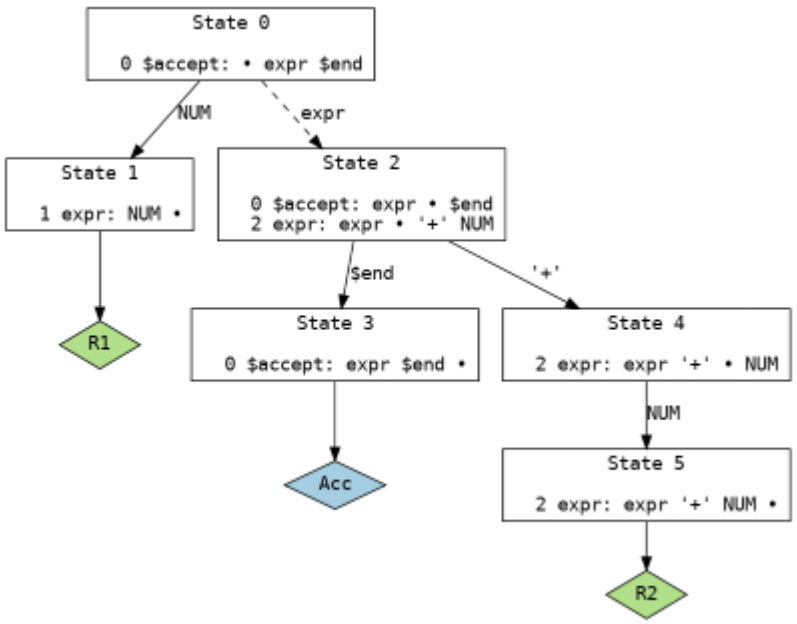
という2つの用語は，重要ですので覚えてください． Yaccが生成する構文解析器については，次回に解説します．

エラーがある場合も見ておきましょう．

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

2つ目の `+` が来たところでエラーというのが分かります．

さらに，Yacc に `--verbose` または `--graph` を付けると，それぞれ `y.output`（テキストファイル）または `y.gv`（Graphvizのdot言語）を出力してくれるので，どんな状態を遷移するかも分かります． 興味がある人は試してみてください．



In [ ]:
```
/* yacc ex1-2.yacc --debug --verbose */
%token NUM;
%%
expr : NUM              { printf("NUM: %s\n", yytext); }
     | expr '+' NUM     { printf("ADD: %s\n", yytext); }
     ;
```

まとめ

今回は，LexとYaccを使い始めてみるということで，字句のパターンにマッチするか，構文規則にマッチするかということを確認する非常に基本的なプログラムだけでしたが，これを発展させれば，単なる文字の並びであるテキストを読み込んで意味を持たせることができます．

次回は，

- 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）

を実際に作ってみましょう．

練習問題

提出は任意です．

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1.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

`ex1-1.lex` として記述した数と識別子を解析するLexのプログラムを元にして，「符号付き整数」と「実数」が扱えるようにパターンを記述してください． たとえば， `-5` や `1.23` などが読み込めるようにしたいということです． `-02` や `+002.3` などは大目に見て気にしないで良いです（エラーにしないでよい，できる人はもちろんチャレンジしても良い）． 符号付き整数は `NUM` のパターンを変更する形で，実数は新たに `RNUM` をリターンするようなアクションのパターン行を追加する形で実現するとよいと思います．

- ヒント1: `sign [+-]` というのを定義に追加して，数の前に `sign` が付くかもしれないようにすればいいです．（かもしれない，が重要．じゃないと元の `123` などが字句解析エラーになってしまう）
- ヒント2: `dot "."` というのを定義に追加して，`dot` を含んだら `RNUM` というパターンを書けばよいです．

この問題では，Lexだけ書けばよいです（なので，実行してちゃんと字句解析ができるかは確認できない）．

In [ ]:
```
/* lex p1-1.lex */
ここに書く
```

問題2

上の問題1の `p1-1.lex` に対応したCコード `p1-1.c` を書いてください． `ex1-1.c` を参考にして，マッチしたパターンを同様に出力してください． a.outを実行して，実行結果が思ったとおりの動作であるか，何通りかの入力で確かめること．

ヒント1: `case` の分岐に `RNUM` の部分を追加するだけなので，簡単．

In [ ]:
```
/* c p1-2.c */
ここに書く
```

In [ ]:
```
/* a.out */
ちゃんとできているか確認するためのテストの入力を自分で考えて書く
```

問題3

問題1・2に加えて，指数形式（ `12.5e-6` とか `3e15` とか）も解析できるように改良してください．

In [ ]:
```
/* lex p1-3.lex */
ここに書く
```

In [ ]:
```
/* c p1-3.c */
ここに書く
```

In [ ]:
```
/* a.out */
ちゃんとできているか確認するためのテストの入力を自分で考えて書く
```

問題4

YaccとLexを組み合わせて `'+'` でつながれた数を構文解析できることが分かったと思います．同様にして，引き算も扱えるように改良してください． 入力が正しいときは何もしないで，構文に合っていないときにはエラーが出ればよいです（なので，Cのコードはそのままだけれど，再コンパイルはする必要がある）．

In [ ]:
```
/* lex p1-4.lex */
ここに書く
```

In [ ]:
```
/* yacc p1-4.yacc */
ここに書く
```

In [ ]:
```
/* c p1-4.c */
#include <stdio.h>
extern char *yytext;

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

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

In [ ]:
```
/* a.out */
ちゃんとできているか確認するためのテストの入力を自分で考えて書く
```

問題5

Yaccの最後の例では，BNFにすると

```
    expr : NUM | expr '+' NUM
```

という2つの構文規則を使いました．exprの適用の順番が代わるだけで，以下のように書いても同じことができそうと考えると思います．

```
    expr : NUM | NUM '+' expr
```

実際にYaccを書き直したものを実行し，どうなるか，なぜそうなるかを下のMarkdownのセルに書いてください（つまり，文章でどういうことが起きたのかを考えて書くということ）．

ヒント: yytextはLexが直前にマッチした文字列を入れているものである．

In [ ]:
```
/* lex ex1-4.lex */
digit     [0-9]
white     [\n\t ]
%%
{digit}+  { return NUM; }
"+"       { return '+'; }
{white}   { ; }
```

In [ ]:
```
/* yacc p1-5.yacc */
%token NUM;
%%
expr : NUM              { printf("NUM: %s\n", yytext); }
     | NUM '+' expr     { printf("ADD: %s\n", yytext); }
     ;
```

In [ ]:
```
/* c ex1-2.c */
#include <stdio.h>
extern char *yytext;

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

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

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

なぜそうなるかを考えて，ここに書く

問題6

今回出てきた，Lex，Yaccを自由に拡張して，新たな字句・構文を読めるようにしてください．Cのコードは，`ex1-2.c` そのままとします．

In [ ]:
```
/* lex p1-6.lex */
ここに書く
```

In [ ]:
```
/* yacc p1-6.yacc */
ここに書く
```

In [ ]:
```
/* c ex1-2.c */
#include <stdio.h>
extern char *yytext;

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

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

In [ ]:
```
/* a.out */
ちゃんとできているか確認するためのテストの入力を自分で考えて書く
```