S実験'95　S3 (LexとYacc) ＃2

久野　靖 *

1995.4.26, 1995.5.24, 1995.6.5

2　YaccとBNF

2.1　BNFとは

さて、前回やっていただいたlexの弱点というのは分かりましたか? 例えば「C言語のif文」という「パターン」をlexのパターンでこなそうとするとどうなるか。「`if(`」までは問題ない。次に条件式がくるのだけど、式というのは変数も定数も四則演算も比較演算も関数呼び出しもあるので、それを1個のパターンに全部押し込もうとしても大変なだけである。しかも、「かっこが対応している」などというごく当然の規則でさえlexのパターンでは書き表すことができない。

実は、このような用途にはlexのパターンは原理的に記述力が足りない。そこで今回はBNFというまた別の記法をやることにする。BNFというのは要するに

```
記号 ::= 記号のならび
```

という形式をたくさん並べたものである。ここで「記号」というのはひらたく言えば「名前」のようなもので、

・端記号　--- 式の左辺に現れない。
・非端記号 --- 式の左辺に現れる。

の2種類がある。例えば次の例を考えよう。

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

蛇足だが記述を短くするために「または」という意味で「`|`」を使うことができるので、こう書いても同じである。

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

ここでNUMというのは整定数だとしよう。また「`'+'`」というのが「名前」というのもちょっとおかしいが、こないだのlexの時と同様、この方が見やすいのでそうしておく。それで、上の記述はこう読む。「整定数1個というのはexprである。また、exprがあって、+があって、整定数がある、というのもexprである。」さて、exprというのは何を表すと思いますか?

答えは... +で結ばれた数の並び、いいかえれば足し算と整定数だけを含む式、ですよね。これは「BNFの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見られる。例えば、

```
1 + 2 + 3
```

であれば、次のような具合である。

*筑波大学大学院経営システム科学専攻

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

明らかに NUM の数はいくつにでもできる。これまでで何となく分かったと思うが、端記号 (NUM とか+とか) は入力に実際に現れる具体的なもの、非端記号というのは構文上の概念 (式とか文とか) に対応している。

2.2 BNF と Yacc

さて、ちょうどパターンと lex の関係に対応して、BNF からこのような「式」を実際に認識してくれるプログラムを生成するツール、というのが Yacc である。上の例に対応する Yacc の記述を見て頂こう。

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

Yacc の記述も lex と類似していて、%%より前に宣言部、後にパターンを書くようになっている。ここでは宣言は「NUM というのは端記号ですよ」というものだけである (Yacc では端記号になる名前で'' 形式でないものは予め宣言しておくことになっている)。あとは明らか、::=の代わりには:が使われ、|はそのまま、1 つの規則は複数行に渡ってもいいので、規則の終りには; を書く、というだけのことである。これを動かすには、たとえばこれが t3.yacc というファイルに入っていたとして、

```
% yacc t3.yacc
```

といえばよい。何も文句をいわれなければ OK で、yacc は y.tab.c という名前のファイルを作り出す。

さて、これを動かすには main もいるが、その前に+とか NUM とかを入力から取り出してくる関数 yylex もないと困る。これはもちろん、lex を使って作る (自前で yylex を書いてももちろんいいけど)。lex ソースを示そう:

```
digit   [0-9]
white   [\n\t ]
%%
{digit}+                        { return NUM; }
"+"                             { return '+'; }
{white}                         { ; }
```

よろしいですね? あとは C のソースだがこれは次の通り。

```
#define TRUE    1
#define FALSE   0
#define yywrap() 1
extern char yytext[];  ← このおまじないも新しく必要。
#include "y.tab.c"     ← y.tab.c をとりこむ。
#include "lex.yy.c"    ← lex.yy.c をとりこむ。
                       ← ※1
main() {
    yyparse();         ← main はこれだけ。
}                      ← ※2

yyerror(s)             ← これもおまじないとして必要。
    char*s; {             (単にエラーを表示する関数。)
    printf("%s\n",s);
}
```

さて、lex ソースが t3.lex、C ソースが t3.c に入っているものとして、さっそくこれを実行させてみる。

```
% lex t3.lex
% cc t3.c
% a.out
1 + 2 + 3
^D
% a.out
1 + + 2
syntax error
%
```

つまり、正しい式を入力した時は「何もしない」で、まちがっているとエラーメッセージを表示して終る。

練習 1: t3.yacc、t3.lex、t3.c 一式を打ち込んで動かせ。

練習 2: 足し算だけでなく引き算も使えるように直してみよ。[1]

2.3 Yacc のアクション

既におわかりと思うが、「何もしない」のは「動作」を何も指定していないからで、実は yacc でも lex と同様にして規則の後に「その規則が現れたら実行する動作」を指定できる。次のような具合である。

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

前回やったように、lex ではパターンにあてはまった文字列を配列 yytext に入れてくれるので、これを表示する動作を入れたわけである。さっきのをこれに入れ替えて実行してみた例を示す。

```
% yacc t31.yacc
% lex t3.lex
% cc t3.c
% a.out
1 + 2 + 3
NUM:1
ADD:2
ADD:3
^D
%
```

確かに、先に説明した通りに規則が使われていることが分かる。

練習 3: 引き算も追加した版の yacc 記述に上のように動作を追加して動かしてみよ。

練習 4: ところで、これまで使ってきた文法を次のように直すことを考えてみる (どう違うか見比べてみること)。

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

これでも「1 + 2 + 3」が生成できることはちょっと考えればわかる (いいですね?)。yacc の記述も対応して直してみよ。それを動かすと動作はどう変化するか? なんでそうなってしまうのか?

[1] t3.lex も直す ('-' が返せるようにする) のを忘れないこと。

2.4 電卓その 1

さて、せっかく「式」が認識できるのだから、その「値」を計算してみよう。それには yacc の動作をちょっと直して次のようにすればよい。

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

なお、atoi というのは文字列 (たとえば"123") からそれが表している整数 (123) に変換するライブラリ関数である。で、要するに最初の数が現れた時には value という変数にその数を入れ、足し算があるごとに新しい数値を足し込む。これが動くためには、変数 value がないと困るからさっきの t3.c の※ 1 のところに

```
int value;
```

を入れ、また計算しただけで結果が表示できないのでは困るから※ 2 の前のところに

```
printf("%d\n", value);
```

を入れる。これで動かしてみると。

```
% yacc t32.yacc
% lex t3.lex
% cc t32.c
% a.out
1 + 2 + 3
^D
6
%
```

確かに計算ができている。ところで、^D を打って入力を終りにしないと結果が出ないことに注意。というのはこの yacc/lex 記述だと式は何行にまたがって書いてもいいから^D が来ないともう終りかどうかが分からないからである。

練習 5: 引き算も追加した版の yacc 記述で上のように計算を行なうようにしてみよ。

練習 6: さっきやった、変更した文法の方だとどうなるか、やってみよ。やる前に何か不都合があるかどうか考えてみること。

2.5 1 行ごとに

やっぱりこれでは不便だから、1 行に式は 1 つで、改行するとすぐに結果が見えるように直してみよう。そのために文法を次のように直す。

```
%token NUM;
%%
explist :
        | explist expr '\n' { printf("%d\n", value); }
        ;
expr    : NUM           { value = atoi(yytext); }
        | expr '+' NUM  { value = value + atoi(yytext); }
        ;
```

追加された部分はこう読む: 「式の並びとは、(a) 空っぽであるか、または (b) 式の並びの後に式と改行が続いたもの。」これで式を含んだ行が何行でもある、という意味になるのは分かりますね? しかも (b) の場合というのはちょうど式 1 個の計算が終ったところであるはずだから、ここで value の値を打ち出せば式の値が計算できる。なお、こうするためにはまたまた lex の方を'\n' が返せるように直す必要がある。もうできると思うけど念のため示しておこう。

```
digit   [0-9]
white   [\t ]
%%
{digit}+                        { return NUM; }
"+"                             { return '+'; }
"\n"                            { return '\n'; }
{white}                         { ; }
```

あと、さっきの main にあった printf はもういらない。これで動かしてみよう。

```
% yacc t33.yacc
% lex t33.lex
% cc t33.c
% a.out
1 + 2
3
3 + 4 + 5
12
^D
%
```

練習 7: 引き算も追加した版のを上のように直して 1 行ごとに式の値を表示するようにしてみよ。もしよかったらかけ算と割り算もどうぞ。

2.6 記号の属性値

ところで、式の計算をするからには「かっこ」が使えるようにしたい。ところが、今までのやり方だと計算の途中結果は変数 value に入っているだけなので、かっこの中の部分式を計算するのに value を使ってしまうと外の式の途中結果が壊れてしまう。

これを解決するのに value を配列にするとかスタックにするとかいう方法も可能だが、もっと yacc ならではの方法を使ってみよう。実は yacc では各非端記号に「属性値」を付属させることができる。おまじないを駆使すると色々な属性値が使えるのだけど、ここでは簡単に整数値が 1 個だけ、ということにしておく。で、それを利用して「式の中間結果」を対応する記号の属性値として扱うわけである。見ていただこう:

```
%token NUM;
%%
explist :
        | explist 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$$、というのは「左辺の記号の属性値」を表し、$n、というのは「右辺の n 番目の記号の属性値」を表す。で、式の値を expr の属性値に入れておくことにしたから、式の値を表示するところでは (expr は右辺の 2 番目だから)$2 を表示すればよい。また、足し算は+の前の式の値 ($1) と+の後の因子の値 ($3) を足したものをこの式の値 ($$) とすればよい。かけ算も同様。なお「因子」は、これまでの「数値」に加えて「かっこでくくった式」も扱いたいために導入したものである。次にこれを動かした例を示す。(lex の方も'*' と'(' と')' が扱えるように直すのを忘れないように!)

```
% yacc t34.yacc
% lex t34.lex
% cc t3.c
% a.out
(1 + 2) * (3 + 1)
12
1 + 2 * 3 + 1
10
^D
%
```

確かに、かっこが使えるようになっている。

練習 8: 引き算も追加した版のを上のように直してかっこが使えるようにしてみよ。

練習 9:* 上の「$1 + 2 * 3 + 1 = 10$」という結果はどうしてそうなってしまうのか。またそれを直してみよ。[2]

2.7 名前と記号表

さて、この電卓は式の計算そのものはだいぶできるようになったが、一度作った中間結果を覚えているのが面倒である。そこで「変数」を入れることにしよう。そのためには「どんな変数があり、それぞれの値はいくつか」を覚えておかないといけない。そこで例えば図 1 のようなデータ構造を使うことにする。表全体は `stab` という名前で、これは `struct stab` 型が 100 個並んだ

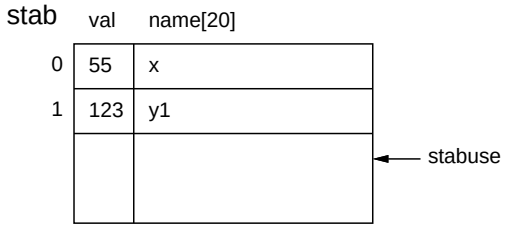

図 1: 簡単な記号表

配列である。なお、`struct stab` 型というのは、整数型の val という欄と、長さ 20 の文字配列の name という欄を持つレコードである。また `stabuse` という変数にこの表をどこまで使っているかを入れておくことにする。

これを扱うために、名前を渡すとそれは表の何番目かを返す関数 `lookup` を用意した。まだ表にない名前がくるとそれは表に追加される。これを含む C ソースファイルを示す。

```
#define TRUE    1
#define FALSE   0
#define yywrap() 1
extern char yytext[];
struct stab {
    int val;
    char name[20]; } stab[100];
int stabuse = 0;
#include "y.tab.c"
#include "lex.yy.c"
```

[2]ヒント: 「式」と「因子」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) というのを導入するとうまくいく。

```
main() {
    yyparse();
}

lookup(s)
    char *s; {
    int i;
    for(i = 0; i < stabuse; ++i)
        if(strcmp(stab[i].name, s) == 0) return i;
    if(stabuse >= 99) {
        printf("table overflow.\n"); exit(1); }
    strcpy(stab[stabuse].name, s); return stabuse++;
}

yyerror(s)
    char *s; {
    printf("%s\n",s);
}
```

なお、strcmp は 2 つの文字列を比較するライブラリ関数、strcpy は文字列のコピーを行なうライブラリ関数である。これを用いて、変数を使うように改造した yacc ソースを示す。

```
%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); }
        ;
```

今度は各行は「式」ではなく代入文である。その時やる動作は、式の値を左辺の変数 (に対応する記号表の場所の val 欄) に入れ、同時にその値を打ち出すことである。また、「因子」として変数が現れた時はその値 (正確にはその変数に対応する記号表の場所の val 欄の値) を因子の値とする。ほとんどの属性値は依然として式の途中結果であるが、var(変数) の属性値だけは「それが記号表の何番目の変数か」を表していることに注意。さて、こんどは lex が IDENT も返す必要があるので lex ソースも一応載せておく。

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

なお、※のところは「改行、+、*、(、)、=のどれかであれば、その文字 (というのは配列 yylex の最初に入るわけですよね) を返す) という意味で、これで 1 文字を返すものをまとめて扱える。では実行例。

```
% yacc t4.yacc
% lex t4.lex
% cc t4.c
% a.out
z = 1 + 1
2
x = z * z
4
^D
%
```

練習 10: これまでに作った電卓のいちばん気にいっている版を上と同様に変数が使えるように改良せよ。

練習 11:* これだと「代入文」なので、C 言語のように式の途中に=を置くことはできない。C 言語のように「代入演算子」方式に直してみよ。

A 本日の練習問題兼出席

「練習 n」と記されているものを順にやってください。ただし、「*」がついているものは時間がなさそうなら飛ばしてください。あとで時間が余ったらやってね。だいたい時間になったら、その時までにできている一番最新の版の yacc ソースと、それの実行例を打ち出してください。そして最後の紙の裏に以下の項目:

- 学籍番号、氏名、所属、本日の日付。
- 以下のアンケートに対する答え (簡単でいいですよ)。

  Q1. BNF について知っていましたか。これまで、Yacc のようなツールを使ったことがありますか。ある人は、それと比べてどうですか。

  Q2. 本日やったことのうち面白かった/興味深かった部分はどこですか。また、難しいと思った部分はどこですか。

  Q3. 本日の感想、今後の要望などお書きください。

を記入したものを出席用に提出してください。出席として認定されるためには、本日 5 時までに事務室のレポートボックスに提出のこと。