# S実験'95　S3 (Lexと Yacc) ＃3

久野　靖 *

1995.5.8, 1995.5.29, 1995.6.7

## 3　Lex + Yacc = 言語処理系

### 3.1　ごく簡単なプログラム言語

前回やったのはあくまでも「電卓」だったが、ちょっと直すとプログラム言語っぽくなる。次のyacc 記述を見て頂きたい。

```
%token NUM;
%token IDENT;
%token READ;
%token PRINT;
%%
prog    : IDENT '{' stlist '}'  { return; }
        ;
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); }
        ;
```

でも結局は構文をちょっとらしくして、read と print を入れただけだが。このため lex 記述も次のようにする。

```
alpha   [a-zA-Z]
digit   [0-9]
white   [\n\t ]
%%
read                            { return READ; }
print                           { return PRINT; }
{alpha}({alpha}|{digit})*       { return IDENT; }
```

*筑波大学大学院経営システム科学専攻

```
{digit}+                        { return NUM; }
[-+()=;{}]                      { return yytext[0]; }
{white}                         { ; }
```

C ソースは前回と全く同じなのだが一応つけておこう。

```
#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"   ※1
#include "lex.yy.c"

main() {
    yyparse();
}

lookup(s)
    char *s; {
    int i;
    for(i = 0; i < stabuse; ++i)
        if(strcmp(stab[i].name, s) == 0) return i;
    strcpy(stab[stabuse].name, s); return stabuse++;
}

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

さて、実行例。

```
% lex t5.lex
% yacc t5.yacc
% cc t5.c
% a.out
main {
  read x;
10
  y = x + 1;
  print y;
11
}
%
```

read 文を打ち込んだとたんにそのデータも打たないといけないところが笑えるが、一応プログラム言語みたいでしょう?

### 3.2 木構造の作成

前の例が笑えてしまうのは、基本的に yacc のアクションでいきなり計算を実行してしまうためである。まっとうなプログラム言語処理系であれば、そうはしないでプログラムの構造に対応した中間形式を作成し、それをもとに実行するのが普通である。

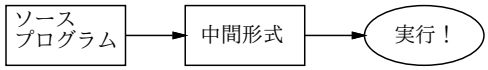


図 1: 中間形式

ここでは、中間形式としてプログラムの構造に対応した木構造を作成することにする。例えば先の実行例のプログラムに対応した木構造は図 2 のようなものになる。T_STLIST などは「この節がどの種類か」を区別するための番号 (もちろん適宜 define する) である。これにプログラムに関するすべての情報が含まれていることに注意。

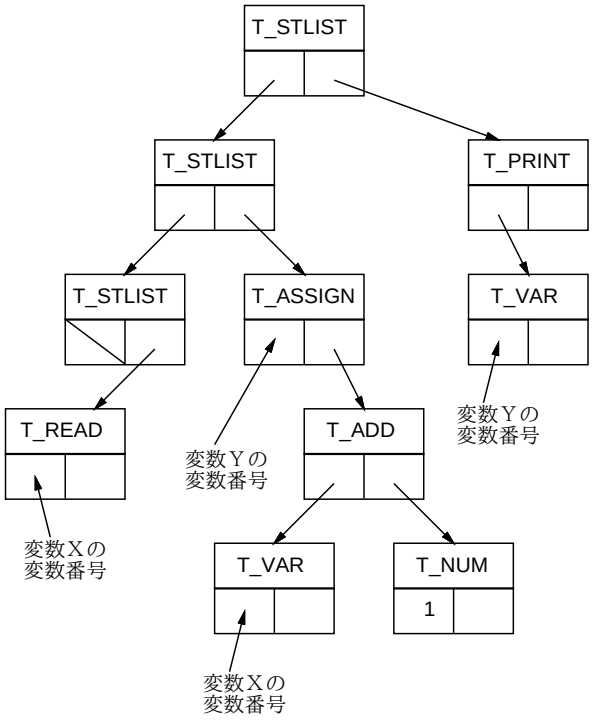


図 2: 木構造の中間形式

このデータ構造を例によって構造体とポインタで作ってもいいのだが、ここではちょっとさぼって ntab という構造体の配列を使用し、ポインタの代わりに配列の何番目、という整数を使うことにした。その宣言部は次の通り。これをさっきの C ソースの※ 1 の上あたりに入れることにする。

```
struct node {
    int type, left, right; } ntab[400];
int ntabuse = 1;
#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
```

この配列上にさっきの図の「代入文」のあたりが入っている様子を図 3 に示す。これを組み立てるのに node という関数を利用する。これは単に配列 ntab の次の使ってない場所に渡された type, left, right を格納してそれが ntab 何番目の場所に入ったかを返す。

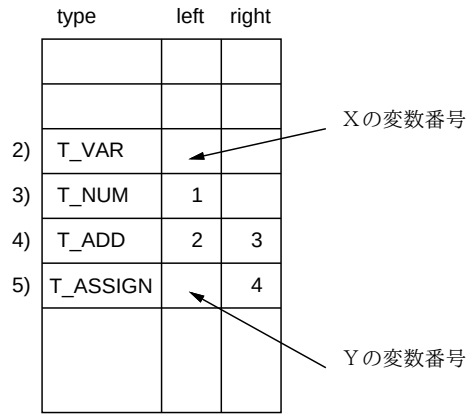


図 3: 配列に入った中間形式

```
node(t, l, r)
    int t, l, r; {
    int i = ntabuse++;
    ntab[i].type = t; ntab[i].left = l; ntab[i].right = r;
    return i;
}
```

これを利用して木構造を組み立てる yacc 記述は次の通り。

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

なお、ちゃんと木構造が組みたたったかどうか調べるために dotree という関数で木をたどってその内容をプリントしている。

```
dotree(i)
    int i; {
    switch(ntab[i].type) {
case T_STLIST: if(ntab[i].left) { dotree(ntab[i].left); printf("; "); }
               dotree(ntab[i].right); break;
```

```
case T_ASSIGN: printf("v%d = ", ntab[i].left);
               dotree(ntab[i].right); printf("\n"); break;
case T_READ:   printf("read v%d\n", ntab[i].left); break;
case T_PRINT:  printf("print "); dotree(ntab[i].left); printf("\n"); break;
case T_ADD:    printf("("); dotree(ntab[i].left); printf(" + ");
               dotree(ntab[i].right); printf(")"); break;
case T_SUB:    printf("("); dotree(ntab[i].left); printf(" - ");
               dotree(ntab[i].right); printf(")"); break;
case T_NUM:    printf("%d", ntab[i].left); break;
case T_VAR:    printf("v%d", ntab[i].left); break; }
}
```

では実行例。(lex はさっきと同じでよい。)

```
% yacc t6.yacc
% cc t6.c
% a.out
main {
  read x;
  y = x + 1;
  print y;
}
read v0
; v1 = (v0 + 1)
; print v1
%
```

別に入力したものとあんまり変わっていないが、変数が変数番号に置き換わっているのでまあちゃんと変換されているなあと思う。

### 3.3 木構造の解釈実行

といっても、やっぱりプリントするだけではつまらない。変わりに、木構造を解釈実行するように直す。これには dotree を次のものにとりかえるだけでよい。

```
dotree(i)
    int i; {
    switch(ntab[i].type) {
case T_STLIST: if(ntab[i].left) dotree(ntab[i].left);
               dotree(ntab[i].right); break;
case T_ASSIGN: stab[ntab[i].left].val = dotree(ntab[i].right); break;
case T_READ:   scanf("%d", &stab[ntab[i].left].val); break;
case T_PRINT:  printf("%d\n", dotree(ntab[i].left)); break;
case T_ADD:    return dotree(ntab[i].left) + dotree(ntab[i].right);
case T_SUB:    return dotree(ntab[i].left) - dotree(ntab[i].right);
case T_NUM:    return ntab[i].left;
case T_VAR:    return stab[ntab[i].left].val; }
}
```

かえってこの方が簡単でしょう? では実行例。

```
% cc t7.c
% a.out
main {
  read x;
  y = x + 1;
  print y;
}
20
21
%
```

こんどは確かに、プログラムを全部打ち込み終ったあとで実行している感じがする。

### 3.4 while 文の追加

それでもやっぱり、ループとかないとプログラムらしくない。そこでとりあえず while 文が使えるようにしてみる。まず、yacc 記述で文の 1 種として while 文と複合文を追加する。

```
        | WHILE '(' cond ')' stat       { $$ = node(T_WHILE, $3, $5); }
        | '{' stlist '}'        { $$ = $2; }
```

もちろん、最初の%token WHILE; 宣言も忘れないように。なお、複合文はこれがないとループ本体に文が 1 つしか書けなくてつまらないから。あと、条件部はさぼって次のものだけにする。

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

当然、lex の記述も WHILE と<と>を返すように直す。また C ソースの頭の T_なんとかの define も追加する。最後は dotree にこれらの処理を追加するだけ。

```
case T_WHILE:  while(dotree(ntab[i].left)) dotree(ntab[i].right); break;
case T_LT:     return dotree(ntab[i].left) < dotree(ntab[i].right);
case T_GT:     return dotree(ntab[i].left) > dotree(ntab[i].right);
```

じゃあ実行しよう。

```
% lex t8.lex
% yacc t8.yacc
% cc t8.c
% a.out
main {
  x = 0;
  while(x < 10) {
    x = x + 1;
    print x; }
}
1
2
3
4
5
6
7
8
9
10
%
```

ぱちぱちぱち。

## A 本日の練習問題兼出席

本日は課題が出ますので、練習問題はなし。といっても、例によって「例題を改造せよ」という課題ですから、まずは例題を打ち込んで追試するのがよいと思います。

ともかく時間中好きなことをやって下さって結構ですが、時間になったらいちばん最後に完成した (実行可能な) ものの yacc 記述と実行例をプリントアウトして最後の紙の裏に以下のもの:

- 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. あなたはこういうの (言語処理系) は好きですか。もし好きなら、もっとどんなことがやってみたいですか。

  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。

## B Lex/Yacc 編の課題

さて、いよいよレポート課題です。自分の実力に応じて、以下の課題群から最低 3 つ以上選んでください。やさしいのも難しいのもありますが、自分の実力に応じているかどうかは過去 3 回の出席を見るとわかるので、不当に易しいものを選ばないように。(といって、無理に難しいのを選んで自爆するのも意味ありませんから。あくまで適切にね。)

**課題 A:** 「小さな言語」で `if` 文が使えるように処理系を改造してみてください。できれば、else 部はあってもなくても大丈夫なのが望ましいです。

**課題 B:** 「小さな言語」で `repeat-until` 文 (Pascal にできるだけ近い形の) が使えるように処理系を改造してみよ。

**課題 C:** 「小さな言語」で配列が使えるように処理系を改造してみよ。[1]

**課題 D:** 「小さな言語」でサブルーチンコールが使えるように処理系を改造してみよ。呼び出しの構文などは自由に決めて良い。パラメタはとりあえず不要。[2]

**課題 E:** サブルーチンではなく、関数 (というのは、式の中に書けて値が返せる) にしてみたらどうか。

**課題 F:** もちろん、パラメータが渡せるようになっていたらすばらしい。

**課題 G:** ちょっと変な構文ですが、if 文を次のような構文のものとして実現してみよ。

`if` 条件 `then` 文の並び `elif` 条件 `then` 文の並び `...` `else` 文の並び `fi`

こういう if 文にすると何かいいことがあるかどうかも検討すること。

**課題 H:** これら以外に、普通の言語にないようなオリジナルな (奇想天外な) 構文ないし機能を考案して、使えるようにしてみよ。

**課題 I:** 木構造を print する例をちょっと改造して、C のソースコードを出力するようにしてみよ。そんなに難しくないです。もちろん C コンパイラで翻訳して実行させてみること。[3]

[1] とりあえずひどくさぼって、すべての変数は配列としても使え、ただし大きさは 10 要素だけである、とかいうのでもいいです。

[2] たとえば stab の val 欄に、その名前のサブルーチンは ntab の何番目の要素を根本に持つ木かを登録するという方法が可能でしょう。ただしサブルーチン名と同じ変数名を使うと破滅しますね。

[3] とりあえず簡単に済ますため、実行させる時に最初に `main() {` と変数宣言、最後に `}` 、を手で追加する、ということでもいいです。

なお、レポートの形式は次の通りとする。これを守らないと提出しても受理されないことがあるので、注意して欲しい。

- A4 版の用紙を縦に使うこと。
- ホチキス等でばらばらにならないよう閉じること。
- 必要にして十分な実行例を含めること。
- 内容構成は次の順とする。

  1. 表紙。レポートタイトル (S 実験'95 課題 S3 Lex/Yacc 編レポート、ですね)、学籍番号、氏名、所属、提出日付のみを記述する。
  2. 課題内容。どんな課題をやったか。
  3. 方針。下敷きのプログラムをどう直そうと思ったか。また、自分固有の機能についてはなぜそういうのがいいと思ったか、どう実現しようと思ったか、など。
  4. 回答。実際にどういうプログラムにしたか。実行例。
  5. 考察。やってみて分かったこと。採点においては考察の内容が重視されます。手を抜かないように。また「感想」ばっかりでもだめよ!
  6. 付録。プログラムリストが長い場合には付録にしてください。

これまでの経験では、考察が手抜きなために B-- (というのは B つまり「並」の半分くらいの得点です) を差し上げる方が多いですから注意してください。〆切については各群とも一律、課題提示日 (今日) から 2 週間後同一曜日の午後 1 時とします。