S実験'95　SA (LexとYacc) ＃4

久野　靖 *

1995.5.31, 1995.6.21

この課題 (SA) は、S コースの人のみの特別メニューであり、より進んだ内容をやります。前提として、lex と yacc はマスターしているものと仮定しますので、そうでない場合は急遽復習すること。

4　シェルの作り方

4.1　インタプリタ言語の意義

さて、S3 でやった課題では最終的に小さいが C「みたいな」言語のインタプリタを作成し、それを改良していただいた。しかし、C 言語があるのになんでいまさらインタプリタを作るのかと言われると釈然としませんね?

実は、lex と yacc でインタプリタを作るのは目新しい言語、普通の言語にない組み込み機能がついた言語を作ってみる場合が多い。そこで今回は (課題 S1 と S2 でやったことの復習を兼ねて)「シェル言語」を作ってみることにしよう。つまり、「プログラムを実行させる」という特殊な組み込み機能をもった言語のインタプリタを作成するわけである。(シェルだってもうあるんだから作らなくても、という突っ込みは却下します。)

ではさっそく、yacc の記述から行ってみよう。

```
%token WORD;
%%
prog    :
        | prog words '\n' { docmd($2); prompt(); }
        ;
words   :               { $$ = 0; }
        | word words    { $$ = node(T_WORDS, $1, $2); }
        ;
word    : WORD          { $$ = intern(yytext); }
        ;
```

ほとんど説明の必要はありませんね? シェルではコマンドというのは「語」の並びで、1 行ごとに (リターンが来るごとに) 実行されるから、これでいい。なお、intern というのは初出だが、実は lookup を簡単にしたものと思えばいい。一応、C のソースを途中までつける。

```
#include <stdio.h>
#include <string.h>       ←この辺の
#include <fcntl.h>        ← include の山は、
```

*筑波大学大学院経営システム科学専攻

```
#include <sys/types.h>   ←システム機能の
#include <sys/wait.h>    ←呼び出しに必要
#define TRUE    1
#define FALSE   0
#define yywrap() 1
extern char yytext[];
char *wtab[1000];        ←各 WORD の文字列ポインタを入れる
int wtabuse = 1;         ← wtab をどこまで使ったか
struct node {            ←これは前とおなじ
    int type, left, right; } ntab[400];
int ntabuse = 1;
#define T_WORDS 1
#include "y.tab.c"
#include "lex.yy.c"

yyerror(s)               ←前とおなじ
    char *s; {
    printf("%s\n",s);
}

main() {
    prompt(); yyparse();  ←「>」を打ってから始める
}

prompt() {               ←単に「>」を打つだけ
    fprintf(stderr, "> "); fflush(stderr);
}

node(t, l, r)            ←これは前とおなじ
    int t, l, r; {
    int i = ntabuse++;
    ntab[i].type = t; ntab[i].left = l; ntab[i].right = r;
    return i;
}

intern(s)
    char *s; {
    int i;
    for(i = 1; i < wtabuse; ++i)
      if(strcmp(wtab[i], s) == 0) return i;
    wtab[wtabuse] = (char*)malloc(strlen(s)+1);
    strcpy(wtab[wtabuse], s);
    return wtabuse++;
}
```

このように、wtab には打ち込んだ文字列を保管しておいて、プログラムの残りの部分ではこの「文字列番号」(wtab の何番目か) を使用する (図 1)。既に同じ文字列がある場合はその文字列番号を

返す (この方が節約になるからで、面倒ならそうしなくてもいい)。まだない場合は、文字列を格納するための領域を malloc で割り当て、そこに文字列をコピーする。

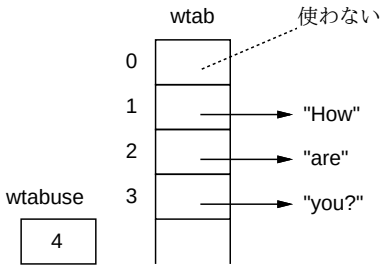

図 1: wtab の構造

4.2 exec で他のプログラムを起動する

さて、いよいよ今回の「きも」である docmd を見るのだが、その前に exec システムコールの復習をしておこう。今回使うのは execvp であり、その呼び方は次の通り (man execvp で確認しておくこと)。

```
execvp(コマンド名, 引数配列);
```

引数配列は文字列 (文字へのポインタ) の配列で、0 番目はコマンド名、1 番目以降に各引数を入れ、最後の引数の次には 0(NULL ポインタ) を入れておく。例えば「ls -l a.out」だと図 2 のようになる。

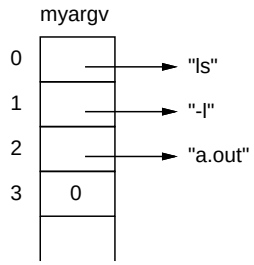

図 2: execvp の引数

結局、docmd がやるべきことは、「語の並び」の木構造をたどって各語を文字列配列に入れて、それから execvp を呼び出すだけである (0 番目の語と第 1 引数のコマンド名は兼ねている)。

```
docmd(i)
    int i; {
    char *myargv[100];
    int myargc = 0;
    while(i) {
      myargv[myargc++] = wtab[ntab[i].left]; i = ntab[i].right; }
    myargv[myargc++] = 0;
    if(myargc > 1) {
       execvp(myargv[0], myargv);
```

```
        fprintf(stderr, "command %s cannot be exec'ed.\n", myargv[0]);
        exit(1); }
    exit(0);
}
```

なお、execvp は戻ってこないはずだが、戻ってきた場合は何らかのエラーなのでその旨打ち出す。ついでに、引数が 0 個の場合は何も言わないで終わるようにした (理由は後で再度説明する)。

では実行、の前に lex も必要なので挙げておく。

```
letter  [^<>()&;|\n\t ]
delim   [<>()&;|\n]
white   [\t ]
%%
{delim}         { return yytext[0]; }
{letter}+       { return WORD; }
{white}         { ; }
```

いろいろな記号を特別扱いにしているが、これはシェルをやるための伏線である (そのたびに lex ファイルを直すのは面倒だから)。

では実行してみよう。

```
% lex t10.lex
% yacc t10.yacc
% cc t10.c
% a.out
> ls -l a.out
-rwx------  1 kuno     54484 May  8 15:57 a.out
%
```

確かに、プログラムは「>」というプロンプトを出し、そこでコマンドを打ち込むと実行するが、それで終りになってシェルへ戻ってしまう。というのは、exec 系のシステムコールは現在のプロセスが「別のコマンドに成り替わってしまう」効果をもたらすので、ようは a.out が ls -l に成り替わってしまい、それが済むとプロセスの実行が終わってしまうわけである (図 3)

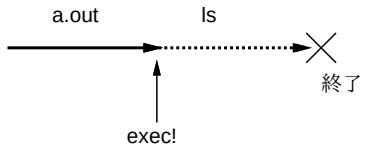

図 3: exec の意味

4.3 fork で親子に分裂する

では、次々とコマンドを実行するのはどうしたらいいか? それには、図 4 のように「まず fork() して親と子に分裂し、親は子の完了を待ち、子は指定されたコマンドに成り替わって実行」すればいい。実はこう直すのはとっても簡単で、さっきの docmd を excmd という名前に変更して、docmd は次のものにすればいい。

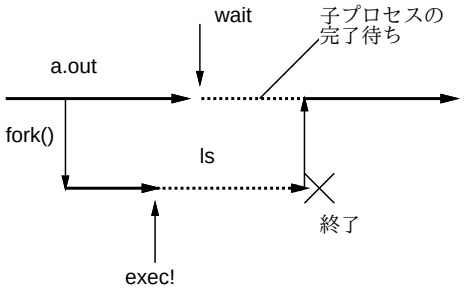

図 4: fork による分岐と wait による待ち合わせ

```
docmd(i)
    int i; {
    int pid;
    if(pid = fork())
       wait4(pid, 0, 0, 0);
    else
      excmd(i);
}
```

なお、wait4 というのは指定したプロセスのみを待ち合わせるのに使える wait である。(残りの引数で色々指定できるが、ここでは使わない。) ではこれで動かしてみよう。

```
% cc t11.c     ← C 以外は同じ
% a.out
> ls -l a.out
-rwx------  1 kuno        59252 May  8 16:07 a.out
> date
Mon May  8 16:07:42 JST 1995
> cp a.out *
> ls
*        a.out     t11.c       lex.yy.c    y.tab.c
> rm *
> ls
a.out    t11.c     lex.yy.c    y.tab.c
>^D              ←^D で終わる
%
```

このように、「*」などという普通のシェルでは特殊記号になっている文字も、特別扱いしなければ「ただの字」だということがわかる。

練習 1 この小さなシェルを打ち込んで動かしてみよ。

練習 2 excmd の最後の「exit(0);」はなぜ必要か? もしなかったらどうなるか? 取ってしまって、空っぽのコマンド行を打ち込んだらどうなるかやってみよ。

4.4 リダイレクション

さて、Unix の「fork と exec の組合せで新しいプロセスを作る」というのはなかなか面白いアイデアである。というのは、fork してから exec するまでの間の子プロセスで入出力にいろいろな細工を行えるからである。例えば、入出力リダイレクションはそうやって作られている。さっそく、我々の小さなシェルにも出力リダイレクションを入れてみよう。まず文法をちょっと増やす。

```
%token WORD;
%%
prog    :
        | prog redir '\n' { docmd($2); prompt(); }
        ;
redir   : words           { $$ = $1; }
        | words '>' word  { $$ = node(T_OREDIRECT, $1, $3); }
        ;
(以下同じ)
```

そして、excmd を次のように直す。

```
excmd(i)
    int i; {
    if(ntab[i].type == T_OREDIRECT) {
      int fd1;
      close(1);
      fd1 = open(wtab[ntab[i].right], O_WRONLY|O_CREAT|O_TRUNC, 0644);
      if(fd1 < 0) { perror("open"); exit(1); }
      excmd(ntab[i].left); }
    else {
      char *myargv[100];
      int myargc = 0;
      while(i) {
        myargv[myargc++] = wtab[ntab[i].left]; i = ntab[i].right; }
      myargv[myargc++] = 0;
      if(myargc > 1) {
         execvp(myargv[0], myargv);
         fprintf(stderr, "command %s cannot be exec'ed.\n", myargv[0]);
         exit(1); }
      exit(0); }
}
```

後半はさっきと同じで、増えているのは T_OREDIRECT の枝だけである。ここでは何をやっているかというと、1 番のファイルディスクリプタ (標準出力) を閉じて、「>」の右に指定したファイルを open している。open はできるだけ小さいファイルディスクリプタを開こうとするので、1 番を閉じた直後にやると 1 番が使われる (図 5)。ということは、指定したファイルが標準出力につながるわけだ。動かしてみよう。

```
% yacc t12.yacc
% cc t12.c
% a.out
```

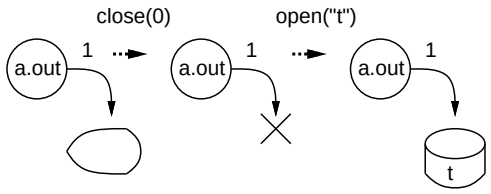

図 5: close と open によるリダイレクション

```
> echo a b c >t
> cat t
a b c
>^D
%
```

練習 3 さっきの版をこのように直して動かせ。

練習 4 入力リダイレクションもできるようにしてみよ。

4.5 パイプラインの作り方

リダイレクションができるようになったら、パイプラインあと 1 歩である。さっそく、文法から。

```
%token WORD;
%%
prog    :
        | prog pipe '\n'  { docmd($2); prompt(); }
        ;
pipe    : redir           { $$ = $1; }
        | pipe '|' redir  { $$ = node(T_PIPE, $1, $3); }
        ;
(以下同じ)
```

こちらは簡単ですよね。問題は実行だが、やはり excmd の中だけ増やせば済む。増やすところだけ示そう。

```
(前の方は略)
if(ntab[i].type == T_PIPE) {
  pipe(p);
  if(pid = fork()) {    ←***
    dup2(p[0], 0); close(p[1]); close(p[0]); excmd(ntab[i].right); }
  else {
    dup2(p[1], 1); close(p[1]); close(p[0]); excmd(ntab[i].left); } }
else if(ntab[i].type == T_OREDIRECT) {
(以下同じ)
```

つまり、まずパイプを作り、次に親と子に別れて、親はパイプの出口をディスクリプタ 0 番にしてから中のコマンドを実行する。子はパイプの入り口をディスクリプタ 1 番にしてから中のコマンドを実行する (図 6)。簡単でしょう? これでちゃんと、N 段のパイプラインだってできる。

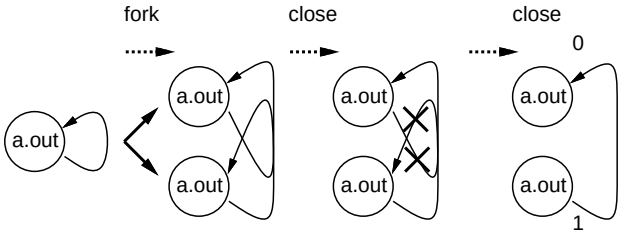

図 6: パイプラインの作り方

```
% yacc t13.yacc
% cc t13.c
% a.out
> ps ax | head -5 | tr e E
  PID TT STAT TIME COMMAND
    0 ?  D    0:34 swappEr
    1 ?  S    0:00 init -
    2 ?  D    0:00 pagEdaEmon
    3 ?  D    0:01  (syncd)
>^D
%
```

練習 5 さっきの版をこのように直して動かせ。

練習 6 ＊＊＊の条件を逆にすると何が起きるか。まず予想し、つぎに実際にやって確かめよ。

A 本日の練習問題兼出席

本日の練習問題は、「練習 x」と書かれたものを順にやっていただくことです。時間が余った人はぜひ、以下のもやってみてください。このうちいくつかは次々回に解説するかも知れませんし、そのままレポート課題になるかも知れません。時間のない人は無理しないで結構です。

練習 7* 本物のシェルにあってこの小さなシェルにない機能を入れてみよ。例えば次のものはどうか。(適当に選ぶ。)

- 「&」や「;」に相当する機能。
- かっこの機能。つまり「(ps; ls) >t」みたいなの。
- クォート機能 ("や')。
- コマンド置換 (‘)。
- while や if などの制御構造。
- シェル変数。

練習 8* 逆に、本物のシェルにないユニークな機能を考えてこの小さなシェルに入れてみよ。その使いやすさはどうか?

だいたい時間になったら、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. あなたは今回取り上げたシェルの機能をどれくらい知っていましたか。また、シェルのなかでそれらの機能をどうやって実現しているかはどれくらい知っていましたか。

  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。