プログラミング環境 第13回

久野 靖 *

1992.1.28

20 プログラムを作り出すプログラム

20.1 自前でシェルを作る?

延々とやってきたこの「プログラミング環境」もようやくここに無事最終回を迎えることとあいなった。めでたし、めでたし。さて、ずっと最初の方で「コマンドインタープリタ」、またはUnixの言葉でいえば「シェル」について色々やったのをまだ覚えておいでかと思うが... 今や、皆様はシェルが結局何をやっているかをかなり良く知っていることに気づかれるはずである。すなわち、シェルは

1. 端末から (または標準出力から)1 行ぶんの文字の列を読む。
2. それを「指令」「引数 1」「引数 2」... のように分ける。
3. fork() を用いて 2 つのプロセスに分裂する。
4. 子プロセスの方では exec() を用いて「指令」のプログラムに成り換わる。
5. 親プロセスの方は wait() で「指令」が終るのを待つ。

というのを繰り返し実行しているだけである。じゃあ早速、自分でもそういうプログラムを書いてみよう!

```
#include <stdio.h>

#define BUFSIZE 1024
#define ARGSIZE 64

char buf[BUFSIZE];        -- コマンド入力の 1 行が入るバッファ
int bufc = 0;             -- 何文字目まで入れたか数える
char *argv[ARGSIZE];      -- 名前の通り argv. 各引数へのポインタ
int argc = 0;             -- いくつ目まで引数を入れたか数える

#define FALSE 0
#define TRUE 1

main() {
  int c, inword = FALSE;
  while((c = getchar()) != EOF) {               -- 1 文字よむ
    if(c != '\n' && c != '\t' && c != ' ') {    -- 語の切れ目以外なら
      if(!inword) argv[argc++] = buf + bufc;    -- 語の先頭なら
      inword = TRUE; buf[bufc++] = c; }         -- 語の各文字を蓄える
    else {                                      -- 語の切れ目
      if(inword) buf[bufc++] = 0;               -- 語の終りなら
      inword = FALSE;
```

*筑波大学経営システム科学専攻

```
        if(c == '\n') {                           -- 改行だ!
          int pid = fork();
          if(pid != 0) {
            pid = wait(0); argc = bufc = 0; }     -- 親なら待つだけ
          else {
            argv[argc++] = 0; execvp(argv[0], argv);  -- 子なら exec
            perror("exec fail"); exit(1); }
        }
      }
    }
  }
```

ちょっと長いかな、と思われるかも知れないが、まあ大したことはない。これでちゃんとシェルになっている。見てみよう。

```
% cc t21.c
% a.out
ls
a.out   t21.c
ls -l
total 25
-rwxr-xr-x  1 kuno        24576 Jan 25 16:36 a.out
-rw-r--r--  1 kuno          670 Jan 25 16:36 t21.c
cp t21.c *
ls
*       a.out   t21.c
rm *
ls
a.out   t21.c
cp t21.c 'How are you?'
Usage: cp [-ip] f1 f2; or: cp [-irp] f1 ... fn d2
^D
%
```

しかし、この通りこの「シェル」は特殊文字の解釈とか引用符の処理など一切やってくれない。まして、パイプラインとかバックグラウンドなんて... もちろん、それをやるようにがりがりプログラムを書くことはパワフルな人には可能だろうけど、でも大変ですよね? しかし、シェルが受け付ける入力というのは実は「正則式」とか「文法」で現せるような規則的なものであり、そういうの扱うプログラムは、がりがり手で書く代わりに計算機に作り出させてしまう、というのがスマートだとされている。そこで今回のテーマ「プログラムを作り出すプログラム」とあいなるわけである。

20.2 lex – 正則式を処理するプログラムの生成

20.2.1 lex の基本

さて、その 1 番手は lex というプログラムであり、正則式 (というか、パターン) を処理するプログラムを生成するのに適したツールである。ときに皆様は awk のプログラムが

```
パターン { 動作 ... }
パターン { 動作 ... }
...
```

という形をしていたのをご記憶だろうか。実は lex のプログラムもそれによく似ていて、

```
定義...
%%
パターン { 動作 ... }
```

```
パターン { 動作 ... }
...
%%
おまけのコード...
```

という形をしている。ただし、最初の「定義」は空でもいいし、2番目の%%以下はなくてもいいので、結局一番基本的な lex のプログラムというのは awk のプログラムにそっくりである。ただ異なる点は、awk がそのプログラムをその場で実行するのに対し、lex は C のプログラムを生成する、という点と、あと「動作」の部分は (最後は結局 C になるんだから) 実は C のコードを記す、という点である。例えば次の lex プログラムを見ていただこう。

```
%%
[^ \t\n]+       { printf("<%s>", yytext); }
\n              { printf("\n"); }
.               { ; }
```

定義の部分がないのでいきなり%%から始まる。まず「空白でもタブでも改行でもない文字が 1 個以上」あった場合には、その並び (あ、実は「動作」が実行される時は yytext という配列にパターンにマッチした文字列が入っていることになっているのです) を<>で囲んで打ち出す。改行だったら出力も改行する。それ以外の場合 (. というのは「任意の字」だったよね!) は何もしない。というプログラムである。[1] 次にこれを動かす場合には、図 1 にあるように、まずなんとか.lex という

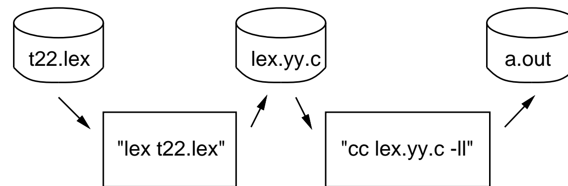

図 1: lex ソースの実行しかた

ファイルに入れたソースを lex に食べさせると lex.yy.c というプログラムが作られ、それをもう一度 C コンパイラに食べさせると (ただし-ll というオプションが必要である)、普通の a.out ができる、という手順による。では動かしてみよう。

```
% lex t22.lex
% wc lex.yy.c
     289     923    6582 lex.yy.c
% cc lex.yy.c -ll
% a.out
This is a pen.
<This><is><a><pen.>
How are you?
<How><are><you?>
^D
%
```

というわけで、さっきのプログラムのように語の切れ目を認識するのを自前でやる必要はなくなるわけである。

20.2.2 lex と C の組み合わせ方

では次に、これを利用するとさっきのシェルはどんな風になるかやってみよう。

[1]ときに「任意の字」には「空白でもタブでも改行でもない字」や「改行」なども含まれるわけだが、そういう風に重複がある場合には上に書かれた方が優先する。

```
%%
[^ \t\n]+       { accum(); }
\n              { doit(); }
.               { ; }
%%
#define BUFSIZE 1024
#define ARGSIZE 64

char buf[BUFSIZE];
int bufc = 0;
char *argv[ARGSIZE];
int argc = 0;

accum() {
  argv[argc++] = buf + bufc;
  strcpy(buf + bufc, yytext);
  bufc += strlen(yytext) + 2;
}

doit() {
  int pid = fork();
  if(pid != 0) {
    pid = wait(0); }
  else {
    argv[argc++] = 0; execvp(argv[0], argv);
    perror("exec fail."); exit(1); }
  bufc = argc = 0;
}
```

つまり、語が来ると accum を呼んでその語を配列 buf にコピーしてその分 bufc を増やす。改行だったら例によって fork して exec する。このように、「おまけ」の部分は必要な変数の宣言とかサブルーチンなどを書くためにあったわけである。そして「動作」の部分からはそういう必要なルーチンを呼び出せばよい。t21.c と比べてどうですか?一応実行例も示す。

```
% lex t23.lex
% cc lex.yy.c -ll
% a.out
ls
a.out           lex.yy.c        t23.lex
^D
%
```

20.2.3 lex はどうありがたいか?

さて、lex がパターンの認識を自動的にやってくれるから、これを使えばエスケープ文字や引用符の処理を付け加えるのもわりと簡単である。次のは先のプログラムをそのように改良したもんである。

```
%%
'[^']*'         { yytext[strlen(yytext) - 1] = 0; accum(yytext + 1); }
\\.             { accum(yytext + 1); }
[^ \t\n\\']     { accum(yytext); }
\n              { endword(); doit(); }
.               { endword(); }
%%
/* buf, argc などの宣言はさっきと同じ */
#define FALSE 0
#define TRUE 1
```

```
int inword = FALSE;

accum(s)
  char *s; {
  if(!inword) argv[argc++] = buf + bufc;
  inword = TRUE;
  strcpy(buf + bufc, s);
  bufc += strlen(s);
}

endword() {
  if(inword) bufc += 1;
  inword = FALSE;
}

/* doit はさっきと同じ */
```

つまり、エスケープ記号+次の文字、とか引用符で囲まれた部分をそれぞれパターンとして認識して処理するように直したわけである。その代わり、1 つの語がそのような「部分語」いくつかを並べたものになるので、accum はこれまでの語の後ろに渡された部分語を追加するようになり、また語の中かどうかを覚えておく旗が要るようになってしまった。それでも、t21.c を直接直すよりはずっと分かりやすいと思う。では実行例。

```
% lex t24.lex
% cc lex.yy.c -ll
% a.out
ls
a.out           lex.yy.c        t24.lex
cp t24.lex 'How are you?'
ls
How are you?    a.out           lex.yy.c        t24.lex
rm How\ are\ you?
ls
a.out           lex.yy.c        t24.lex
^D
%
```

20.3 yacc – 文脈自由文法を処理するプログラムの生成

20.3.1 yacc の基本

前の節でやったことをもう一度まとめると、入力が「正則式」という「規則」に基づいてうまく「認識」できるような場合には、その「認識」の部分は lex にまかしてしまい (実はこれが自前でやると結構大変)、自分はそれぞれのパターンが認識されたあとの「動作」のみを記すことに専念することができる、ということであった。ところで、「正則式」というのは比較的単純な (入れ子構造を持たない) 入力を記述するためのものなので、例えばシェルの `;` `&` `|` などを処理しようと思うとうまく行かない。でも、既にやったように BNF でなら次のように書ける。

プログラム ＝空 │ プログラム 指令
指令 ＝文
文 ＝単純文 │ 文 | 文 │ 文 ; 文 │(文)
単純文 ＝空 │ 単純文 語

こういう、BNF で書き表すようなものを処理したい場合には、今度は yacc というのを使う。さっそく、yacc を使ってコマンドを認識する、というのをやってみよう。

```
%start  prog
%token  word
%left   '|'
%left   ';'
%%
prog    :
        | prog comd
        ;
comd    : stat '\n'
        ;
stat    : simple
        | stat '|' stat
        | stat ';' stat
        | '(' stat ')'
        ;
simple  :
        | simple word
        ;
%%
#include "lex.yy.c"
main() { yyparse(); exit(0); }
yyerror(s) char *s; { fprintf(stderr, "%s\n", s); }
```

yacc のソースも lex と同様、定義、パターン+動作、おまけの 3 つの部分から成る。まず定義部分ではファイル全体が 1 つの「prog」に対応していること、word というのは yacc に渡す入力部分が用意してくれる単位であること (先の日本語版文法でも「語」というのは定義されていないことに注意)、; と | はともに左からグループ化され (つまり、`a ; b ; c` は `(a ; b) ; c` の意味、なおかつ | の方が順位が高い (つまり、`a ; b | c` は `a ; (b | c)` の意味になる) ことを定義している。次のパターン+動作部分は見ての通り普通の「文法」そのものである (ただし ＝は : で現され、各規則の終りに ; が必要でる)。まだこの例では動作は一つもない。最後のおまけでは、とりあえず `main` と `yyerror` という関数を用意する必要がある。その他の下請けはこの例では無いが、ただし「語」などを認識するのは lex を使うので lex の出力である `lex.yy.c` を取り込むようになっている。

次に、これといっしょに使う lex のソースを示す。

```
%%
[^ \t\n;|()]+   { return word; }
[\n;|()]        { return yytext[0]; }
.               { ; }
```

今度は lex は yacc に「語」とかその他必要な文字 (';' など) を値として渡さなければいけないので、そのために動作の部分に `return` を書く。簡単のためここでは脱出文字などの処理はしないことにした。よって、まず区切り記号以外の並びの場合には「語が来た」と教え、改行と `; | ( )` の場合にはその文字そのものを返し (その文字は配列 `yytext` の先頭に入っていることに注意)、それ以外の文字は無視する。

これらを組み合わせて実行するには図 2 のようにする。すなわち、yacc が生成する C のプログラムは `y.tab.c` というファイルにできるので、それを cc でコンパイルするが、それに先だって lex ソースも変換しておけば `lex.yy.c` も `#include` でとりこまれるのでまとめて翻訳されることになるわけである。さて、これを実行してみよう。

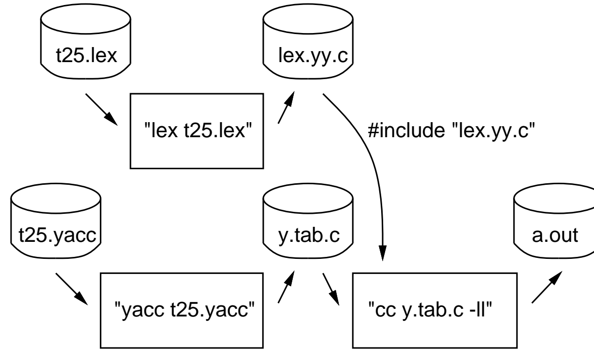

図 2: yacc ソースの実行しかた

```
% yacc t25.yacc
% wc y.tab.c
     218     854    5052 y.tab.c
% lex t25.lex
% cc y.tab.c -ll
% a.out
ls -l
date ; who | wc
(date ; who) | wc
(date ; who | wc
syntax error
%
```

このプログラムは「認識する」だけだから、正しいプログラムを入れれば何も言わず、間違ったプログラム (かっこが対応してないだけだが) を入れれば `syntax error` といって終ってしまう、それだけである。

20.3.2 抽象構文木とそのデータ構造

さて、実際に役に立つ仕事をするには認識した結果に対応する木構造 (抽象構文木) をプログラム内部でデータ構造として組み立てると便利であるので、まずはそれをやることにしよう。つまり図 3 上のような入力に対して下のような木構造を表すデータ構造を作るわけである。具体的なデー

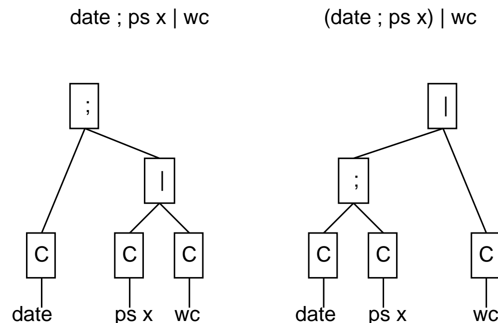

図 3: 入力と抽象構文木

タ構造はここでは、次のようにする。まず、入力の各語の文字は先にやったように配列 `buf` に順に詰めていく。またその各語へのポインタも先にやったように配列 `argv` に順に詰めていく (指令の終

りには 0 を入れる)。そして指令と指令の組み合わせを記憶するために新たに (kind,left,right) という 3 つの欄を持つレコードの配列 tbuf を作り、そこに図 3 でいうと「四角い箱」に相当する情報を持つ。具体的には kind 欄に箱の種類を入れ、left,right に左と右の枝の情報を入れる (指令の場合は枝が一つで足りるので left だけを使う)。図 3 左に対応するデータ構造を図 4 に示す。

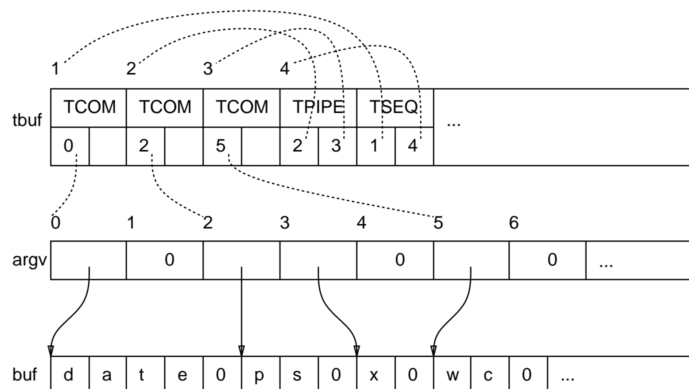

図 4: 図 3 左に対応する具体的なデータ構造

さて、これに対応する宣言の枠組みを示そう。

```
#define BUFSIZE 1024
#define ARGSIZE 64
#define TREESIZE 64

char buf[BUFSIZE];
int bufc = 0;
char *argv[ARGSIZE];
int argc = 0;
struct tree {
  int kind, left, right; } tbuf[TREESIZE];
int tbufc = 0;

#define T_COM 0
#define T_SEQ 1
#define T_PIPE 2
```

次に、各単語 (これは lex によって配列 yylex に用意される) を buf と argv に詰めていく部分を示す。これは散々やったパターンである。

```
makearg() {
  return argc;
}

addarg() {
  argv[argc++] = buf + bufc;
  strcpy(buf + bufc, yytext);
  bufc += strlen(yytext) + 1;
}

endarg() {
  argv[argc++] = 0;
}
```

makearg は単に次の空いている argv の位置を返す。addarg は yylex から次の語をコピーすると同時に argv の次の場所にその先頭アドレスを入れる。endarg は指令 1 つの終りの印として argv の次の場所に 0 を入れる。

さて、次は木のノード、つまり tbuf の各要素を作る関数である。が、これは簡単で、種類と枝の情報を入れて tbufc を進めるだけである。ただし戻り値としては進める**前の** tbufc の値を返す。

```
makesimp(x) int x; {
  tbuf[tbufc].kind = T_COM; tbuf[tbufc].left = x; tbuf[tbufc].right = 0;
  return tbufc++;
}

makeseq(x, y) int x, y; {
  tbuf[tbufc].kind = T_SEQ; tbuf[tbufc].left = x; tbuf[tbufc].right = y;
  return tbufc++;
}

makepipe(x, y) int x, y; {
  tbuf[tbufc].kind = T_PIPE; tbuf[tbufc].left = x; tbuf[tbufc].right = y;
  return tbufc++;
}
```

さて、こうして木構造ができたとして、それを確認する方法がないとつまらないので、あいまいさがないように全ての場所にかっこをつけて出力する、という関数を用意した。

```
dotree(x) int x; {
  ptree(x);
  printf("\n");
  bufc = argc = tbufc = 0;
}

ptree(x) int x; {
  if(tbuf[x].kind == T_COM) {
    int i = tbuf[x].left;
    while(argv[i] != 0) printf(" %s ", argv[i++]); }
  else if(tbuf[x].kind == T_SEQ) {
    printf(" ("); ptree(tbuf[x].left); printf(";");
    ptree(tbuf[x].right); printf(") "); }
  else if(tbuf[x].kind == T_PIPE) {
    printf(" ("); ptree(tbuf[x].left); printf("|");
    ptree(tbuf[x].right); printf(") "); }
}
```

20.3.3 yacc の動作指定

これで「下請け」関数は全部そろったので、さっきの yacc ソースに「動作」を入れてこれらを呼び出すようにしよう。

```
%start  prog
%token  word
%left   ';'
%left   '|'
%%
prog    :
        | prog comd
        ;
comd    : stat '\n'     { dotree($1); }
        ;
stat    : simple        { endarg(); $$ = makesimp($1); }
        | stat '|' stat { $$ = makepipe($1, $3); }
        | stat ';' stat { $$ = makeseq($1, $3); }
        | '(' stat ')'  { $$ = $2; }
        ;
simple  :               { $$ = makearg(); }
```

```
        | simple word   { addarg(); $$ = $1; }
        ;
%%
#include "lex.yy.c"
main() { yyparse(); exit(0); }
yyerror(s) char *s; { fprintf(stderr, "%s\n", s); }

/* ここにさっきの宣言と関数を入れる */
```

「動作」の中に`$1`とか`$$`とか出てくるのが目につく。実は yacc では文法の中に出てくる各要素 (stat とか simple とか) に値を持たせることができる。そして、「動作」のなかで`$`nと書くと、その文法規則の右辺に出てくる n 番目の要素の値を参照することができ、また`$$`に代入するとそれが左辺の要素の値になる、というしかけで値を操作できるようになっている。実際に図 3 の実例を認識させると図 4 のデータ構造ができる様子を、文法と動作を組みにして追いかけて見るとよくわかるはずである。では実行例 (ちなみに lex の方は変えてないので `lex.yy.c` はそのまま使える)。

```
% yacc t26.yacc
% cc y.tab.c -ll
% a.out
date ; ps x ; who
 ( ( date ; ps  x ) ; who )
date ; ps x | wc
 ( date ; ( ps  x | wc ) )
date ; ( ps x | wc )
 ( date ; ( ps  x | wc ) )
^D
%
```

確かに、; だけだと左からグループ化されるが | が混ざると両方の順位によって変化し、しかしかっこを使うとそれが変更できる様子が分かる。

20.3.4 抽象構文木の解釈実行

さて、ここまで来れば実は表示する代わりにシェルのように動作させるのはとっても簡単!である。実は `dotree` 以下を次のように直すだけでよい。

```
dotree(x) int x; {
  int pid = fork();
  if(pid == 0) extree(x);
  pid = wait();
  bufc = argc = tbufc = 0;
}

extree(x) int x; {
  int a[2], pid;
  if(tbuf[x].kind == T_COM) {
    int i = tbuf[x].left;
    if(argv[i] == 0) exit(0);
    execvp(argv[i], argv + i);
    perror("extree: exec fail."); exit(1); }
  else if(tbuf[x].kind == T_SEQ) {
    if((pid = fork()) == 0) extree(tbuf[x].left);
    pid = wait(0); extree(tbuf[x].right); }
  else if(tbuf[x].kind == T_PIPE) {
    pipe(a);
    if((pid = fork()) == 0) {
      dup2(a[1], 1); close(a[1]); close(a[0]);
      extree(tbuf[x].left); }
    else {
```

```
        dup2(a[0], 0); close(a[1]); close(a[0]);
        extree(tbuf[x].right); } }
}
```

つまり dotree は分裂してから子プロセスの側で実行したいプロセス (群) に成り換わるための下請け関数 extree を呼ぶ。extree の中では普通の指令ならそのまま成り換わる。; であればまず分裂して左の指令を子プロセスで実行してから右を実行する。そして | であれば分裂して両側でパイプを適切に配線してから左右の枝をそれぞれ実行する。簡単でしょう? 実行例も示そう。

```
% yacc t27.yacc
% cc y.tab.c -ll
% a.out
date ; ps x ; who
Sat Jan 26 21:25:44 JST 1991
  PID TT STAT  TIME COMMAND
 8376 p2 I     0:04 -usr/new/csh (csh)
 9428 p2 S     0:00 a.out
 9429 p2 S     0:00 a.out
 9430 p2 R     0:00 ps x
kuno     ttyp0   Jan 26 13:59   (smr03:0.0)
hanaoka  ttyp1   Jan 26 20:09   (sme00)
hanaoka  ttyp4   Jan 26 20:59   (sme00)
date ; who | wc
Sat Jan 26 21:26:01 JST 1991
       3      18     118
(date ; who) | wc
       4      24     147
^D
%
```

20.3.5 抽象構文木の翻訳

さて、上のプログラムはデータ構造をたどりながら実行していた... つまりインタープリタなわけね。で、そうする代わりに実行するはずのコードを出力するように dotree 以下を直してみると。

```
dotree(x) int x; {
  printf("pid = fork();\n");
  printf("if(pid == 0) {\n");
  extree(x);
  printf("}\n");
  printf("pid = wait(0);\n");
  bufc = argc = tbufc = 0;
}

extree(x) int x; {
  int a[2], pid;
  if(tbuf[x].kind == T_COM) {
    int i = tbuf[x].left;
    if(argv[i] == 0) { printf("exit(0);\n"); return; }
    printf("execlp(\"%s\"", argv[i]);
    while(argv[i] != 0) printf(",\"%s\"", argv[i++]);
    printf(",0);\n");
    printf("perror(\"extree: exec fail.\"); exit(1);\n"); }
  else if(tbuf[x].kind == T_PIPE) {
    printf("pipe(a);\n");
    printf("if((pid = fork()) == 0) {\n");
    printf("dup2(a[1], 1); close(a[1]); close(a[0]);\n");
    extree(tbuf[x].left);
    printf("} else {\n");
```

```
        printf("dup2(a[0], 0); close(a[1]); close(a[0]);\n");
        extree(tbuf[x].right);
        printf("}\n"); }
      else if(tbuf[x].kind == T_SEQ) {
        printf("if((pid = fork()) == 0) {\n");
        extree(tbuf[x].left);
        printf("}\n");
        printf("pid = wait(0);\n");
        extree(tbuf[x].right); }
    }
```

えらく読みにくいが、よく見るとさっきのプログラムとほぼ対応しているのが分かるはず。さて、これを実行すると。

```
% yacc t28.yacc
% cc y.tab.c -ll
% a.out >t.c
date | tr a-z A-Z
( date ; who ) | wc
^D
% cat t.c
pid = fork();
if(pid == 0) {
pipe(a);
if((pid = fork()) == 0) {
dup2(a[1], 1); close(a[1]); close(a[0]);
execlp("date","date",0);
perror("extree: exec fail."); exit(1);
} else {
dup2(a[0], 0); close(a[1]); close(a[0]);
execlp("tr","tr","a-z","A-Z",0);
perror("extree: exec fail."); exit(1);
}
}
pid = wait(0);
pid = fork();
if(pid == 0) {
pipe(a);
if((pid = fork()) == 0) {
dup2(a[1], 1); close(a[1]); close(a[0]);
if((pid = fork()) == 0) {
execlp("date","date",0);
perror("extree: exec fail."); exit(1);
}
pid = wait(0);
execlp("who","who",0);
perror("extree: exec fail."); exit(1);
} else {
dup2(a[0], 0); close(a[1]); close(a[0]);
execlp("wc","wc",0);
perror("extree: exec fail."); exit(1);
}
}
pid = wait(0);
%
```

字下げがなくてえらく読みづらいが、一応プログラムらしきものになっている。これに、先頭`main() int pid, a[2];`、最後に`}`という行を加えると完全なプログラムになり、次のように立派に実行できる！

```
% cc t.c
```

```
% a.out
SAT JAN 26 22:09:52 JST 1991
       4      24     145
%
```

というわけで、シェル語らしきものを C に変換するコンパイラができたわけだ?! コンパイラとインタプリタは紙一重、というのが実感できますね、こういうのをやると。

A 演習、課題

A.1 20 節の演習

今回は最終回だし、甘い演習問題を多くしたげました。でもどっちにしろ、プログラムを書く (か例題を直す) ことは避けられませんが。

20-1. t21.c をどんな風にでもいいから改良してみよ。例えばシェルにあるような機能を何か付け加える、というのでもいいし、いっそシェルにもないような変わった (?) 機能を考案してつけてみるというのでもいい。

20-2. lex を使って、「ファイルの中にあるパターンが含まれている行を打ち出す (またはその数を数える) プログラム」を作れ。パターンは適当に選んで固定する。そして、それを grep などと競争させてみよ。

20-3. HP の電卓のまねっこを作れ。具体的には次のようなものを作る。

- 数が入力されたら、その数を内部スタックに積む。
- 演算子+,-,*,/が来たらスタックから 2 つ値を取り降ろし、それらに対して演算を施し結果をスタックに積む。
- 改行が来たらスタックの先頭の値を表示する。
- それ以外の文字は無視する。

なお、「数」は実数を使用する。できれば 1.15e-4 のような数値も入るようにすること (でないと lex を使う意味がない!)。[2]

20-4. t25.yacc の yacc の文法に、&, <, >を追加して動かしてみよ。lex の方も対応して修正するのを忘れないように。

20-5. t26.yacc などの%left を%right に変更したり|; の順位を入れ換えて、その影響を観察せよ。できれば、t27.yacc の自家製シェルについてもやってみて、どう変わったか、またそれはなぜかを考えよ。

20-6. t27.yacc の自家製シェルを改良して、新しい機能を何か追加してみよ。たとえば&が使えるようにする、というのでもいいが、できれば既存のシェルにないような機能を何か考えると面白い (たとえば「右から順に実行する;」とかね)。

20-7. t28.yacc のシェルコンパイラ (?) で、生成したプログラムと普通のシェルスクリプトとで同じ仕事をさせて、実行時間を測定し、どっちがどのくらい速いか示せ。

20-8. 何でもいいから、「C プログラムを生成するプログラム」を作成してみよ。できれば、引数あるいは入力に応じて生成されるプログラムが変化することが望ましい。[3]

---

[2] 実数形式の文字列を実数値に変換するには関数 atof(s) を使うとよい。宣言 double atof(); を忘れないように。
[3] 明らかにこれが一番のボーナス問題である。わかりますよね。

A.2　本日の課題

本日の課題は優しいことに、どれか 1 問でいいことにします。どうも長らくおつき合いいただいてありがとうさんでした。