S実験'95　SA (LexとYacc) ＃6

久野　靖 *

1995.5.29, 1995.6.28

6　「小さいシェル」例題プログラムの最終版

では、前々回にやったシェルの続きとして、次の機能を増やしてみよう。

- 「&」と「;」の機能を入れる。
- 「(」と「)」で囲めるようにする。
- シェルの while コマンドを入れる。

構文を増やすのはもはやどうにでもできますね? 一応、Yacc 記述全体を示しておく。

```
%token WORD;
%token WHILE;
%token DO;
%token DONE;
%%
prog    :
        | prog list '\n'  { docmd($2); prompt(); }
        ;
list    : cmd           { $$ = $1; }
        | list ';' cmd  { $$ = node(T_SEMI, $1, $3); }
        | list '&' cmd  { $$ = node(T_AMP, $1, $3); }
        ;
cmd     : pipe          { $$ = $1; }
        | WHILE list ';' DO ';' list ';' DONE
                        { $$ = node(T_WHILE, $2, $6); }
        ;
pipe    : redir   { $$ = $1; }
        | pipe '|' redir { $$ = node(T_PIPE, $1, $3); }
        ;
redir   : simple                  { $$ = $1; }
        | simple '>' word  { $$ = node(T_OREDIRECT, $1, $3); }
        | simple '<' word  { $$ = node(T_IREDIRECT, $1, $3); }
        ;
simple  : words         { $$ = $1; }
        | '(' list ')'  { $$ = $2; }
```

*筑波大学大学院経営システム科学専攻

```
        ;
words   :               { $$ = 0; }
        | word words    { $$ = node(T_WORDS, $1, $2); }
        ;
word    : WORD          { $$ = intern(yytext); }
        ;
```

これを見ると、「(」と「)」は構文だけ直せばいいので、excmd に手を入れる必要はない! 不思議でしょう? なぜかはよく考えてみよう。(こういうことを理解するかどうかでレポートの出来が決まりますね。) では Lex も示しておく。

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

なお、while の記法は/bin/sh 風にしてある。さて、残りは excmd に手を入れなければならない。その部分は次の通り。

```
excmd(i)
    int i; {
    int pid, fd1, p[2];
    union wait statusp;
    if(ntab[i].type == T_WHILE) {
      while(TRUE) {
        if(pid = fork()) {
          wait4(pid, &statusp, 0, 0);
          if(statusp.w_retcode) exit(0);
          docmd(ntab[i].right); }
        else
          excmd(ntab[i].left); } }
    else if(ntab[i].type == T_SEMI) {
      if(pid = fork()) {
        wait4(pid, 0, 0, 0); excmd(ntab[i].right); }
      else
        excmd(ntab[i].left); }
    else if(ntab[i].type == T_AMP) {
      if(pid = fork()) {
        excmd(ntab[i].right); }
      else
        excmd(ntab[i].left); }
    else if(ntab[i].type == T_PIPE) {
    (以下同じ)
```

まず、「;」であるが、2 つに分裂して、子プロセスで左側のリストを実行し、親はそれが完了するのを待ってから右側のコマンドを実行する。「&」はこの待つのを省略するだけである。(ところで今気が付きましたが、これは switch 文にした方が見易いですねえ。)

では、いよいよ最後の難関、while である。while の条件とは何かというと (知っています?)、任意のコマンド列を実行し、その最後のコマンドの実行終了状態が 0 なら成功、0 でなければ失敗を意味する。終了状態を取るには、このコードにあるように、wait4 の第 2 引数に終了状態を受け取るアドレスを渡せば取れる。で、while の動作そのものは

- 親子に分裂し、
- 子プロセスでは条件部の list を実行し、
- 親プロセスはその状態が 0 なら while の本体を実行する。

というのを繰り返し実行し、状態が 0 でなければ終りにすればよい。

では実行例をやろう。

```
% lex t19.lex
% yacc t19.yacc
% cc t19.c
% a.out
> (date;who) >t
> cat t
Mon May 22 17:06:01 JST 1995
kuno     ttyp0   May 22 15:09   (:0.0)
kuno     :0      May 22 15:09
> sleep 10 & ps
  PID TT STAT TIME COMMAND
 1579 p0 I    0:01 -csh (tcsh)
 1603 p0 I    0:00 rsh green00 -n sh -c
 1680 p0 S    0:00 a.out
 1684 p0 R    0:00 ps
 1685 p0 S    0:00 sleep 10
> while test -f t; do; date; sleep 5; done
Mon May 22 17:07:05 JST 1995
Mon May 22 17:07:10 JST 1995
Mon May 22 17:07:15 JST 1995  ←ここで別の窓から rm t すると
>                             ← while が終わる
```

7 「小さいコンパイラ」例題プログラムの最終版

7.1 複雑な式でも正しく計算するには…

さて、前回の「複雑な式が正しく計算できない」理由はわかりましたか? つまり、2 項演算子 (加算など) ではレジスタ 9 番に右辺中間結果を保存しておいて、その後左辺を計算するのだが、その中でまた 9 番を使ってしまうと「壊されて」しまうわけだ。

じゃあどうしたらいい? それには、例えば普通の変数と同様に作業変数 (テンポラリという) を割り当ててそこに中間結果を保存する、というのが定石だが、ここではもっと簡単でちょっと面白い方法を使う。これまで emittree は引数 1 個だったが、もう 1 個の引数を渡して、これを

- 式の計算などの場合は、結果を入れるレジスタ番号
- 条件の場合には飛び先ラベル番号

を入れておくものとする。そして、作業レジスタを使いたければこの番号より大きい番号のレジスタを使うものとする。だから例えば演算の左辺を 8 番に計算し、右辺を 9 番に計算するとすれば、右辺を計算する間は 9 番より大きい作業レジスタしか使わないから 8 番の中身は安泰なわけである。なかなか面白いでしょう?

という風に直した emittree を示しておく。

```
emittree(i, t)
    int i, t; {
    static int labelno = 32;
    int l;
    switch(ntab[i].type) {
case T_STLIST: if(ntab[i].left) emittree(ntab[i].left, 0);
               emittree(ntab[i].right, 0); break;
case T_READ:   printf(" la $4,$$0\n");
               printf(" la $5,%d($sp)\n", ntab[i].left*4+24);
               printf(" jal scanf\n"); break;
case T_PRINT:  emittree(ntab[i].left, 5);
               printf(" la $4,$$1\n");
               printf(" jal printf\n"); break;
case T_IF:     l = labelno++;
               emittree(ntab[i].left, l); emittree(ntab[i].right, 0);
               printf("$%d:\n", l); break;
case T_WHILE:  l = labelno; labelno += 2;
               printf("$%d:\n", l);
               emittree(ntab[i].left, l+1); emittree(ntab[i].right, 0);
               printf(" j $%d\n", l);
               printf("$%d:\n", l+1); break;
case T_ASSIGN: emittree(ntab[i].right, 8);
               printf(" sw $8,%d($sp)\n", ntab[i].left*4+24); break;
case T_LT:     emittree(ntab[i].left, 8); emittree(ntab[i].right, 9);
               printf(" bge $8,$9,$%d\n", t); break;
case T_GT:     emittree(ntab[i].left, 8); emittree(ntab[i].right, 9);
               printf(" ble $8,$9,$%d\n", t); break;
case T_ADD:    emittree(ntab[i].left, t); emittree(ntab[i].right, t+1);
               printf(" addu $%d,$%d,$%d\n", t, t, t+1); break;
case T_SUB:    emittree(ntab[i].left, t); emittree(ntab[i].right, t+1);
               printf(" subu $%d,$%d,$%d\n", t, t, t+1); break;
case T_MUL:    emittree(ntab[i].left, t); emittree(ntab[i].right, t+1);
               printf(" mul $%d,$%d,$%d\n", t, t, t+1); break;
case T_DIV:    emittree(ntab[i].left, t); emittree(ntab[i].right, t+1);
               printf(" div $%d,$%d,$%d\n", t, t, t+1); break;
case T_REM:    emittree(ntab[i].left, t); emittree(ntab[i].right, t+1);
               printf(" rem $%d,$%d,$%d\n", t, t, t+1); break;
case T_NUM:    printf(" li $%d,%d\n", t, ntab[i].left); break;
```

```
case T_VAR:     printf(" lw $%d,%d($sp)\n", t, ntab[i].left*4+24); break;
default:        printf("NotImplemented: %d\n", ntab[i].type); }
}
```

7.2 「手続き呼び出され」機能の付加

さて、せっかくコンパイラができたのだから、これを C から呼べるように直してみよう。要点は次の通り。

- プログラムの名前はこれまで無視していたが、アセンブリコードに反映させるように直す。そうすれば、C からその名前で呼べる。
- 引数を扱えるようにする。これは$4〜$7 に入っている値を「引数」として指定された変数にまず格納すればいい。
- 戻り値を扱えるようにする。今回は Pascal ふうに、「関数名と同じ変数に代入しておくとその値が返る」ようにする。

そのためには、まず文法を直す。

```
(この前まで同じ)
prog    : var '{' stlist '}'
                                { dotree($1, -1, -1, -1, -1, $3); }
        | var '(' var ')' '{' stlist '}'
                                { dotree($1, $3, -1, -1, -1, $6); }
        | var '(' var var ')' '{' stlist '}'
                                { dotree($1, $3, $4, -1, -1, $7); }
        | var '(' var var var ')' '{' stlist '}'
                                { dotree($1, $3, $4, $5, -1, $8); }
        | var '(' var var var var ')' '{' stlist '}'
                                { dotree($1, $3, $4, $5, $6, $9); }
        ;
(以下同じ)
```

つまり、「プログラム名」も変数にしてしまい、引数は最大 4 個にしたので、全部場合分けで書いてある。そして dotree にこれらの情報をすべて渡す (引数が少ない場合はその部分には負の数を渡すことにしてある)。で、dotree は次の通り。

```
dotree(n, p1, p2, p3, p4, i)
    int n, p1, p2, p3, p4, i; {
    int s = stabuse * 4;
    char *name = stab[n].name;    ←名前文字列を取り出す
    printf("      .text\n");
    printf("      .globl %s\n", name);   ←「main」の代わりに
    printf("      .ent   %s\n", name);   ←その名前を使う
    printf("%s: subu   $sp,%d\n", name, s+24);
    printf("      sw     $31,20($sp)\n");
    if(p1 >= 0) printf(" sw $4,%d($sp)\n", p1*4+24);  ←引数が
    if(p2 >= 0) printf(" sw $5,%d($sp)\n", p2*4+24);  ←あれば
    if(p3 >= 0) printf(" sw $6,%d($sp)\n", p3*4+24);  ←値を
```

```
    if(p4 >= 0) printf(" sw $7,%d($sp)\n", p4*4+24);  ←格納
    emittree(i);                          ← emittree は不変
    printf(" lw $2,%d($sp)\n", n*4+24);  ←関数名の変数値を$2 に
    printf("      lw     $31,20($sp)\n");
    printf("      addu   $sp,%d\n", s+24);
    printf("      j      $31\n");
    printf("      .end   %s\n", name);
    printf("      .sdata\n");
    printf("$$0:  .ascii \"%%d\\X00\"\n");
    printf("$$1:  .ascii \"%%d\\X0A\\X00\"\n");
}
```

これでちゃんと呼べるようになる。

```
% cat test4.uec
subr(x y) {
  subr = x * y;
}
% uecc <test4.uec >test4.s
% cat test4x.c
main() {
  printf("%d\n", subr(5, 6));
}
% cc test4x.c test4.s
test4x.c:
test4.s:
% a.out
30
%
```

8 SA のレポート課題

本日は出席課題は一切ありません。レポートに取り掛かってください。レポートはご想像通り

- シェルコース: 「小さいシェル」の機能を追加する
- コンパイラコース: 「小さいコンパイラ」の機能を増やす

の 2 コースがあります。どちらか好きなほうを選んでください。それぞれについて、さらに複数課題から複数選択して頂きます。

課題の提出期限は 8 月一杯 (厳守!) とします。夏休みに遊びたい人は夏休み前に済ませるように。9 月になってからの提出は受領しません。通常通り、久野のレポートボックスに提出してください。

なお、どちらのコースにも「独創的な機能を追加せよ」というのがありますが、**たまたま他人と同じ「独創的な機能」を考えて実現してしまった場合には点数をさしあげません。** (写したとかどうかということではなく、「独創的」の定義に反するからです。) 従って、この問題を選ぶ場合には他人が考え付かないような斬新なアイデアを出すようにしてください。よろしく。

シェルコース

今回作成した「小さなシェル」を改良して、次のような機能を入れてみよ。A〜C について、そのような機能を知らないなら自分で勉強すること。

課題 A シェル変数機能。

課題 B for 文ないし foreach 文。

課題 C コマンド置換 (‘…‘の機能)。

課題 D Ctrl-C を押すと実行中のプログラムを中止させる (厳密にはそのプロセスにシグナル 2 番を送る) 機能。(man 2 kill も参照。)

課題 E シェルの繰り返し文は、同じまたは類似したコマンドを多数繰り返し実行する機能を持つ。この類似版で、同じまたは類似したコマンドをずらっとパイプラインに並べるという機能を実現してみよ。構文も自分で適当に決めてよい。これができたら、フィルタを 100 個くらいつなげて実行させて、パイプライン 1 段あたりどれくらい通過時間が掛かるか測定を試みよ。

課題 F 普通のパイプラインでは、出力プロセスも入力プロセスも 1 つずつである。これを拡張して、$N > 1$ 個のプロセスが出力し、その結果を 1 個のプロセスが読むという「合流パイプライン」機能を実現してみよ。構文も自分で適当に決めてよい。

課題 G 上の方法では、まだ完全に自由なパイプライン配線を行うことはできなかった。そこで、パイプにすべて名前をつけ、「A のパイプはこのコマンドの入力、このコマンドの出力」というふうに名前つきで配線することにより、任意のパイプライン配線を行える機能をつくれ。これができたら、円周状にパイプラインを配線し、同じデータが無限に廻り続けるプロセス群を作ってみよ。

課題 H まだ誰も考え付いていないような、独創的なシェル機能を考案し、実現せよ。

これらのうちから、3 課題以上を選択し実現せよ。なお、自分の実力にあった課題を選ぶこと。実力に比してやさしいものばかり選んだ場合は採点が辛くなります。

コンパイラコース

今回作成した「小さなコンパイラ」を改良して、次のような機能を入れてみよ。

課題 a Pascal の repeat-until 文を入れて見よ。

課題 b C 言語の`&&`、`||`(条件のかつ、または) を入れてみよ。[1]

課題 c 関数呼び出し機能も実現せよ。当然、パラメタは渡せるようにすること。[2]

課題 d 配列機能を追加してみよ。配列の添字には少なくとも変数が書けること。

課題 e 今回の例題コンパイラのコードを見ると、余計な lw/sw があって遅そうである。これを工夫して改良せよ。どれくらい改良できたかを時間計測して評価せよ。

[1] できれば、`f1(...) && f2(...)` のような条件を書いた場合 f1 が 0 を返した時は f2 が呼ばれないようにしたい。

[2] 一番簡単なのは、「`x = func(...);`」のような関数呼び出し文を専用に設ける方法。しかしできれば「`f1(f2(...)+...)`」のように任意の式の中で任意の関数が呼べる方が望ましい。その場合何が問題になるか考えてみること。

課題 f さらに、C 言語と自分のコンパイラで同一内容のプログラムを作り、時間計測して競ってみよ。cc の「`-O3`」オプション指定で作ったコードと比べて勝つようなら素晴らしい。

課題 g switch 文のような機能を実現してみよ。構文は好きに決めてよい。

課題 h まだ誰も考え付いていないような、独創的な言語機能を考案し、実現せよ。

これらのうちから、3 課題以上を選択し実現せよ。なお、自分の実力にあった課題を選ぶこと。実力に比してやさしいものばかり選んだ場合は採点が辛くなります。