S実験'95　SA (LexとYacc) ＃5

久野　靖 *

1995.6.5, 1995.6.26

5　コンパイラの作り方

5.1　コンパイラはどれくらい大変か?

うすうす気がついているだろうが、lexやyaccはもともとは「コンパイラを楽に作るためのツール」として作られたものである。だから、作ってくれた人の恩に報いるために(?)、やはり最後はコンパイラを作ってみようではないか。(あーあ、やっぱり、と思いましたか?)

しかし皆様は既にアセンブラを手で書いたことはあるのだから、手で書く代わりに生成させる、というのは実はそんなに難しくはないのである。基本的に、やるべきことは次の通り。

- 木構造を作るところまでは前と同じ。
- 主手続きの入口と出口は「決まり文句」の通り出力。
- あとは、木をたどりながら各動作ごとに対応するアセンブラを出力。

今回はいちいちyaccやlexを直すのはめんどいので次のものを通して使う。まずlex。掛け算や割り算も入れてある。

```
alpha   [a-zA-Z]
digit   [0-9]
white   [\n\t ]
%%
while                           { return WHILE; }
if                              { return IF; }
read                            { return READ; }
print                           { return PRINT; }
{alpha}({alpha}|{digit})*       { return IDENT; }
{digit}+                        { return NUM; }
[-+()=;{}<>*/%]                 { return yytext[0]; }
{white}                         { ; }
```

次にyacc。これも特に目新しくはない。

```
%token NUM;
%token IDENT;
%token READ;
%token PRINT;
```

*筑波大学大学院経営システム科学専攻

```
%token WHILE;
%token IF;
%%
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 ';'   { $$ = node(T_PRINT, $2, 0); }
        | WHILE '(' cond ')' stat { $$ = node(T_WHILE, $3, $5); }
        | IF '(' cond ')' stat    { $$ = node(T_IF, $3, $5); }
        | '{' stlist '}'          { $$ = $2; }
        ;
cond    : expr '<' expr { $$ = node(T_LT, $1, $3); }
        | expr '>' expr { $$ = node(T_GT, $1, $3); }
        ;
expr    : prim          { $$ = $1; }
        | expr '+' prim { $$ = node(T_ADD, $1, $3); }
        | expr '-' prim { $$ = node(T_SUB, $1, $3); }
        | expr '*' prim { $$ = node(T_MUL, $1, $3); }
        | expr '/' prim { $$ = node(T_DIV, $1, $3); }
        | expr '%' prim { $$ = node(T_REM, $1, $3); }
        ;
prim    : NUM           { $$ = node(T_NUM, atoi(yytext), 0); }
        | var           { $$ = node(T_VAR, $1, 0); }
        | '(' expr ')'  { $$ = $2; }
        ;
var     : IDENT         { $$ = lookup(yytext); }
        ;
```

ではついでに、C の方もわかっているところは掲載してしまう。

```
#define TRUE    1
#define FALSE   0
#define yywrap() 1
extern char yytext[];
struct stab {
    int val;
    char name[20]; } stab[100];
int stabuse = 0;
struct node {
    int type, left, right, work; } 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_MUL    7
#define T_DIV    8
#define T_REM    9
#define T_NUM   10
#define T_VAR   11
#define T_WHILE 12
#define T_IF    13
#define T_LT    14
#define T_GT    15
#include "y.tab.c"
#include "lex.yy.c"

main() {
    yyparse();
}

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

lookup(s)
    char *s; {
    int i;
    for(i = 1; i < stabuse; ++i)
        if(strcmp(stab[i].name, s) == 0) return i;
    strcpy(stab[stabuse].name, s); return stabuse++;
}

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;
}
```

さていよいよ dotree であるが、基本的に「おきまりの」出入口を出力して、その間で emittree を呼んで「中身の」コードを出力させる。なお、スタックをどれくらい使用するかは、stabuse 個の変数があるのだからその個数の 4 倍に 24 を足せばいいわけである。

```
dotree(i)
    int i; {
    int s = stabuse * 4;
```

```
    printf("      .text\n");
    printf("      .globl main\n");
    printf("      .ent   main\n");
    printf("main: subu   $sp,%d\n", s+24);
    printf("      sw     $31,20($sp)\n");
    emittree(i);
    printf("      lw     $31,20($sp)\n");
    printf("      addu   $sp,%d\n", s+24);
    printf("      j      $31\n");
    printf("      .end   main\n");
    printf("      .sdata\n");
    printf("$$0:  .ascii \"%%d\\X00\"\n");
    printf("$$1:  .ascii \"%%d\\X0A\\X00\"\n");
}
```

ところでラベル$$0 と$$1 のところには scanf と printf で使う文字列"%d"と"%d\n"がそれぞれ入れてある。さて、emittree は次の通り。

```
emittree(i)
    int i; {
    switch(ntab[i].type) {   ←※
case T_STLIST: if(ntab[i].left) emittree(ntab[i].left);
               emittree(ntab[i].right); break;
case T_READ:   printf(" la $4,$$0\n");
               printf(" la $5,%d($sp)\n", ntab[i].left*4+24);
               printf(" jal scanf\n"); break;
case T_PRINT:  emittree(ntab[i].left);
               printf(" move $5,$8\n");
               printf(" la $4,$$1\n");
               printf(" jal printf\n"); break;
case T_NUM:    printf(" li $8,%d\n", ntab[i].left); break;
case T_VAR:    printf(" lw $8,%d($sp)\n", ntab[i].left*4+24); break;
default:       printf("NotImplemented: %d\n", ntab[i].type); }
}
```

一度に全部やると大変だから、まずはここに挙げたノードのみを処理して、それ以外のは「作ってないよー」と言うことにした。作った部分の説明は次の通り。

- T_STLIST — 左側に文の並びがあればまずそのコードを出し、続いて右側にある文のコードを出す。それだけ。
- T_READ — 文字列"%d"のアドレスをレジスタ 4 番、読み込もうとする変数のアドレスをレジスタ 5 番に入れて scanf を呼び出せばよい。変数は 24(sp)、28(sp)、…にあることに注意。
- T_PRINT — まず、emittree で式のコードを出力すると、式の結果をレジスタ 8 番に入れるコードが出るものと (勝手に) 決める。従ってやるべきことは、式のコードを emittree で出し、レジスタ 8 番から 5 番に値を転送すれば 5 番に式の値が入る。そして、レジスタ 4 番に文字列"%d\n"のアドレスを入れて printf を呼び出す。
- T_NUM — 上で決めたことにより、レジスタ 8 番にその値を入れればよい。それだけ。

- T_VAR — こちらは、レジスタ 8 番に変数の値を持ってくればよい。

さて、コンパイラだと名前が a.out のままではコンパイルしたプログラムの a.out と同じで不都合なので、言語とコンパイラに名前をつけよう。言語は UEC(Unbelieavably Elementary C)、コンパイラは uecc(UEC Compiler) にしておく。(皆様のはまた別の言語だからそれぞれ勝手な名前をつけるように。)

```
% lex t20.c
% yacc t20.yacc
% gcc -o uecc t20.c  ← a.out ではなく uecc に
% cat test1.uec
main {
  print 8;
  read x;
  print x;
}
% cat <test1.uec >test1.s
```

では、アセンブリコードを見てみよう。

```
% cat test1.s
      .text
      .globl main
      .ent   main
main: subu   $sp,32
      sw     $31,20($sp)
 li $8,8
 move $5,$8
 la $4,$$1
 jal printf
 la $4,$$0
 la $5,28($sp)
 jal scanf
 lw $8,28($sp)
 move $5,$8
 la $4,$$1
 jal printf
      lw     $31,20($sp)
      addu   $sp,32
      j      $31
      .end   main
      .sdata
$$0:  .ascii "%d\X00"
$$1:  .ascii "%d\X0A\X00"
```

これだけでも結構長い。では実行してみよう。

```
% cc test1.s  ←これでアセンブラから翻訳してくれる
% a.out
```

```
8
11
11
%
```

練習 1 打ち込んで動かせ。

5.2 代入文と式

では代入と演算を入れよう。emitcode の中の switch の枝だけ示す。

```
case T_ASSIGN: emittree(ntab[i].right);
               printf(" sw $8,%d($sp)\n", ntab[i].left*4+24); break;
case T_ADD:    emittree(ntab[i].left);
               printf(" move $9,$8\n");
               emittree(ntab[i].right);
               printf(" addu $8,$9,$8\n"); break;
case T_MUL:    emittree(ntab[i].left);
               printf(" move $9,$8\n");
               emittree(ntab[i].right);
               printf(" mul $8,$9,$8\n"); break;
```

つまり、代入はレジスタ 8 番に右辺の値を計算させておいて、それを変数の場所にストアすればいい。加算はまず左辺を計算し、その値をレジスタ 9 番に取っておいて、次に右辺を 8 番に計算し、最後に加算命令で足せばいい。掛け算も同様。[1]

```
% cc -o uecc t21.c
% cat test2.uec
main {
  read x;
  read y;
  x = x + 1;
  print x + y;
  print x * y;
}
% uecc <test2.uec >test2.s
% cc test2.s
% a.out
5
3
9
18
%
```

練習 2 上のように直して動かせ。

[1]何かここでいやーな気持ちになった人はいませんか。あなたはするどい。

練習 3 引き算、割り算、剰余演算も入れて見よ。なお、アセンブラ命令が分からない時は C で引き算などを含む簡単なプログラムを作り、「`cc -S` ファイル」によりアセンブラ出力を出してそれを観察すればちゃんとわかる。

練習 4 この方式だと、次のようなプログラムは値が正しく計算できない。

```
main {
  read x;
  print (x + 1) * (x + 1);
}
```

どのような場合に正しく計算できないのか? なぜそうなのか? 探求せよ。

練習 5* 正しく計算できるようにするにはどうしたらいい? アイデアを出して、直してみよ。

5.3 制御の流れ

さて、いよいよ「文」に行って見よう。その場合、飛び先ラベルを$32、$33、…のように生成しないといけないので、その数を数えたりする変数を emittree の※印の前に追加する。

```
static int labelno = 32;
int l;
switch(....) {  ←※
```

では、「条件」をどうするか考えてみよう。IF や WHILE に使うことを考えると、「条件」というのはその条件が成り立たなかった時に「別のラベルへ飛び出す」という動作をするべきであろう。そこで:

```
case T_LT:      emittree(ntab[i].left);
                printf(" move $9,$8\n");
                emittree(ntab[i].right);
                printf(" bge $9,$8,"); break;
case T_GT:      emittree(ntab[i].left);
                printf(" move $9,$8\n");
                emittree(ntab[i].right);
                printf(" ble $9,$8,"); break;
```

つまり、bge 命令や ble 命令を生成するのだが、飛び先ラベルはこれを呼び出す側が出せるように残しておく。では if 文は:

```
case T_IF:      l = labelno++;
                emittree(ntab[i].left);
                printf("$%d\n", l);
                emittree(ntab[i].right);
                printf("$%d:\n", l); break;
```

つまり、まずラベル番号を用意し、条件を生成し、成り立たなかった時の飛び先を埋める。ついで中身の文を生成し、その後でラベルを生成する。では試そう:

```
% cc -o uecc t22.c
% cat test4.uec
main {
  read x;
  if(x > 0) print 1;
  if(x < 0) print 0;
}
% uecc <test4.uec >test4.s
% cc test4.s
% a.out
3
1
% a.out
-4
0
%
```

練習 6 上のように直して動かせ。

練習 7 while 文を使えるようにしてみよ。(とっても簡単!)

練習 8 簡単だがループ回数の多いプログラムを作り、時間計測をしてみよ。C でも同等のプログラムを動かし、どっちが速いか比べよ。(時間計測は「`time a.out`」を使うのでしたね?) `cc` に`-O3` オプションをした場合はどうか? C のアセンブリ言語コードも見て、優劣の原因を分析せよ。

練習 9* この版では、条件の両側が複雑な場合もうまく動かない。どういう場合に動かないか検討し、動くように直してみよ。

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。「*」がついたものは高度ですからはしょっても構いません。だいたい時間になったら、特に興味深かった問題 1 つ程度の報告を簡潔にまとめて、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. あなたはコンパイラの動作について、この演習以前にどれくらい知っていましたか。また、この演習によってどれくらいまでコンパイラについて理解したと自分で思いますか。

  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。