[第６回目２００２・１０・８]

構文解析の実際：yacc の使い方

前回は、yacc の基本的な使い方について解説した。さて、今回は yacc を使って tiny C のインタプリターを作ることにする。yacc のクローンである bison のマニュアルは、

http://www.omoikane.co.jp/i/info/html/bison-1.28/bison-ja_toc.html

に解説してあるために、参考にするとよい。

yacc の動きは、以下のように動作する。

1. yylex を呼び出して、token を読み込み、その token から始まる文法規則を探す。
2. その文法規則が終るまで、token を読み、遷移(shift)を続ける。
3. 文法に非終端記号がある場合は、その文法規則をスタックに積み、1 からやり直す。
4. 文法規則の最後まで遷移したら、その規則を還元(reduce)する。
5. スタックから一つ前に処理していた規則に戻り、3.で還元した非終端記号をつかって、さらに shift する。
6. 2 に戻る。

実際に yacc が出力する parser のコードを解読するのは無理であるが、参考として、-v を付けて yacc を起動することによって、y.output というファイルができるので、これを見るとどのように shift、reduce しているかを見ることができる。

yacc の action と意味値(semantic value)

前回つくった式の yacc のプログラムでは、単に構文が定義した文法にあっているかをチェックするものであったが、構文解析の仕事は定義した構文にあっているかとチェックするとともに、構文を表現する構文木(抽象構文木：abstract syntax tree, AST)を作ることである。構文木は意味解析でその意味に従った処理が行われる。今回は、Lisp のインタプリターをベースに使うために、構文木は Lisp のデータ構造を使う。

yacc では、構文解析の途中で、何らかの動作を行う action の指定ができる。構文木を作る作業はこの action の中で行う。action は構文規則の中に{ } で囲んで，C 言語で記述する。例えば、

```
term: factor { printf("factor is coming"); }
        | term '*' factor { printf("factor is added"); }
        ;
```

この例では、term の各規則が reduce されたときに、{}の中の action が実行される。通常、action は各構文規則の最後に書き、reduce された時に実行されるようにするが、途中に書いてもよい。その場合には、そこまで、shift されたときに action が実行されるようになる。

構文規則の主な仕事は、構文木を作ることである。yacc では各構文規則で生成される値を意味値(semantic value)を持つことができ、その構文で認識された構文木を意味値として、action でその意味値を生成（計算）する。例えば、上の例では

```
term: factor { $$ = $1; }
        | term '*' factor { $$ = addSymbol(mulSym,makeList2($1,$2)); }
        ;
```

$1,$2,...は、右に現れる非終端記号の意味値であり、これを使って、$$は右の記号 term の意味値を計算する。この値のデータ型は構文木すなわち Lisp のオブジェクト型にする。なお、makeList2 は２つのオブジェクトのリストを作る関数で、addSymbol は先頭に symbol を付け加える関数である。

宣言部に、以下の記号のデータ型を定義する。

```
%union {
  Object *val;
}
%type <val> term factor
```

%union は、意味値に使うデータ型を定義するもので、この中のデータ型は複数でもよい。この union のメンバーを使って、%type で構文規則の記号の意味値を定義する。さて、factor の定義では、終端記号の意味値を使う。

```
%type <val> NUM SYM
...
factor: NUM  | SYM ;
```

{ $$ = $1; }の場合は省略してもよい。
終端記号に対しては、字句解析ルーチン yylex からは、yylex の値を NUM,SYM を返すとともに、意味値を yylval という変数（これは yacc から生成されるルーチンの中で定義されている）を介して、意味値を返す。

```
int yylex(){
   .... /* NUM の時  */
   yylval.val = makeNum(n);
   return NUM;
  .... /* SYM の時*/
  yylval.val = makeSymbol(yytext);
  return SYM;
  .... }
```

なお、意味値のデータ型は、yacc の中では YYSTYPE という名前になっており、

```
#define YYSTYPE ...
```

として、直接定義する方法もある。
右のプログラムが tiny C の字句解析部である。

```
/* lex.c */
char yytext[100];

int getChar(){
      int  c; c = getc(stdin); return c;
}

void ungetChar(int c)
{
       ungetc(c,stdin);
}

int yylex()
{
      int  c,n;
      char  *p;
again:
      c = getChar();
      if(isspace(c)) goto again;
       switch(c){
      case '+':case '-':case '*':case '>':
     case '<':case '(':case ')':case '{':
     case '}':case ']':case '[':case ';':
      case ',':case '=':
      case EOF:
                return c;
      case '"':
            p = yytext;
            while((c = getChar()) != '"'){
                  *p++ = c;
            }
            *p = '¥0';
          yylval.val = makeNum((int)strdup(yytext));
           return STRING;
       }
       if(isdigit(c)){
              n = 0;
              do {
                    n = n*10 + c - '0';
                    c = getChar();
                } while(isdigit(c));
                ungetChar(c);
                yylval.val = makeNum(n);
                return NUMBER;
       }
       if(isalpha(c)){
             p = yytext;
             do {
                   *p++ = c;
                   c = getChar();
             } while(isalpha(c));
             *p = '¥0';
            ungetChar(c);
            if(strcmp(yytext,"var") == 0)
                       return VAR;
            else if(strcmp(yytext,"if") == 0)
                       return IF;
            else if(strcmp(yytext,"else") == 0)
                       return ELSE;
            else if(strcmp(yytext,"return") == 0)
                       return RETURN;
          else if(strcmp(yytext,"while") == 0)
                       return WHILE;
            else if(strcmp(yytext,"for") == 0)
                       return FOR;
            else {
             yylval.val = makeSymbol(yytext);
               return SYMBOL;
               }
       }
      fprintf(stderr,"bad  char  '%c¥n",c);
       exit(1);
}

void yyerror()
{
      printf("syntax  error!¥n");
       exit(1);
}
```

優先度の定義

yacc は LALR parser であり、一文字先読みをしているため、演算子の結合規則と、優先度を定義できる。%left は左結合規則を持つ演算子であることを指定する。例えば

```
%left '+'
```

と指定すると、

```
expr: expr '+' expr { ...} ;
```

の文法規則を使って、x + y + z に対して((x + y) + z)のように処理される。%right は右結合規則を持つもので、例えば代入の'='は右結合規則を持つものである。%left、%right は同時に演算子の優先度も指定する。後から、指定したほうが高い優先度を持つものと解釈される。これを使うと優先度をもつような規則を簡単に書くことができる。

```
%left '+' '-'
%left '*'
%left UMINUS
...
expr:   factor
      |  expr '+' expr { $$=addSymbol(plusSym,makeList2($1,$3)); }
      | exp '-' exp { $$=addSymbol(minusSym,makeList2($1,$3)); }
      | exp '*' exp { $$=addSymbol(mulSym,makeList2($1,$3)); }
      | '-' exp %prec UMINUS { .... }
      ;
```

なお、最後の%prec は単項演算子を最も優先度の高い処理をするための指定である。

あいまいな文法と shift/reduce conflict，reduce/reduce conflict

文法にあいまいさがあると、LR 構文解析ができなくなるので、yacc は警告メッセージをだす。メッセージには２種類あり、shift/reduce conflict，reduce/reduce conflict がある。shift/reduce conflict とは、文法規則が shift（つまり、さらに長い非終端記号に reduce できる）なのか、reduce(そこで打ち切って、非終端記号にしてしまう)か、解釈ができることを示す。この conflict は一概に文法定義が間違っているということではない場合がある。有名な例として、IF 文の定義がある。

```
statement : IF '(' expression ')' statement
          | IF '(' expression ')' statement ELSE statement
```

```
            ....;
```
これは次の場合にあいまいになる。
```
  if (a > 0)
   if (b > 0) c = 100;
    else
   c = 2000;
```
else を読んだとき、この token は内側の if 文の一部であると考え、遷移すればよいのだろうか、それとも、内側の if 文は完了したと考え、還元して、読みこんだ else は外側の if 文の一部であるとして遷移すればよいのだろうか？一般に yacc は、shift/reduce conflict がおきたときには、例外条件として、遷移(shift)を優先させる。したがって上の else は内側の if 文の一部と解釈される。この解釈は、C 言語を始めほとんどの言語の仕様と一致するので、一般に if 文にまつわる shift/reduce conflict はそのままにしておいて問題ない。

他方、reduce/reduce conflict は、同時に還元できる文法規則が複数あることを意味する。便宜上、yacc でははじめに現れた文法規則を優先させるが、これは望ましいことではないので、この conflict がないように文法を作る必要がある。例えば、良くある例として 0 個以上の word 列を読む場合を考えてみる。
```
sequence: /*  */  { printf ("empty sequence\n"); }
        | maybeword
        | sequence word  { printf ("added word %s\n", $2); }
        ;
maybeword: /*  */  { printf ("empty maybeword\n"); }
        | word  { printf ("single word %s\n", $1); }
        ;
```
この場合は、word は mayword に reduce でき,sequence でも reduce できてしまう。この場合は単に、以下のように定義してやればよい。
```
sequence: /*  */  { printf ("empty sequence\n"); }
        | sequence word  { printf ("added word %s\n", $2); }
        ;
```
もう一つの注意点として、再帰的な定義がある。例えば、','で区切られた列を表現する場合に次の２つの方法がある。
```
seq:  item |  seq ',' term ;  /* left recursion */
seq:  item | term ',' seq ;   /* right recursion */
```
yacc では、right recursion では、途中の状態をスタックにとっておく必要があるため、なるべく、left recursion で書いておくべきである。

エラー回復処理

通常使っているコンパイラでは、途中で文法エラーを見つけたとしてもなるべく、他の部分も parse して一度に多くの文法エラーを見つけることができるようにしてある。文法エラーを見つけたときに、次にどこから構文解析を再開するかの処理をエラーからの回復処理という。どこから処理を再開するか、どうやって再開するかについてはコンパイラの使いやすさの要素の一つにもなり、結構むずかしい問題である。ここでは、yacc での簡単なエラー処理だけについて述べておく。

yacc では、予約の非終端記号として、error という予約語があり、yyerror が呼び出されて、これが終了(retrun)すると、error という記号に reduce されるように処理してある。例えば、
```
    statement: ....
          | error ';'
          ;
```
とすることによって、statement の構文解析で文法エラーが起きた場合には、';'がくるまで読みとばす処理をすることになる。

tiny C のインタプリター

www 上に、tinyC のインタプリターの全体プログラムがおいてある。cparser.y が yacc のプログラムである。clex では、token と意味値を返す字句解析 yylex が定義してある。cparser.y では、tiny C で書かれたプログラムを構文解析して、同等の Lisp プログラムのデータ構造を作り、external_definition

のところで、eval0bject を呼び出して、関数の定義、配列の定義をする。main では、yyparse を呼び出して、プログラム全体を読み、定義したあとで、main プログラムを呼び出している。

```
/* tiny C parser */
%token NUMBER
%token SYMBOL
%token STRING
%token VAR
%token IF
%token ELSE
%token RETURN
%token WHILE
%token FOR

%{
#include <stdio.h>
#include "lisp.h"

%}

%union {
      Object *val;
}

%right '='
%left '<' '>'
%left '+' '-'
%left '*'

%type <val> parameter_list block local_vars symbol_list
%type <val> statements statement expr primary_expr arg_list
%type <val> SYMBOL NUMBER STRING

%start program

%%

program: /* empty */
                  | external_definitions
                  ;
external_definitions:
                      external_definition
                  | external_definitions external_definition
                  ;

external_definition:
                     SYMBOL parameter_list block  /* fucntion definition */
                  { evalObject(addSymbol(defSym,makeList3($1,$2,$3))); }
                  | VAR SYMBOL ';'
                  | VAR SYMBOL '=' expr ';'
            { evalObject(addSymbol(eqSym,makeList2($2,$4))); }
                  | VAR SYMBOL '[' expr ']' ';'
                  { evalObject(addSymbol(arraySym,makeList2($2,$4))); }
                  ;
parameter_list:
                   '(' ')'
                   { $$ = NULL; }
                  | '(' symbol_list ')'
                   { $$ = $2; }
                  ;
parameter_list:
                   '(' ')'
                   { $$ = NULL; }
                  | '(' symbol_list ')'
                   { $$ = $2; }
                  ;
block: '{' local_vars statements '}'
                  { $$ = addSymbol(blockSym,addList($2,$3)); }
                  ;
local_vars:
                    /* NULL */ { $$ = NULL; }
                  | VAR symbol_list ';'
                    { $$ = $2; }
                  ;
symbol_list:
                      SYMBOL
                   { $$ = makeList1($1); }
                  | symbol_list ',' SYMBOL
                   { $$ = addLast($1,$3); }
                  ;
```

```
statements:
                      statement
                   { $$ = makeList1($1); }
                  | statements statement
                   { $$ = addLast($1,$2); }
                  ;
statement:
                    expr ';'
                   { $$ = $1; }
                  | block
                   { $$ = $1; }
                  | IF '(' expr ')' statement
                   { $$ = addSymbol(ifSym,makeList3($3,$5,NULL)); }
            | IF '(' expr ')' statement ELSE statement
                   { $$ = addSymbol(ifSym,makeList3($3,$5,$7)); }
                  | RETURN expr ';'
                   { $$ = addSymbol(returnSym,makeList1($2)); }
                   /*
                   * kadai 6, WHILE and FOR
                    */
                  ;
expr:                 SYMBOL '=' expr
                   { $$ = addSymbol(eqSym,mak eList2($1,$3)); }
                  | SYMBOL '[' expr ']' '=' expr
                   { $$ = addSymbol(setArraySym,makeList3($1,$3,$6)); }
                  | expr '+' expr
                   { $$ = addSymbol(plusSym,makeList2($1,$3)); }
                  | expr '-' expr
                   { $$ = addSymbol(minusSym,makeList2($1,$3)); }
                  | expr '*' expr
                    { $$ = addSymbol(mulSym,makeList2($1,$3)); }
                  | expr '<' expr
                   { $$ = addSymbol(lessSym,makeList2($1,$3)); }
                  | expr '>' expr
                   { $$ = addSymbol(greaterSym,makeList2($1,$3)); }
                  | primary_expr
                  ;
primary_expr:
                     SYMBOL
                  | NUMBER
                  | STRING
                  | SYMBOL '[' ex pr ']'
                    { $$ = addSymbol(getArraySym,makeList2($1,$3)); }
                  | SYMBOL '(' arg_list ')'
                   { $$ = addList($1,$3); }
                  ;
arg_list:
                     expr
                   { $$ = makeList1($1); }
                  | arg_list ',' expr
                   { $$ = addLast($1,$3); }
                  ;

%%
#include "clex.c"
```

```
main()
{
       initEval();
       yyparse();

      /* execute main */
       printf("executing main ...¥n");
        callFunc(lookupSymbol("main"),NULL);
       printf("main end ..¥n");
}
```