情報処理演習II
コンパイラの作成
第3回 yaccを用いた構文解析器の作成 (2)

2005年度担当: 中井央

概 要

前回に引き続き、Yaccを使った処理を演習する。

0 宿題の解答

前回の宿題の解答と若干の解説を以下に述べる。なお、これらの解答例のソースプログラムを ~nakai/IPP2 以下に設置した。

1

まず、線形リストのおさらいとして通常のC言語で、標準入力から受け取った文字列を線形リストにつなぐ問題である。解答例を図1に示す。

6〜9行目では線形リストの1つのセルのための構造体を宣言している。`main` の外側で宣言しているのは、実際的なプログラミングでは他の関数でもこの構造体を参照するからである。

次に `main` の中の説明をしよう。ここでは先頭にダミーのセルを1つ持った線形リストとした。もちろん、最初は先頭などを指すポインタが `NULL` 値を持ち、挿入するたびに `NULL` かどうかチェックするオーソドックスなやりかたもある。しかし、最初の1個さえ追加されれば、あとはそのようなチェックは不要であることから、メモリ効率よりも実行効率を考慮して、ダミーのセルを置く方式を採用している。

15〜23行目はこのリストの初期化である。ダミーのセル用のメモリを確保し、リストの先頭および末尾を指すポインタによって指させている。

25行目から48行目までは標準入力から文字列を受け取りリストへの追加を行うという作業の繰り返しである。

26〜30行目では標準入力からの文字列の受付を処理している。

32行目〜36行目ではその文字列用のセル用のメモリを確保している。

38行目〜43行目では同様にしてその文字列自体を入れるためのメモリの確保を行い、その文字列をコピーしている。

45〜47行目でリストの末尾にその要素をつないでいる。

50〜52行目は表示のための `for` 文である。

2 (a)

この問題の解答例を図2に示す。

これは前回のテキストの図2をもとにして作成したものである。このため途中部分は省いてある。これは図1をYacc記述に当てはめただけのものである。詳細は省略する。

2 (b)

こちらは図2の識別子を認識する部分のアクションの変更のみである。変更された部分を図3に示す。

ここではいま得られた識別子を挿入すべき位置を探し、そこに挿入している。得られた識別子は `yytext` に入っている。一時変数 `tmp2` は最初リストの先頭を指している。ここではリストの先頭はダミーであり、どんな文字列よりも前に並ぶはずであるヌルストリング (`""`) が入っている。そして、`tmp2->next->word`、つまり、`tmp2` が指しているセルの次のセルの文字列と `yytext` を比較し、`tmp2->next->word` の方が辞書順で後ろであれば、`tmp2` の次で、`tmp2->next` の前、つまり、両者の間が `yytext` の収まる位置である。50

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define MAXSIZE 1000
 5
 6 struct LIST {
 7   struct LIST * next;
 8   char *word;
 9 };
10
11 main(){
12   char buf[MAXSIZE];
13   struct LIST *h, *t, *tmp;
14
15   /* for making dummy leading node */
16   h = (struct LIST*)
         malloc(sizeof(struct LIST));
17   if (h == NULL){
18     perror("memory allocation error");
19     exit(EXIT_FAILURE);
20   }
21
22   t = h;
23   h->next = NULL;
24
25   while(1){
26     printf("input a word: ");
27     fgets(buf, MAXSIZE, stdin);
```

```
28     if (strcmp(buf, "owari\n") == 0){
29       break;
30     }
31
32     tmp = (struct LIST*)
           malloc(sizeof(struct LIST));
33     if (tmp == NULL){
34       perror("memory allocation error");
35       exit(EXIT_FAILURE);
36     }
37
38     tmp->word = (char*)malloc(strlen(buf)+1);
39     if (tmp == NULL){
40       perror("memory allocation error");
41       exit(EXIT_FAILURE);
42     }
43     strcpy(tmp->word, buf);
44
45     t->next = tmp;
46     tmp->next = NULL;
47     t = tmp;
48   }
49
50   for(tmp = h->next;
       tmp != NULL;
       tmp = tmp->next){
51     printf("%s", tmp->word);
52   }
53 }
```

図 1: 第 2 回テキストの宿題 1 の解答例

〜56 行目ではこの処理をしている。もし、適切な位置が見つからなかった場合、tmp2 は最後の要素を指している (for の条件による)。この場合はリストの一番最後にその要素が追加される。

どちらにしても for ループを終了した時点で、tmp2 の「次」に yytext のためのセルを挿入すればよい。残りはメモリ確保と挿入であるので詳細は省略する。

2 (c)

こちらも同様に識別子を認識する部分のアクション部分のみを図 4 に示す。こちらは図 3 を少し変えたものである。挿入位置が決まった際、tmp2 の word と yytext が同じであれば、挿入操作をしない。

3

解答例を図 17 に示す。ただし、紙面の都合上、一部省略している。1 つの識別子は 2 箇所以上に現れる可能性がある。このため、1 つの識別子用のセルは 2 箇所以上の行番号を保持するためのリストの先頭へのポインタを持たせるとする。同様にそのようなリストへの追加の時間を削減するためにその末尾へのポインタも持たせるとする。これを行っているのが 15, 16 行目である。

説明が前後するが、その行番号のリストのセルは 7〜9 行目で宣言している。これまでの問題と同様に識別子にマッチした際のアクション部分にその操作を記述する。

68〜75 行目はその識別子の行番号を入れるためのセルの処理である。

その識別子がこれまでに出現していないものであった場合、その識別子用のセルの確保、その接続をし、そのセルに先ほど行った処理で得られた行番号のセルをつなぐ (77〜96 行目)。

その識別子が既に出現していた場合は、その識別子のセルの行番号リストの末尾に先ほど行った処理で得られた行番号のセルをつなぐ (97〜100 行目)。

また、main では、それらをきれいに表示するため、123〜129 行目のように処理を行っている。この詳細は

```
 1 %{
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 #define MAXSIZE 1000
 6
 7 struct LIST {
 8   struct LIST * next;
 9   char *word;
10 };
11
12 int lineno=1;
13 struct LIST *h, *t, *tmp;
14 %}
15
16 %%
...
50 [_A-Za-z][_A-Za-z0-9]*  {
51          tmp = (struct LIST*)
                  malloc(sizeof(struct LIST));
52          if (tmp == NULL){
53            perror("memory allocation error");
54            exit(EXIT_FAILURE);
55          }
56
57          tmp->word = (char*)
                        malloc(strlen(yytext)+1);
58          if (tmp == NULL){
59            perror("memory allocation error");
60            exit(EXIT_FAILURE);
61          }
62          strcpy(tmp->word, yytext);
```

```
63
64          t->next = tmp;
65          tmp->next = NULL;
66          t = tmp;
67        }
68 .               { /* do nothing */ }
69 "\n"            { lineno++; }
70
71 %%
72 main(){
73
74   /* for making dummy leading node */
75   h = (struct LIST*)
         malloc(sizeof(struct LIST));
76   if (h == NULL){
77     perror("memory allocation error");
78     exit(EXIT_FAILURE);
79   }
80
81   t = h;
82   h->next = NULL;
83
84   while(yylex()!=0){
85   }
86
87   for(tmp = h->next;
         tmp != NULL;
         tmp = tmp->next){
88     printf("%s\n", tmp->word);
89   }
90 }
```

図 2: 前回の宿題 2(a) の解答例 (一部省略)

省略する。

4

この問題の Yacc 記述の例を図 5 に、Lex 記述の例を図 6 に示す。括弧で括った式はどの演算子よりも優先順位が高いので、もう 1 つ非終端記号を導入する。ここでは 18、19 行目の生成規則を追加した。このパターンは覚えてしまうのがよい。

1 あいまいな文法に対する処理

前回、あいまいな文法について少し触れた。あいまいな文法とはある入力に対して 2 つ以上の構文解析木が作成できるようなものをいう。この典型的なものとしてはプログラミング言語の `if` 文がある。

ここでは Pascal-S という言語を取り上げてみる。C 言語などでもほぼ同様である。Pascal-S の場合、条件式が成り立った場合は `TEHN` 部分のステートメントが実行され、そうでない場合は `ELSE` 部分が実行されるかもしくは `ELSE` 以降の記述がなければ何も実行されない。

`if` 文の部分を文法で書き表してみると次のような感じ になる。

```
stmt : if_stmt
       | ....
       ;

if_stmt :  IF expr THEN stmt else_stmt
       ;

else_stmt :
         ELSE stmt
       | /* empty */
       ;
```

`expr` の部分には条件式が来る。この部分の文法は別に定められている。`stmt` はステートメントを表し、文法としては通常の 1 つの文か `BEGIN` と `END` (C 言語

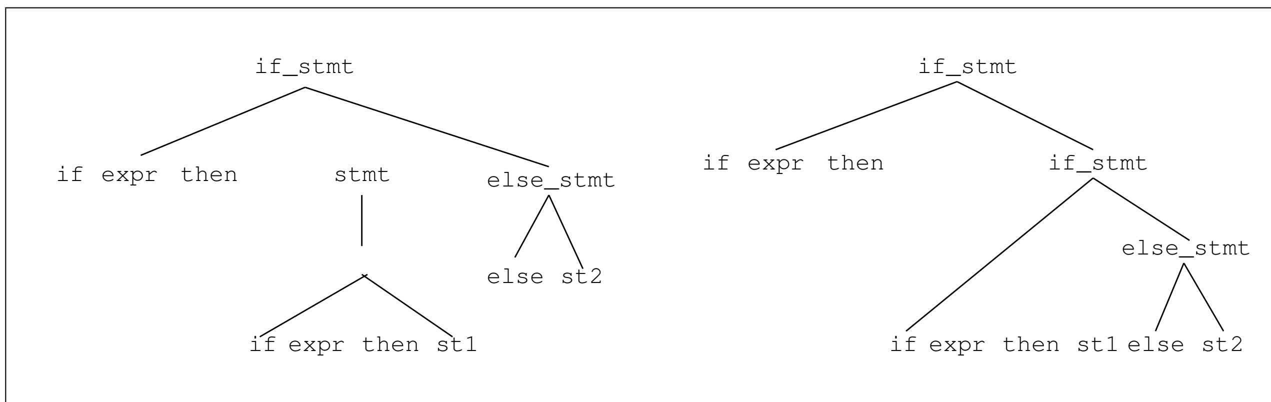

図 7: if 文を表すあいまいな文法による解析木の例

なら{ と }) で囲んだ複数の文を表している。ELSE 部分は上で述べたようにあるかもしれないしないかもしれない。ELSE 部分は ELSE が来て文が来るか何もないかである。

問題は次のような場合である。

```
if expr THEN if expr THEN st1 ELSE st2
```

この文では果たして ELSE はどちらの if に対応するのだろうか。我々の常識(??) では 2 番目の if である。

しかし、if_stmt は stmt であるから、上記の if_stmt を左辺とする規則によると、if expr THEN st1 の部分を stmt とみなし、ELSE は最初の if と対になるとも考えられる。

すなわち、この if 文の文法規則はあいまいなのである。1 つの解決方法として文法をあいまいではないように書き直すことが考えられる。しかし、その文法は複雑でわかりにくい。

解析器の側から見るとあいまいな文法が与えられた場合、上の ELSE のようにどちらの if に対応してよいか判断できなくなる。つまり、ELSE を見たときに、先に if expr THEN st1 を stmt にまとめてしまうべきなのか、それとも ELSE とその先を読んで if expr THEN st1 ELSE st2 を stmt にまとめてしまうべきなのかの判断に困る (図 7)。

Yacc ではこのような場合、文法があいまいであることに対し、Shift Reduce Conflict (シフト還元衝突) というエラーメッセージを出力する。

詳細はコンパイラの教科書に譲るが、ここでシフトとは、上記の例でいうと ELSE を先に読む (図 7 右) ことを言う。一方、還元とは 2 番目の if から st1 までを先にまとめてしまうこと (図 7 左) を言う。つまり、上記で述べた通り、先に ELSE 以降を処理するのか ELSE より前をまとめるのかどっちの処理をすればいいかわからない。つまり、「処理が衝突している」ということである。Yacc による解析器ではこのような場合には処理を行わないのではなく、シフト、つまり ELSE を先に読むという処理を行うことにしている。このことはこの節の冒頭で述べたように我々の常識にも合致する。

曖昧さ除去規則

一般にプログラミング言語の文法を書く際、文法をあいまいに記述するほうが文法は簡単になる。そこで上の if 文の場合だと、「ELSE が見つかったらもっとも近い IF と対応させる」というような文法とは別のルールを設けて、解析を行う方法も考えられる。このような別に用意した、あいまいさを取り除くためのルールのことを「**あいまいさ除去規則**」という。

前回示した算術式の文法においても文法をあいまいにしておいて、結合のしかたを示すあいまいさ除去規則を導入することもできる。算術式などの演算子の場合は演算子の優先順位、結合性などの指定であいまいさをなくしている。通常、我々は 1+2*3 を見たら先に 2*3 を演算するというルールに則って計算をする。

Yacc ではこの演算子の優先順位や結合性を指定することであいまいさを除去する仕組み (記述方法) も備わっているが、このテキストではそれは扱わない。各自で調べられたい。

```
50 {  for(tmp2=h;
51        tmp2->next != NULL;
52        tmp2 = tmp2->next){
53      if (strcmp(tmp2->next->word, yytext)>0){
54        break;
55      }
56    }
57
58    tmp = (struct LIST*)
            malloc(sizeof(struct LIST
59    if (tmp == NULL){
60      perror("memory allocation error");
61      exit(EXIT_FAILURE);
62    }
63
64    tmp->word = (char*)malloc(strlen(yytext)+1);
65    if (tmp == NULL){
66      perror("memory allocation error");
67      exit(EXIT_FAILURE);
68    }
69    strcpy(tmp->word, yytext);
70
71    tmp->next = tmp2->next;
72    tmp2->next = tmp;
73 }
```

図 3: 前回の宿題 2(b) の解答例 (識別子のアクションのみ)

2 算術式の拡張

これまで、算術式を扱う文法を用いてきた。もう少し難しい計算式を扱いたい場合でも、どうやって新しい演算子を導入すればよいかもわかってきたであろう。

ここではこのような電卓もどきから少し拡張して、代入文を扱えるようにしたい。式として、`a = 1+2` のように入力すると `a` には 3 が保持されていて、次に何かの式で `2*a` などと出てきたら、その式を評価すると結果として 6 という値を得たい。

ここでは順を追ってそのような文法と処理を考えていこう。

2.1 文法規則を考える

まずは変数を導入した場合の式の文法を考えよう。どういうパターンがあるだろうか。

1. `a = ...`
2. `2*3-a ...`

```
50 {
51   for(tmp2=h;
52       tmp2->next != NULL;
53       tmp2 = tmp2->next){
54     if (strcmp(tmp2->next->word, yytext)>0){
55        break;
56     }
57   }
58
59   if (strcmp(yytext, tmp2->word)!=0){
60     tmp = (struct LIST*)
             malloc(sizeof(struct LIST));
61     if (tmp == NULL){
62       perror("memory allocation error");
63       exit(EXIT_FAILURE);
64     }
65
66     tmp->word = (char*)
                   malloc(strlen(yytext)+1);
67     if (tmp == NULL){
68       perror("memory allocation error");
69       exit(EXIT_FAILURE);
70     }
71     strcpy(tmp->word, yytext);
72
73     tmp->next = tmp2->next;
74     tmp2->next = tmp;
75   }
76 }
```

図 4: 前回の宿題 2(c) の解答例 (識別子のアクションのみ)

文法として記述するにはもう少し詳しく考える必要がある。今までは 2 の形式を 1 つの行とし、それを何回も繰り返し入力できるとしていた。今回は 1 と 2 の両方を入力として受け付けたい。

ここで図 5 を見てみよう。これは前回の宿題の 4 の解答例である。ここではこれを拡張することにしよう。

できるだけ、この文法を崩さないようにするには、新しい非終端記号を導入する必要がある。

図 8 に解答例を示すが、それを見る前にできるだけ各自で考えてみてほしい。

図 8 のための Lex 記述を図 9 に示す。

図 8 について少し説明しておく。まず、1 行を構成する式は、`a = ...` という代入文の形式と単純な算術式の形式がある。1 行にはこのどちらかが含まれるので、その部分が選択になるようにする必要がある。そのため、5 行目では新たな非終端記号 `Es` を導入した。

これまでは `E` だけだったが、`E` とは別に代入文形式を扱う規則も必要となる。そこで、9 行目で `Es` を左辺

```
 1  %token NUM
 2  %token NL
 3  %%
 4  LIST    : LIST E NL { printf("%d\n", $2); }
 5          | /* empty */
 6          ;
 7
 8  E       : E '+' T { $$ = $1 + $3; }
 9          | E '-' T { $$ = $1 - $3; }
10          | T { $$ = $1; }
11          ;
12
13  T       : T '*' F { $$ = $1 * $3; }
14          | T '/' F { $$ = $1 / $3; }
15          | F { $$ = $1; }
16          ;
17
18  F       : NUM { $$ = $1; }
19          | '(' E ')' { $$ = $2; }
20          ;
21
22  %%
23
24  #include "lex.yy.c"
25
26  main(){
27    yyparse();
28  }
```

図 5: 前回の宿題 4 の解答例 (Yacc 記述 hw2_4.y)

とする規則を導入する。1 つはこれまでと同様の E を導出するものであり、もう 1 つは代入文を許す規則である。10 行目では代入文を表す生成規則の左辺として新たな非終端記号 A [1] を導入した。13 行目はその代入文に対する生成規則である。

なお、この文法に対する Lex 記述では、識別子は英小文字 1 文字として、それにマッチしたら IDENT を返す規則を入れた (3 行目)。

演習 1

図 8 と図 9 を入力し、Yacc と Lex に与えて、コンパイルし、実行せよ。ここではまだアクションをきちんと書いていないため、(計算) 結果の表示は正しいものが得られない。しかし、a=2*3 のような代入文や a*b のような式を受理することを確認できる。

[1]「代入」には、英語では asignment という単語を使う。ここではこの頭文字をとった。

```
 1 %%
 2 [0-9]+   { yylval = atoi(yytext); return NUM; }
 3 "+"      { return '+'; }
 4 "*"      { return '*'; }
 5 "-"      { return '-'; }
 6 "/"      { return '/'; }
 7 "("      { return '('; }
 8 ")"      { return ')'; }
 9 [ \t]    { /* do nothing */ }
10 "\n"     { return NL; }
11 .        { return yytext[0]; }
```

図 6: 前回の宿題 4 の解答例 (Lex 記述 hw2_4.l)

2.2 アクションを考える

それぞれの入力を受け取ったらどんな処理をするべきかを次に考えていかなければならない。

以下を読む前にどんな処理があるか考えてほしい。整数定数からなる算術式を計算する部分は以前と同じである。ここでは識別子が出てくる場合の処理を考える必要がある。

識別子が出てくるのは図 8 では 13 行目と 27 行目である。なお、ここでは簡単のため、英小文字 26 文字の 1 字を変数としている。つまり、a=3 や b=6 などとして 26 個まで変数を使えるわけである。

まずは最初なので簡単にこれらを処理する方法を考えよう。実際にどんな長さでもよい識別子 (C などと同等のもの) を考慮するにはもっと複雑なデータ構造を考える必要がある (26 個ではなく無限に変数名を作り出せるから)。

26 個の変数のみを扱う例

では 26 個の変数のみを扱う例をここでは考えよう。後ほど、どんな識別子が来てもよいように拡張する。

ここでは解答例をまず示そう (図 10)。

なお、この Yacc 記述用の Lex 記述は図 9 の 3 行目を次のように変更したものである。

```
3  [a-z]   { yylval = yytext[0]; return IDENT; }
```

ここでは 26 個限定で変数が使える。つまりアルファベット小文字のどれか 1 文字を変数に使えるのである。それが代入文の左辺の変数として使われた場合、代入

```
 1 %{
 2 int ident[26];
 3 %}
 4
 5 %token NUM
 6 %token IDENT
 7 %token NL
 8 %%
 9 LIST : LIST Es NL { printf("%d\n", $2); }
10      | /* empty */
11      ;
12
13 Es   : E { $$ = $1 }
14      | A { /* 右辺が 1 個だけなら
                $$=$1 は省略可能 */ }
15      ;
16
17 A    : IDENT '=' E {
18                       $$ = ident[$1 - 'a'] = $3;
19                    }
20      ;
21
22 E    : E '+' T { $$ = $1 + $3; }
23      | E '-' T { $$ = $1 - $3; }
24      | T { $$ = $1; }
25      ;
```

```
26
27 T    : T '*' F { $$ = $1 * $3; }
28      | T '/' F { $$ = $1 / $3; }
29      | F { $$ = $1; }
30      ;
31
32 F    : NUM { $$ = $1; }
33      | IDENT { $$ = ident[$1 - 'a']; }
34      | '(' E ')' { $$ = $2; }
35      ;
36
37 %%
38
39 #include "lex.yy.c"
40
41 main(){
42   int i = 0;
43
44   /* the initialization of ident */
45   for(i=0; i<26; i++){
46     ident[i] = 0;
47   }
48
59   yyparse();
50 }
```

図 10: 26 個の変数を扱える算術式の文法 (Yacc 記述 ex7.y)

文の右辺の式を評価した結果をそこにしまっておく必要がある。

ここでは簡単な例として、26 個の変数用に int 型の要素を 26 個持つ配列を用意した (図 10 の 2 行目)。

通常、計算機上では文字コードには ASCII が使われている。C 言語では'a' とやると実際には a を表す ASCII でのコード番号が得られる。ident[0] を a に ident[1] を b に、…、ident[25] を z に割り当てたい場合、変数として a が入力されたら'a'-'a' とすることで 0 が得られる。同様に z が入力されたら'z'-'a' とすることで 25 が得られる。

この原理を使うと入力された文字のコードを保持しておけば ident での位置 (添え字) がわかる。

このため、Lex 記述では yylval にその 1 文字の ASCII としての値を収めている。

次に再び Yacc 記述に目を向けるとまず 33 行目では変数が「使用」されている。つまり式中のどこかに現れる形になっている。このとき、この識別子の値は配列 ident のどこかの位置にある。この位置は先ほど述べた計算の原理によって求められる。したがって、33 行目のようなアクションになる。$1 には字句解析器で yylval に代入されたその文字のコードが入っている。したがって、$1 - 'a' は ident のその変数に対応する位置を計算している。そしてそこから取り出した値をその生成規則にマッチしたときの値として$$に代入している。

今度は 17 行目に目を向ける。E で計算した値は$3 に得られている。この値を代入文の左辺に示した変数に対応する格納場所、すなわち、ident のある位置にしまう必要がある。

なお、計算結果を画面に表示するためにはこの代入文も「値」を持つ必要がある。このため、この生成規則の左辺の記号 E に値を持たせるため、$$への代入も行っている。

演習 2

ex7.y と ex7.l を入力し、コンパイル、実行せよ。a=3 や b=a*2 などと入力して、代入文が正しく実行されていることを確認せよ。また、代入が行われた変数のみを入力として、その変数の値が確認できることを確かめよ。

```
 1 %token NUM
 2 %token IDENT
 3 %token NL
 4 %%
 5 LIST    : LIST Es NL { printf("%d\n", $2); }
 6         | /* empty */
 7         ;
 8
 9 Es      : E
10         | A
11         ;
12
13 A       : IDENT '=' E { /* some actions */ }
14         ;
15
16 E       : E '+' T { $$ = $1 + $3; }
17         | E '-' T { $$ = $1 - $3; }
18         | T { $$ = $1; }
19         ;
20
21 T       : T '*' F { $$ = $1 * $3; }
22         | T '/' F { $$ = $1 / $3; }
23         | F { $$ = $1; }
24         ;
25
26 F       : NUM { $$ = $1; }
27         | IDENT { /* some actions */ }
28         | '(' E ')' { $$ = $2; }
29         ;
30
31 %%
32
33 #include "lex.yy.c"
34
35 main(){
36   yyparse();
37 }
```

図 8: 識別子を導入した文法 (ex6.y)

通常の識別子を許す拡張

変数名は自由につけたいものである。英小文字 26 個などとけちなことは言わず、C 言語と同じように変数名をつけたい。

では、その処理を考えよう。

まず、Lex 記述で識別子を認識するための正規表現を C 言語でのものに変更する必要がある。

```
[_a-zA-Z][_a-zA-Z0-9]*        { /* action */ }
```

識別子を字句解析器が認識したとき、何を行うべきか、構文解析器では何をするかを考える必要がある。

ここでは字句解析器ではその文字列を「確保」し、構文解析器ではそれを使って、それ用のメモリの割り当て、それによる値の参照などをするとしよう。

```
 1 %%
 2 [0-9]+  { yylval = atoi(yytext); return NUM; }
 3 [a-z]   { return IDENT; }
 4 "+"     { return '+'; }
 5 "*"     { return '*'; }
 6 "-"     { return '-'; }
 7 "/"     { return '/'; }
 8 [ \t]   { /* do nothing */ }
 9 "\n"    { return NL; }
10 .       { return yytext[0]; }
```

図 9: 図 8 のための Lex 記述 (ex6.l)

つまり、次のような感じになる。

1. 字句解析器で識別子であると認識した
2. 代入文の左辺なら、
   (a) 初出なら右辺の値をしまう「場所」を作り、そして右辺の値をしまう
   (b) 既出なら右辺の値をしまう「場所」を探し、そして右辺の値をしまう
3. 式 (代入文の右辺) 中なら、値がしまってある「場所」を特定し、その値を取り出す

字句解析器から構文解析器へ値を渡す

26 個の変数のみを扱う場合は、26 個の要素をもつ配列を用意しておけばよかった。また、配列の添え字が `int` 型であることも幸い (??) し、字句解析器からは `yylval` で値を返却すればよかった。

今回はどんな文字列が入力されるかわからないし、何個変数が利用されるかもわからない。このような状況に対応するには**動的な**データ構造を用いる。

ここでは検索効率のことはあまり考えずに線形リストを使うことにしよう。

上記の 2(a) では新しいセルを用意し、2(b) ではリスト中にその名前を探しに行く。

ここで 1 つ問題がある。字句解析器で得られた文字列をどのようにして Yacc が生成する構文解析器に渡すかである。これまで `yylval` は `int` 型であった。`yylval` が任意の型を使用できれば、この問題はうまく解決できそうである。

Yacc 記述で `yylval` の型を変更するには `YYSTYPE` というマクロを使う。たとえば、`int` 型ではなく `double` 型にしたいならば、次のように Yacc 記述の冒頭部分

に書けばよい。

```
%{
#define YYSTYPE double

...
%}
```

さて、ここではさらに問題がある。あるときには字句解析器から値として int 型を構文解析器に渡したいが、別の時には char*型を渡したい、という場合もある。いま我々が直面している問題で言えば、整数を認識したときにはその値を構文解析器側に渡したいが、識別子が見つかった場合にはその字面を構文解析器に渡したい。

これを実現するには C 言語の共用体を用いる。共用体は構造体に書き方が似ている。ただし、本来はメモリを節約するための仕組みであるため、たとえば、int が 4 バイト、double が 8 バイトである共用体は全体では 8 バイトしか領域を持たない。

```
struct ST {
  int    val;
  double dval;
};

union UNI {
  int    val;
  double dval;
};
```

このような宣言があったとしよう。struct ST はすでに知っているようにメンバとして宣言された分だけメモリ領域が確保される。union UNI の方は、val と dval には同じメモリ領域を割り振る。使用されるときの状況によって、そのメモリ領域が int として扱われたり double として扱われたりする。

共用体は同時には参照しないような変数を 1 つの領域にまとめて割り当てる際に有効である[2]。なお、共用体のメンバの参照方法は構造体のものと同様である。

字句解析器からは 1 度には 1 つの値しか返さない。このため、字句解析器が構文解析器とやり取りをする変数 yylval を共用体型にし、複数の型を扱えるようにすればよい。

Yacc にはこのための記法が用意されている。

```
%union {
  char *name;
  int   val;
}
```

このように書いておくと、出力される C プログラムには次のように記述される。

```
typedef union  {
  char *name;
  int   val;
} YYSTYPE;
extern YYSTYPE yylval;
```

ではこの機能がわかったところで、再び、lex での記述を考えよう。まず、整数にマッチした場合を考慮しよう。

いままでは yylval に整数値を代入していた。ここでは yylval は共用体であり、そのメンバ val に値を設定する必要がある。つまり、yylval.val = ... とする必要がある[3]。

今度は識別子にマッチした場合である。構文解析器に渡したいのはマッチしたパターンの文字列である。マッチした段階ではこの文字列は yytext に入っている。しかし、yytext は字句解析器内の一時的に使用される変数であるため、次の解析のために文字を読み始めれば以前の内容は失われてしまう。このため、マッチした時点でその文字列を別の場所に保存しておく必要がある。このためにはその文字列用に動的にメモリを確保し、yytext の中身をそこへコピーする必要がある。そしてその先頭番地を構文解析器へ渡すため、yylval.name = ... のようにする。

共用体を用いたことで字句解析器側から構文解析器側へ型の異なる値を渡すことができるようになった。

では代入文の左辺か式中かで構文解析器での処理を考えよう。

---

[2]現在はメモリをふんだんに使えるからこの機能をあまりありがたいと思わないかもしれない。

[3]ある C コンパイラでは atoi の引数として yytext を直接書くとエラーになるため、ここでは、atoi((char*)yytext) のように記述している。

代入文の左辺の場合

右辺の値を左辺の識別子に対応する「場所」に格納し、同時にその値をその生成規則の値とする。左辺の識別子に対応する「場所」が既に登録されているかどうか探す必要があり、登録されていれば、そこへ値を入れ、登録されていなければ新たに「場所」を作ってそこへ値を入れる必要がある。

式中の場合

式中で識別子が現れたら、その識別子が持つ値を利用することを意味する。このため、その識別子の値がしまわれている「場所」を探し、値を取り出す必要がある。もし、その「場所」がまだなければ、何らかの処理をしなければならない。考えられる処理は、登録されていないからという理由で、エラーメッセージを表示するものである。あるいは、エラーメッセージを出す代わりに適当に定めたデフォルト値を与えるということも考えられる。ここでは後者を採用するとし、デフォルト値は 0 とする。

コーディング

以上で実際に識別子を扱える「計算機」を作ることができる。

図 10 を参考にして ex8.y として Yacc 記述を、図 9 をもとにして ex8.l として Lex 記述を作成しよう。

コーディング例を図 11、図 12 に示す。

演習 3

コーディング例を見てもよいが、図 9 と図 10 を参考にして、自分で作成せよ。

3 ファイル入出力

これまでは標準入力からデータを受け取っていた。ここではコマンドライン引数からファイル名を指定して解析を行うようにする方法を述べる。

典型的な C プログラム断片は図 13 のようになる。1 行目は常套手段であり、この通りに覚えるべきものである。2 行目は Lex が持っている yyin というファイルポインタ変数である。これには最初に標準入力が割り当てられている。4 行目はコマンドライン引数があるかどうかの確認である。ない場合は標準入力からデータを受け取るとする。コマンドライン引数がある場合、その引数をファイル名とみなしてファイルを開く。ここではあらかじめ stdin として開かれているものを、コマンドライン引数として与えられたファイル名で開きなおしをするため、freopen を使っている。詳細は man コマンドで確認されたい。

演習 4

ex8.y の main 関数をコマンドライン引数を扱えるようにし、適当に入力ファイルを作成し、コマンドラインから与えて実行してみよ。

```
 1 main(int argc, char* argv[]){
 2   extern FILE *yyin;
 3
 4   if (argc>1){
 5     if((yyin = freopen(argv[argc-1],"r",stdin))
         == NULL){
 6       fprintf(stderr,
               "Can not open file %s\n",
               argv[argc-1]);
 7       exit(1);
 8     }
 9   }
10   ...
```

図 13: コマンドライン引数でファイル名を指定する方法

4 分割コンパイル

以下は飛ばして次回テキストから始めても良い。

通常、C 言語でのプログラム開発では分割コンパイルが使われる。プログラムを設計するときは、まず、問題をいくつかの部分に分け、それぞれの部分ごとにファイルを作り開発する。できるだけその部分は独立して試験ができるようになっていることが望ましい。

Lex や Yacc を利用して開発を行う場合も、それぞれの解析器ごとにソースファイルが作成される。Lex は lex.yy.c を出力し、Yacc は y.tab.c を出力する。この他、必要に応じて関数などを定義するファイルが

```
 1 %{
 2 #include <stdlib.h>
 3
 4 char *tname;
 5 %}
 6
 7 %%
 8 [0-9]+                  { yylval.val = atoi((char*)yytext);
 9                           return NUM; }
10 [_a-zA-Z][_a-zA-Z0-9]*  { tname = (char*)malloc(strlen(yytext)+1); }
11                           if (tname==NULL){
12                             perror("memory allocation");
13                             exit(EXIT_FAILURE);
14                           }
15                           strcpy(tname, yytext);
16                           yylval.name = tname; return IDENT; }
17 "+"                     { return '+'; }
18 "*"                     { return '*'; }
19 "-"                     { return '-'; }
20 "/"                     { return '/'; }
21 [ \t]                   { /* do nothing */ }
22 "\n"                    { return NL; }
23 .                       { return yytext[0]; }
```

図 11: 図 12 用の Lex 記述

存在する。

最初のうちは、別々にすることが面倒なため、`#include` 文などを使って、1 ファイルに見せかけておいてコンパイルを行っていた。図 12 などでは Yacc の 2 つめの`%%`以降には `main` を含めて 3 つの関数定義がある。これらを別途ファイルを開いてそこに収めることもできる。ここではそのファイル名を `main.c` とする。`#include` を利用していた場合には不要であったが、別々にコンパイルする場合には、終端記号名などを何らかの形で取り込む必要がある。

すなわち、今までは終端記号に対して、`%token` で宣言したものにはマクロにより数値が割り振られていた。独立にコンパイルする際にはそのような情報がないと困る。この情報は Yacc を実行する際に`-d` オプションをつければよい。そうすると必要な情報を `y.tab.h` というファイルに出力してくれる。これによる作業手順は次のようになる。

1. 必要ファイルの編集。
   `hoge.l` には次のような 1 行を入れておく必要がある。

   ```
   %{
   #include "y.tab.h"
   ...
   %}
   ```

2. `yacc -d hoge.y` の実行
3. `lex hoge.l` の実行
4. `cc -c y.tab.c` (`y.tab.o` ができる)
5. `cc -c lex.yy.c` (`lex.yy.o` ができる)
6. `cc main.c y.tab.o lex.yy.o -ly -ll`

しかし、これらを毎回やるのはばかばかしいし、変更がなければ再コンパイルは不要である。それをタイムスタンプを見ながら行ってくれるのが `make` である。

`make` に指示を与えるためのファイル名は `Makefile` である。これにはファイル間の依存関係を記述する。基本的にはあるファイルを作りたい場合、それを作るのに必要なファイルを`:`(コロン) に続けて並べる。実際にそれを作るために必要なコマンドなどはその次の行に**先頭に必ずタブを入れてから**、記述する。ここでは図 14 に Yacc と Lex を使う場合の `Makefile` の例を示す。なお、最後の `clean` はコマンドプロンプトから `make clean` と打つことで途中で作成されたファイルの削除を行う。

図 14 のように書けたら、コマンドプロンプトから `make` と打つことで、`hoge.l`、`hoge.y` から `a.out` を自

```
 1 %union {
 2   char *name;
 3   int   val;
 4 }
 5
 6 %{
 7 #include <stdio.h>
 8 #include <stdlib.h>
 9
10 struct VALS {
11   int          val;
12   char        *name;
13   struct VALS *next;
14 };
15
16 struct VALS *h, *t;
17
18 int getval(char*);
19 int setval(char*, int);
20 %}
21
22 %token <val>  NUM
23 %token <name> IDENT
24 %token NL
25
26 %type <val> Es
27 %type <val>  A
28 %type <val>  E
29 %type <val>  T
30 %type <val>  F
31
32 %%
33 LIST    : LIST Es NL { printf("%d\n", $2); }
34         | /* empty */
35         ;
36
37 Es      : E { $$ = $1; }
38         | A { $$ = $1; }
39         ;
40
41 A       : IDENT '=' E {
42                         $$ = setval($1, $3);
43                       }
44         ;
45
46 E       : E '+' T { $$ = $1 + $3; }
47         | E '-' T { $$ = $1 - $3; }
48         | T { $$ = $1; }
49         ;
50
51 T       : T '*' F { $$ = $1 * $3; }
52         | T '/' F { $$ = $1 / $3; }
53         | F { $$ = $1; }
54         ;
55
56 F       : NUM { $$ = $1; }
57         | IDENT { $$ = getval($1); }
58         | '(' E ')' { $$ = $2; }
59         ;
60 %%
61
62 #include "lex.yy.c"
```

```
63
64 main(){
65   int i = 0;
66   struct VALS *tmp;
67
68   h = t = (struct VALS*)
           malloc(sizeof(struct VALS));
69   if (h==NULL){
70     perror("memory allocation");
71     exit(EXIT_FAILURE);
72   }
73
74   h->next = NULL;
75
76   yyparse();
77 }
78
79 int getval(char* name){
80   struct VALS *tmp;
81
82   for(tmp=h->next;
       tmp != NULL;
       tmp = tmp->next){
83     if (strcmp(tmp->name, name)==0){
84       return tmp->val;
85     }
86   }
87
88   return 0;
89 }
90
91 int setval(char* name, int val){
92   struct VALS *tmp;
93
94   for(tmp=h->next;
       tmp != NULL;
       tmp = tmp->next){
95     if (strcmp(tmp->name, name)==0){
96       break;
97     }
98   }
99
00   if(tmp == NULL){
01     tmp = (struct VALS*)
           malloc(sizeof(struct VALS));
02     if (tmp == NULL){
03       perror("memory allocation");
04       exit(EXIT_FAILURE);
05     }
06     tmp->name = name;
07     tmp->val  = val;
08
09     t->next = tmp;
10     tmp->next = NULL;
11     t = tmp;
12   }
13   else {
14     tmp->val = val;
15   }
16
17   return val;
18 }
```

図 12: 識別子を使用可能である計算機の Yacc 記述例 (ex8.y)

```
CC      = cc

a.out   : lex.yy.o y.tab.o main.o
        $(CC) lex.yy.o y.tab.o main.o -ly -ll

lex.yy.o        : lex.yy.c
        $(CC) -c lex.yy.c

y.tab.o : y.tab.c
        $(CC) -c y.tab.c

lex.yy.c        : hoge.l
        lex hoge.l

y.tab.c : hoge.y
        yacc hoge.y

main.o  : main.c
        $(CC) -c main.c

clean   :
        rm -rf *~ *.o y.tab.c lex.yy.c
```

図 14: Makefile の例

動で作ってくれる。

演習 5

ex8.y と ex8.l を別々にコンパイルし、make を使うようにしよう。手順は以下のようになる。

1. 図 14 をもとにして hoge を ex8 に変えて、Makefile を作成する。
2. ex8.l の 2 行目の次の行に#include "y.tab.h" を入れる。
3. ex8.y の 2 つめの%%のあとの #include "lex.yy.c"を削除し、それ以降の部分を別のファイル main.c に移す (ex8.y からは削除する)。
4. ex8.y の先頭の%{と%}で囲まれた中身を main.c の先頭部分に写す (ex8.y にも残す)。
5. make clean を実行後、make とやってみる。lex や yacc が順に起動され、a.out が作成される。ここで ex8.l を適当に編集してみる。内容が変わらなくてもよいが、編集し保存をすることで変更時刻を更新する[4]。その後、make を実行してみよう。

[4] touch コマンドを使うと変更時刻だけ更新される。

5 木 (tree)

コンパイラの内部では解析を行う際、木を作成し、その上で様々な操作を行う場合が多い。

ここでは「木」について少し触れておく。

5.1 木の概念と C による 1 実装方法

まず、計算機で扱う木とはどういうものかを簡単に説明し、具体的な実装方法、取り扱う方法を述べる。

5.1.1 構文解析木と抽象構文木

これまでに構文解析木が何回か出てきている。構文解析をする際、各文法記号に対応するノードを作成して木を構成していったものが、構文解析木となる。ところで、これまでに見た算術式に関する構文解析木は冗長なところがある。演算子の優先順位をつけるため、非終端記号を導入したわけであるが、たとえば、足し算の式を解析しても、必ず掛け算に関する生成規則を通るため、途中にたくさんのノードが付随することがある。

構文解析木と違って、たとえば算術式では、演算子と被演算子の関係だけを示すような木もある。これを抽象構文木という。構文解析木と抽象構文木の違いを図 15 に示す。

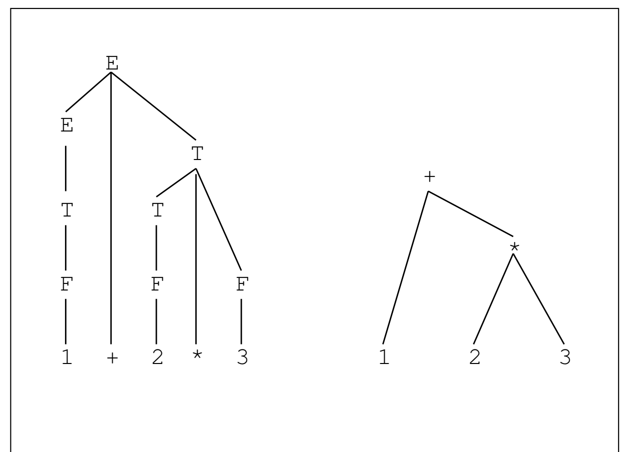

図 15: 1+2*3 に対する構文解析木（左）と抽象構文木（右）

5.1.2　木の訪問

計算機のプログラムで木を使う場合、根から順に各ノードを訪問し、それぞれのノードでなんらかのアクションを起こす。このときの訪問にはいくつかのやり方がある。大きく分けると深さ優先と幅優先のやり方である。幅優先では、自分の子ノードのすべてを訪問してから、それぞれの子ノードのさらに子ノードへ訪問する。深さ優先では、たとえばまず左の子を訪問したら、そのノードからはさらにそのノードの子ノードを探索し、葉まで到達したら、その親に戻り、次の子ノードをまた葉に到達するまで訪問していく。

処理を行うタイミングにもいくつかのやり方がある。深さ優先でいうと、あるノードに到達したときと、そこからさらに深く訪問し、それらが終わって戻ってきたときがある。そのノードに到達し、さらに深く訪問する前になんらかのアクションを起こす順序のことを「前順」という。逆にすべての子ノードの訪問が終わって戻ってきた段階でアクションを起こす順序のことは「後順」という。2 分木の場合だと、左の子を訪問し、右の子を訪問する前にアクションを起こすような順序のことを「中順」という。

なお、木を順に訪問することを「走査する (traverse)」ともいう。

5.1.3　C 言語による一実装方法

ここでは簡単のため、2 項演算のみを扱うとする。実際には必要に応じて拡張すればよい。このため、ここでの抽象構文木とはその演算子のノードに対し、被演算子を子ノードとする木のことである。被演算子は整数か部分式 (subexpression) であることに注意する必要がある。

では、このような木を C 言語で実装する方法を検討しよう。まず、ノードは次のような構造体をで表す。

```
struct NODE {
  int opcode; /* the number of operator */
  int val;
  struct NODE *left_child, *right_child;
};
```

`opcode` はノードの種類を表す。ここで扱う算術式は 1 文字の記号を使っているのでその文字コードを値とする。整数の場合には 0 をとるとする。`val` は整数ノードの場合のその整数の値を入れるためのメンバである。

そして、`left_child` と `right_child` はそれぞれ左右の子ノードへのポインタである。

この実装方法は、1 つのやり方であり、他のやり方もある点に注意しよう。

次にノードの作成であるが、ノードを作成する必要があるところで一々、そのためのコードを書くのは大変である。このため、ノードの作成は関数 `makenode` によって行うとする。

```
struct NODE* makenode
(struct NODE *l, struct NODE *r, int no, int val);
```

ここでは左の子、右の子、ノードの種類、(整数である場合の) 値を受け取って、新規にノードを作成し、そのノードに各値を設定するとする。そして、そのノードへのポインタをこの関数は返す。

5.2　Yacc 記述内で木を扱う

ここでは中置記法の算術式に対する構文解析時に抽象構文木を作成し、構文解析後、その抽象構文木を走査して後置記法を出力するプログラムを Yacc を用いて作るとする。

まず、文法を少し変更し、各生成規則でのアクションを付け、最後に得られた木の訪問を行う。

5.2.1　文法の変更

ここでは図 5 を改造することにしよう。

まず、図 5 では式を何回も入力できるようになっているが、ここでは 1 つの式だけを扱いたい。したがって 4〜5 行目を次のように変更する。

```
LIST : E NL ;
```

5.2.2　各生成規則でのアクション

字句解析器で整数を認識したときは `yylval` にその整数値を設定し、Yacc 記述の他の部分ではノードへのポインタを受け渡しする必要がある。もう 1 つの選択肢として、字句解析器で整数を認識した際には、そのためのノードを作成し、そのノードへのポインタを `yylval` の値とする方法もある。

ここでは前者を採用することにしよう。このため、%union を使う必要がある。

次に各生成規則でのアクションを考えよう。

図 5 の 18 行目では整数の処理をする。すなわち、その整数に対するノードを作成し、$$にそのポインタを渡す。同様に同図の 8, 9, 13, 14 行目では各演算子のためのノードを作成し、その被演算子のノード ($1 と $3 で得られる) を子ノードとする。

その他の部分では単にノードへのポインタを渡していくだけである。

1 つの式の入力が終了したら、つまり、改行が入力されたら、渡された根ノードへのポインタをどこかに保持しておく必要がある。ここでは Yacc 記述の冒頭部分で大域変数として、struct NODE *root; を宣言しておくとする。そして先ほど変更した規則では root = $2; として式 E から渡されてきたノードへのポインタをしまっておくとする。

5.2.3 木の訪問 (結果の出力)

木の訪問部分はコードを示しておく (図 16)。再帰関数になっていることに注意されたい。

```
 1 void traverse(struct NODE *n){
 2   if (n->lc != NULL){
 3     traverse(n->lc);
 4   }
 5   if (n->rc != NULL){
 6     traverse(n->rc);
 7   }
 8   if (n->nodecode == 0){
 9     printf(" %d ", n->val);
10   }
11   else {
12     printf(" %c ", n->nodecode);
13   }
14 }
```

図 16: 木を訪問する関数

簡単に説明すると、左の子があれば左の子を訪問する。そして、同様に右の子があれば右の子を訪問する。最後にそのノードが何であるかによって、出力を変える。そのノードが演算子であれば演算子を出力し、整数であれば整数値を出力する。

6 宿題

これまでと同様に提出義務はない。

ここで作成するプログラムは標準入力から式を受け取るとする (ファイルから受け取るようにしても構わない)。1 行を入れたら最後には Ctrl-d を入力する必要がある (ファイルからの入力の場合は不要)。

1. 上記の中置記法の算術式を後置記法に置き換えるプログラムを完成させよ。なお、このプログラムの実行形式を~nakai/IPP2/post として置いた。動作などを確認されたい。
2. 後置記法ではなく前置記法が出力されるプログラムを作成せよ。なお、このプログラムの実行形式を~nakai/IPP2/pre として置いた。動作などを確認されたい。
3. いま、C 言語の関数内の変数の宣言と使用の対応をチェックするプログラムを作りたいとしよう。C 言語の詳細を実装するのは大変なので、ここではいくつか制限を設けて簡単にしておく。
   1) 型は int 型だけとする。
   2) 宣言は int a; のように 1 つの宣言文では 1 つの変数だけが宣言できるとする。
   3) 宣言文は複数あってもよい。
   4) 宣言文の後ろでないと使用文が出現してはならない。
   5) 使用文は C 言語の識別子が使える算術式とする。
   6) 使用文は複数出現してよい。

   このような制限のもとで、使用文に現れる識別子がちゃんと宣言されているかチェックし、宣言されていない場合には「適切な」エラーメッセージを出力するようにせよ。

```
  1  %{
  2  #include <stdio.h>
  3  #include <stdlib.h>
  4
  5  #define MAXSIZE 1000
  6
  7  struct LNUM {
  8    int lineno;
  9    struct LNUM *next;
 10  };
 11
 12  struct LIST {
 13    struct LIST * next;
 14    char *word;
 15
 16    struct LNUM *h;
 17    struct LNUM *t;
 18  };
 19
 20  int lineno=1;
 21  struct LIST *h, *t, *tmp, *tmp2;
 22  struct LNUM *tmp3;
 23  %}
 24
 25  %%

省略

 59  [_A-Za-z][_A-Za-z0-9]*  {
 60    for(tmp2=h;
 61        tmp2->next != NULL;
 62        tmp2 = tmp2->next){
 63      if (strcmp(tmp2->next->word,
                   yytext)>0){
 64        break;
 65      }
 66    }
 67
 68    tmp3 = (struct LNUM*)
           malloc(sizeof(struct LNUM));
 69    if (tmp3 == NULL){
 70      perror("memory allocation error");
 71      exit(EXIT_FAILURE);
 72    }
 73
 74    tmp3->lineno = lineno;
 75    tmp3->next = NULL;
 76
 77    if (strcmp(yytext, tmp2->word)!=0){
 78      tmp = (struct LIST*)
            malloc(sizeof(struct LIST));
 79      if (tmp == NULL){
 80        perror("memory allocation error");
 81        exit(EXIT_FAILURE);
 82      }
 83
 84      tmp->word = (char*)
                  malloc(strlen(yytext)+1);
 85      if (tmp == NULL){
 86        perror("memory allocation error");
 87        exit(EXIT_FAILURE);
 88      }
 89      strcpy(tmp->word, yytext);
 90
 91      tmp->h = tmp3;
 92      tmp->t = tmp3;
 93
 94      tmp->next = tmp2->next;
 95      tmp2->next = tmp;
 96    }
 97    else {
 98      tmp2->t->next = tmp3;
 99      tmp2->t = tmp3;
100    }
101  }
102  .               { /* do nothing */ }
103  "\n"            { lineno++; }
104
105  %%
106
107  main(){
108
109    /* for making dummy leading node */
110    h = (struct LIST*)
          malloc(sizeof(struct LIST));
111    if (h == NULL){
112      perror("memory allocation error");
113      exit(EXIT_FAILURE);
114    }
115
116    t = h;
117    h->next = NULL;
118    h->word == "";
119
120    while(yylex()!=0){
121    }
122
123    for(tmp = h->next;
          tmp != NULL;
          tmp = tmp->next){
124      printf("%s : ", tmp->word);
125      for(tmp3 = tmp->h;
            tmp3 != NULL;
            tmp3 = tmp3->next){
126        printf("%5d", tmp3->lineno);
127      }
128      printf("\n");
129    }
130  }
```

図 17: 前回の宿題 3 の解答例