情報処理演習 II

コンパイラの作成

第 2 回 yacc を用いた構文解析器の作成

2005 年度担当: 中井央

宿題の解答

宿題の 1, 2 について、それぞれ解答例[1]を図 1、 図 2 に示す。

```
 1 %{
 2 int lineno=0;
 3 int wordno=0;
 4 int charno=0;
 5
 6 %}
 7
 8 %%
 9 [ \t]+     { charno+=strlen(yytext); }
10 "\n"       { lineno++; charno++; }
11 [^ \t\n]+  { wordno++; charno+=strlen(yytext);}
12
13 %%
14
15 main(){
16   while(yylex()!=0){
17   }
18
19   printf("%10d %10d %10d\n",
           lineno, wordno, charno);
20 }
```

図 1: 第 1 回テキストの宿題の 1. の解答例

1 概要

前回の Lex は字句解析器生成系であった。コンパイラは字句解析器からもらったトークンを使って構文解析を行う。

今回は Lex と連携して構文解析を行うプログラムを生成する Yacc について扱う。

[1]あくまでも「例」である。同じ動作をするならば他の記述の仕方でもよい。

2 構文と文法

ここでは構文と文法の関係について述べる。また、今後出てくる重要な事柄についての定義も行う。

2.1 文脈自由文法

前回のテキストでも述べたように言語の構文を表現するには文法を用いる。ここで扱う文法は文脈自由文法 (Context Free Grammar – CFG) と呼ばれるものである。CFG の例を示そう。

```
1 文    : 主部 述部;
2 主部 : 名詞
3       | 名詞 関係代名詞 文
4       ;
5 述部 : 動詞
6       | 動詞 目的語
7       | 動詞 目的語 補語
8       ;
```

図 3: CFG の例 (英文法もどき)

左端についている番号は説明のためのものである。1 つの CFG は複数の規則からなり、1 つの規則は

左辺 : 右辺;

の形をしている。この 1 つの規則のことを**生成規則** (production rule) と言う。

ここで、左辺には**1 つだけ**記号が来る。右辺には記号が幾つか (0 個でもよい) 並ぶ。同じ左辺を持つ場合は左辺を 1 つだけ書いて右辺を|で区切って複数並べることもできる。

```
 1 %{
 2 int lineno=1;
 3 %}
 4
 5 %%
 6 "auto"          { /* do nothing */ }
 7 "break"         { /* do nothing */ }
 8 "case"          { /* do nothing */ }
 9 "char"          { /* do nothing */ }
10 "const"         { /* do nothing */ }
11 "continue"      { /* do nothing */ }
12 "default"       { /* do nothing */ }
13 "do"            { /* do nothing */ }
14 "double"        { /* do nothing */ }
15 "else"          { /* do nothing */ }
16 "enum"          { /* do nothing */ }
17 "extern"        { /* do nothing */ }
18 "float"         { /* do nothing */ }
19 "for"           { /* do nothing */ }
20 "goto"          { /* do nothing */ }
21 "if"            { /* do nothing */ }
22 "int"           { /* do nothing */ }
23 "long"          { /* do nothing */ }
24 "register"      { /* do nothing */ }
```

```
25 "return"        { /* do nothing */ }
26 "short"         { /* do nothing */ }
27 "signed"        { /* do nothing */ }
28 "sizeof"        { /* do nothing */ }
29 "static"        { /* do nothing */ }
30 "struct"        { /* do nothing */ }
31 "switch"        { /* do nothing */ }
32 "typedef"       { /* do nothing */ }
33 "union"         { /* do nothing */ }
34 "unsigned"      { /* do nothing */ }
35 "void"          { /* do nothing */ }
36 "volatile"      { /* do nothing */ }
37 "while"         { /* do nothing */ }
38
39 [_A-Za-z][_A-Za-z0-9]*  { printf("%4d: %s\n",
                             lineno, yytext); }
40 .               { /* do nothing */ }
41 "\n"            { lineno++; }
42
43 %%
44 main(){
45   while(yylex()!=0){
46   }
47 }
```

図 2: 第 1 回テキストの宿題の 2. の解答例

図 3 を見ると左辺に現れる記号と右辺にしか現れない記号がある。右辺にしか現れない記号のことを**終端記号**(terminal symbol) という。左辺に現れる記号のことは**非終端記号**(non-terminal symbol) という。

文法の適用を始める最初の非終端記号のことを特に**開始記号**(start symbol) という。開始記号が 2 つ以上の右辺を持つことは可能である。しかし、右辺をただ 1 つになるようにしておくと構文解析をする際、都合が良い。元の開始記号 S に対し、S' : S のような規則を導入することでこれを実現できる。このように文法を拡張したものを**拡大文法**(augmented grammar) と呼ぶ。

2.2 記号の使い方の慣習

一般的な話をするためには記号を用いると便利である。これ以降の記述では以下の慣習に従って記号を用いる。なお、各記号が何を意味するかはできる限り付すようにする。

- 非終端記号はアルファベットの前の方の大文字 $(A, B, C, \cdots)$ を用いる
- 終端記号はアルファベットの前の方の小文字 $(a, b, c, \cdots)$ を用いる
- 文法記号の列にはギリシャ文字を用いる $(\alpha, \beta, \gamma, \cdots)$
- アルファベットの終わりの方の大文字 (X, Y, Z など) は文法記号 (非終端記号もしくは終端記号) を表すことがある
- アルファベットの終わりの方の小文字 (x, y, z など) は終端記号列 (終端記号の 0 個以上の並び) を表すことがある
- つづりをそのまま終端記号として扱う場合、表記としてタイプライタ体 (通常の a と書くのと違い a と書く) を用いる[2]
- 以下で扱う Yacc に与える記述に関しては全体をタイプライタ体で記す。

2.3 構文解析木

図 3 を使って構文解析について説明していこう。ここで I love you という入力があったとしよう。まず、

[2] フォントの違いに注意してほしい。

1 の規則により、文は主部と述部からなる。この入力の場合、主部は 2 の規則により名詞になる。同様に述部は 6 の規則により動詞と目的語からなる。このようにして入力された `I love you` という文が文法に従っていることが確認できた。これを図示してみよう。

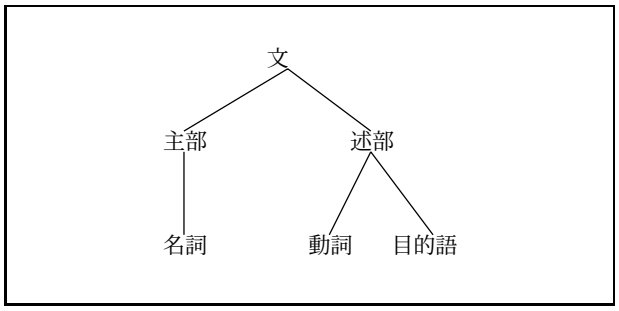

図 4: `I love you` に対する構文解析木

図 4 のような図を構文解析木という。一般にコンピュータサイエンスでは図 4 のような木は、この図での「文」、「主部」などの節とそれらをつなぐ線で描かれる。節のうち、図の一番下側に書かれていてそれ以上線が出ていない節を特に「**葉**」(leaf) と呼ぶ。同様にすべての節の中で一番上にある節を「**根**」(root) という。(プログラマの??) 日常、日本語の会話の中でも「ルート」という呼び方をする場合もある。

2.4 導出と還元

構文解析木は、生成規則を開始記号からいくつか適用することで入力の文を**生成した**軌跡とみなせる。つまり、開始記号 (図の根) から順に生成規則を適用し、入力に合うように木を構築していった。このことを「**導出**」(derivation) という。

もちろん、最初に与えられるのは入力文であるから、それをもとにどの規則が適用されたのか推測しながら、最後に根の節を表す「文」に行き着くようにして解析木を作っていくことも考えられる。つまり、導出とは逆の方向に作業を行っていくのである。このことを「**還元**」(reduction) という。

2.5 文法と言語

もう一度「導出」という考え方に着目しよう。文法があれば任意の文を作り出すことができる。我々は、図 3 の文法よりももう少しちゃんとした文法を中学や高校で習っている。文法では数十個か数百個の規則でほとんどすべての文 (英文) のパターンを示している。そして我々は文を作ることについて無限の可能性があることを知っている。

ある文法によって導出される文の集合を**言語**という。

図 3 を見ると 3 行目の規則には右辺に「文」が出てきている。これにより、無限に長い文が生成できることがわかるだろうか。解析木にして考えてみるとわかるが、文から導出した部分の中にまた文が出てくるのである。

“The car that I used is my father’s.” [3]のような文を考えるとこれ全体は文であるが、“I used” の部分も文である。

このように文法が定義されていることを**再帰的に**(recursively) 定義されているという。この考え方は文法を記述する上では非常に重要になってくるので注意しておいてもらいたい。

2.6 構文解析法の種類

プログラミング言語の構文解析 (syntax analysis もしくは parsing) は上の概念を基にしてソースファイルに書かれた内容を文法に沿っているかチェックしていくことである。発想としては導出をもとに考えられた下向き構文解析法と還元をもとにして考えられた上向き構文解析法がある。Yacc では上向き構文解析法を用いている。

上向き構文解析法は、葉を見てそこに適用された規則を決定し、これを繰り返すことで根まで木を作り上げていく方法である。より詳細にこの解析方法の理論について知りたい場合は [3] などを参考にするとよい。

3 Yacc 入門

入力された文字列がある文法に沿っているかどうかを判定するプログラムを構文解析プログラムあるいは構文解析器もしくはパーザ (parser) という。

まず、ここでは算術式 (arithmetic expression) を表す文法から始めることにしよう。

[3] この文は例文としてあまりよくないけど…。

3.1 Yacc 記述の例

最初は足し算のみの式の文法を考える。これを Yacc の仕様記述 (以降では Yacc 記述と述べる) として書いたものを図 5 とする。

```
 1 %token NUM
 2 %token NL
 3 %%
 4 LIST    : LIST E NL
 5         | /* empty */
 6         ;
 7
 8 E       : E '+' NUM
 9         | NUM
10         ;
11
12 %%
13
14 #include "lex.yy.c"
15
16 main(){
17   yyparse();
18 }
```

図 5: 算術式の yacc 記述 (ex1.y)

図 5 のための Lex 記述を図 6 に挙げる。

```
1 %%
2 [0-9]+          { return NUM; }
3 "+"             { return '+'; }
4 [ \t]           { /* do nothing */ }
5 "\n"            { return NL; }
6 .               { return yytext[0]; }
```

図 6: 図 5 のための lex 記述 (ex1.l)

では、図 5 について説明しよう。まず、1 行目の %token NUM は、文法中に出てくる NUM という記号が終端記号であることを宣言している。2 行目も同様である。ここで NL は new line つまり、改行コードを表すつもりで付けられている。ここで注意だが、非終端記号と終端記号の名前としてはどのように付けても構わない。しかし、通常はそれが意味するものの名前をつける。だからここでは NL とせず、NEWLINE としてもよいわけである。別の言い方をすれば C 言語などで変数名をつけるのと同じである。

3 行目の%%は Lex の場合と同様であり、これ以降が文脈自由文法[4] を記述する部分であることを表している。4〜10 行目までが文脈自由文法である。まず、4〜6 行目は式の並びを表している。そして 8〜10 行目は式を表している。

まず、先に 8〜10 行目の式の構文について説明しよう。

8 行目では、E や NUM のほかに'+' という書き方をしている。1 文字からなる終端記号に限りこのようにシングルクォートで括って書いてもよいことになっている。

8 行目と 9 行目を理解するには次のように捉えるとよい。まず先に 9 行目から見てみる。E は式を表している (そのつもりで Expression の頭文字である E をつけている)。すると 9 行目の規則は「数は式である」と解釈できる。単独の数のみを式と見なすのである。次に 8 行目の規則に着目しよう。「式に数を足したもの、つまり、式 + 数は式である」と解釈できる。

この再帰的な表現によって、...+...+...+... といった演算子として + のみを含むすべての式を表現できていることに注意されたい。

さて、戻って 4〜6 行目に着目しよう。4 行目も再帰的な表現になっている。これにより、LIST が繰り返し出てきてもよいことになる。LIST は LIST に式と改行コードが続いたもの、と表現されている。ここで重要な役割を担うのが 5 行目の/* empty */である。実際には何も書かなくてもよい。C 言語のコメントを書いているのである。つまり、C 言語の場合と同様にこの記述は無視される。「**何もない**」という規則が重要になる。つまり、1 行分の式は LIST 部分が「何もない」で、次に式 E が来てそして改行コード NL が来る形となる。さらに次の式が現れた場合は、「この」LIST に式 E と改行コード NL が続いたものとなる。

12 行目の%%も Lex の場合と同様であり、以降に C 言語の関数を記述することができる。

14 行目は C 言語での#include マクロである。ここでは lex.yy.c を読み込んでいる。つまり、字句解析プログラムを直接ここに埋め込んでいる。これについては後述する。

16〜18 行目は main 関数である。前回やったのと同

[4] 厳密には文脈自由文法のサブクラスとなる LALR(1) 文法である。

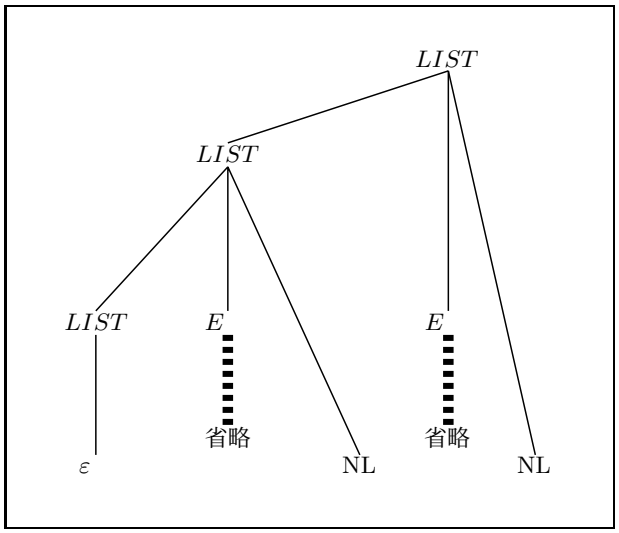

図 7: 図 5 の解析木の例

様に今度は構文解析関数を呼び出す。構文解析関数の名前は yyparse である。

図 6 は図 5 のための字句解析器を作るための Lex 記述である。こちらについても説明しておこう。

2 行目は整数にマッチした場合の処理を記述している。アクション部分に注目してほしい。return NUM として、NUM を返している。Yacc は%token を使って終端記号として記述された文字列をそのままマクロ名として、それぞれに一意な番号をつけている。たとえば、%token NUM とあった場合、それに対しては Yacc が生成するプログラムに#define NUM 257 のようなコードを作り出している。したがって、Lex ではその終端記号に一致したら、NUM のような、Yacc 記述の中で書いた終端記号の文字列を使って return 文を書くことで、その終端記号が見つかったことを yyparse に知らせることができる。

3 行目は + が見つかった場合の処理である。Yacc 記述 (図 5) で'+' のように記述しているので、ここでも return '+' のようにして値を返している。つまり、+ のアスキーコードを返り値としている。

4 行目は半角空白、タブが見つかった場合には何もしないことを定義している。つまり、単に読み飛ばしている。

5 行目は改行コードが見つかった場合の処理である。

6 行目ではそれ以外の文字が見つかった場合の処理をしている。ここではその文字のアスキーコードを返却している。yytext にはマッチした**文字列**が入れられている。実際には yytext はその文字列が格納されているメモリへのポインタであるから、yytext[0] とやることでその先頭文字、つまり、今マッチした文字のアスキーコードを取り出せるわけである。

3.2 Yacc と Lex によるコンパイラの作成

Yacc は Yacc 記述ファイルを受け取るとその文法に従った構文解析プログラムを y.tab.c に出力する。このため、実行可能形式を得るまでの手順は次の通りである。

```
% yacc ex1.y
% lex ex1.l
% cc y.tab.c -ly -ll
```

なお、ここで Yacc 記述中の#include "lex.yy.c" の部分について述べる。Lex は lex.yy.c を出力し、Yacc は y.tab.c を出力する。本来、それぞれは main 関数を含んでいない。また、C 言語でのプログラミングでは一般的にはいくつかのファイルに分割してプログラムの作成を行う。このとき、てっとり早くいくつかのプログラムファイルを 1 つにまとめる方法として #include を使う方法がある。#include は指定したファイルをその部分に読み込むことを C プリプロセッサ (preprocessor 前処理をするプログラム) に指示する文である。このため、上の例では y.tab.c だけをコンパイルの対象としている。

ここで-ly は Yacc に関するライブラリを使用するためのオプションである。これは常に付けてコンパイルする必要がある。また、通常は Lex のライブラリも併用するが、並べる順番は-ly -ll となる。

演習 1

図 5 と図 6 をそれぞれ入力し、上の手順に従ってコンパイルせよ。また、できたプログラムを実行してみよ。

できたプログラムは標準入力 (キーボード) から入力を受け付けるようになっている。

例えば、1+2+3 などと入力しリターンキーを押す。同様に abc と入力したらどうなるか。

何回かこのようなことを行って動作を確認しよう。

3.3 Yacc でのアクション

Yacc でもある生成規則にマッチしたら何かアクションを起こしたい。例えば、先ほどの例の場合、1 つの式にマッチしたらその式に一致したことによる何か行動を起こしたいわけである。

まわりくどく言うのはやめてここでは足し算をすることにしよう。図 8 が算術式を認識したときに足し算を計算する Yacc 記述であり、図 9 がその Yacc 記述のための Lex 記述である。

```
 1 %token NUM
 2 %token NL
 3 %%
 4 LIST : LIST E NL { printf("%d\n", $2); }
 5      | /* empty */
 6      ;
 7
 8 E    : E '+' NUM { $$ = $1 + $3; }
 9      | NUM { $$ = $1; }
10      ;
11
12 %%
13
14 #include "lex.yy.c"
15
16 main(){
17   yyparse();
18 }
```

図 8: 算術式を計算する Yacc 記述 (ex2.y)

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

図 9: ex2.y のための Lex 記述 (ex2.l)

では、まずは Lex 記述の方から見ていくことにしよう。図 9 の 2 行目では yylval という変数を利用している。これは Lex で字句を認識した際になんらかの計算を行って、その結果を Yacc でのアクションに利用したい場合に使う変数である。

```
 1 #include <stdio.h>
 2
 3 main(){
 4   char a[100];
 5   int x, y;
 6
 7   fgets(a, 100, stdin);
 8   x = atoi(a);
 9
10   fgets(a, 100, stdin);
11   y = atoi(a);
12
13   printf("%d\n", x+y);
14 }
```

図 10: atoi の使用例

2 行目のアクションは数字に一致した場合の処理である。図 6 のときは単に NUM というのを返し、数字として認識したことを Yacc に伝えるだけであった。今回はその数字の持つ値 (value) を Yacc 側に渡したいわけである。数 字 の列は所詮文字の列でしかない。これを**数値**に変換する必要がある。整数を表す数の文字列から整数値へ変換するには atoi という関数を使う。

Lex 記述の変更はこれだけである。つまり構文解析器側に認識した数字に対応する数値を渡すことを付け加えた。

テストプログラムを書く

ここではおさらいの意味もこめて図 10 のようなプログラムを入力し、実行してみよう。
このような単純なテストプログラムが書けることは重要なことである。fgets でキーボードからの入力として 123 などを持ってきても、それは文字列であるから直接それによって足し算などの演算はできない。演算をするには数値に変換する必要がある。

次に Yacc 記述の変更点について述べる。

まずは図 8 の 8 および 9 行目について述べる。ここは算術式を表している文法規則が書かれている。付け足されたのはその文法規則に一致した場合のアクションである。{ と }の間に C 言語の断片を記述することができるのは Lex と同様である。ここではこの他の特別な記法として$のついたものを利用している。

9 行目に着目すると右辺は終端記号である NUM だけであり、その左辺は非終端記号 E である。一般には右辺に複数の文法記号 (非終端記号もしくは終端記号) が並ぶ。この n 番目に対応する記号の値を取り出すときに$\$n$ を使う。9 行目の場合は$1 と書いてあり、右辺の 1 番目の記号、つまり NUM **に対応する値**を取り出している。この値とは何かというと NUM は終端記号であり、字句解析器で認識されていることを思い出せば、Lex 記述で使用していた yylval の値であると見当がつく。

厳密に言えば、終端記号の値は Lex 記述で yylval に代入することによって、対応する Yacc 記述の終端記号に関係するシステム内の場所[5] に値を代入していることになる。

さて、細かいことを抜きにすると、字句解析器である字句を認識したとき、構文解析器に値を渡したければ、Lex 記述において yylval に代入すればよく、字句解析器でそれを使用したい場合には、Yacc 記述内で$\$n$(くどいが n は生成規則中の右辺での出現の番号)として参照すればよいわけである。

同様に$$は左辺の文法記号に対応する値をしまう場所を示す。

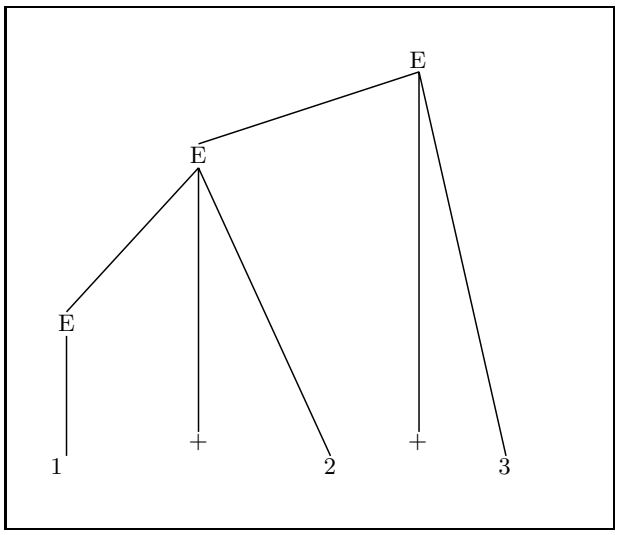

図 11: 算術式 (1+2+3) の解析木

いま、算術式 1+2+3 を解析したときの解析木は図 11 のようになる。この図で言うと左端の 1 とそこから伸びている線上の E は 9 行目の規則によるものである。このとき、1 を認識するのは字句解析器であり、そのとき atoi で得た値を yylval に代入している。そして 9 行目の規則にマッチしたとき、$$=$1 が実行される。これはすなわち字句解析器から与えられた値が E のノードに伝えられたことになる。

もう少し進めてみると、 + を読んで 2 を読むと今度は左端はもう E になっているから E、+ 、2 という形になり、今度は 8 行目の規則にマッチしたことになる。8 行目の規則にマッチしたら今度は$$=$1+$3 が実行される。ここでは$1 は先ほどの E に伝えた値のことであり、$3 は字句解析器から渡された 2 に対応する値である。このため、ここでは結果として $1+2$ の演算が行われ、左辺である (図 11 の左から二番目の)E に対応する構文解析器内での 場所 に値がしまわれる。

残りの + と 3 を読んだ場合も同様である。こうしてこの式の値としては 6 が得られることになる。つまり、図 11 の右端の E は値 6 を持つ。

さて、この段階で改行が入力されると今度は 4 行目の生成規則にマッチすることになる。ここでは$2 は E に対応しており、つまり、先ほど計算された 6 をしまってある場所を指している。よって、ここでのアクション printf("%d\n", $2) では先ほど計算された 6 が出力されるのである。

3.4 演算子の優先順位

これまでは + 演算だけだったのであまり面白くもなかった。ここでは演算子を増やすことを考えてみよう。優先順位の違う演算子を導入する場合には注意しなければならないことがある。ここではまず、失敗例を示し、その後、正しく解析する例を示す。

3.4.1 失敗例

まずは掛け算を導入しよう。掛け算の記号は * を使う。ここでは図 8 をまねて作ってみよう。図 8 の 8 行目と 9 行目の間に

```
    | E '*' NUM { $$ = $1 * $3; }
```

と入れ、図 9 の 3 行目と 4 行目の間に

```
"*"     { return '*'; }
```

[5] もう少しきちんと理論を習えばわかることだが、構文の認識にはスタックを用いている。そのスタックのことをここでは「場所」と言っている。

と入れることにしよう。このファイルを便宜上それぞれ ex3.y と ex3.l とする。まず、これらをそれぞれ Yacc と Lex に与えて、コンパイルし、実行してみよう。

例えば、1+2*3 を入れたら果たしてどうなるだろうか。この時点では上記の通りにまず、実行してみてほしい。

3.4.2 正しく動く例

さて、まず、上の方法ではなぜ失敗なのかを見てみよう。これまでと同様に 1+2*3 に対する構文解析木を描いてみよう。図 12 に ex3.y による構文解析器で入力 1+2*3 に対する解析を行った際の構文解析木を示す。

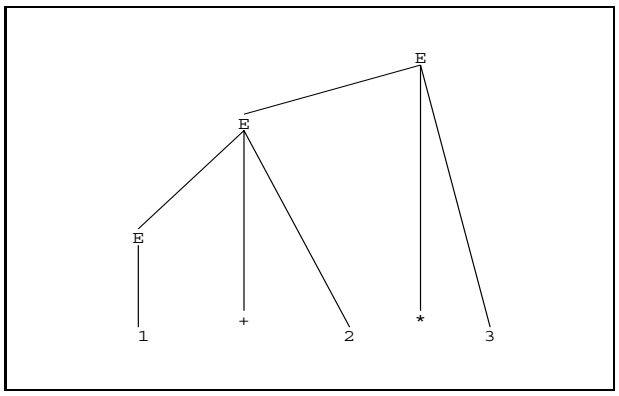

図 12: 1+2*3 に対する構文解析木

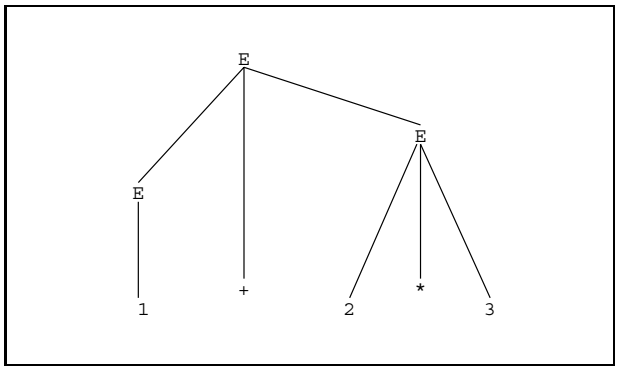

図 13: 1+2*3 に対する (正しい) 構文解析木

構文解析器は入力を左から順番に処理していく。まず、1 を見て、それは図 8 の 9 行目に一致する。次に + を見て、2 を見たら、図 8 の 8 行目の規則に一致する。それが E になり、その次の入力は * で、さらに次は 3 であり、これにより、先ほど挿入した規則に一致する。これで図 12 のように解析木ができる。この木では先に 1+2 が結び付けられている。つまり、先に 1+2 という演算が行われている。これは我々の知っている四則演算の順序とは違う。すなわち、構文解析木で言うと図 13 のようになっていてほしいわけである。実際にはこの図とは少し異なるのだが、先に 2*3 の方を処理したいという意味で、この図は我々の意図を反映している。

このようにするためのテクニックを入れた文法を図 14 に示す。

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

図 14: 演算子の順位を導入して ex3.y を拡張 (ex4.y)

図 14 による解析器で 1+2*3 を解析した際の構文解析木を図 15 に示す。

では、図 15 の解析木を見ながら、図 14 による解析器に 1+2*3 を入力をした際の構文解析の様子を見ていこう。

まず最初の 1 を読んだときは 13 行目の規則にマッチする。これにより、T のノードが作られる。詳しい説明を省略するが、次の入力が + なのでここでは 9 行目の規則が適用され、左端の E のノードが作られる。この段階で + が読まれ、次に 2 の処理に移る。まず、

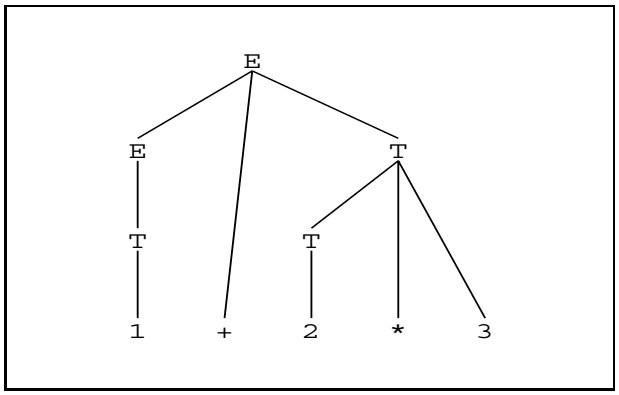

図 15: 1+2*3 に対する（正しい）構文解析木

13 行目の規則にマッチする。これにより左から二番目の T のノードが作られる。この段階では選択肢が 2 つある。1 つは 8 行目の規則によってまとめられる場合ともう 1 つはそれ以降の部分を読んで 12 行目の規則でまとめる場合である。このとき次の入力が + や NL ならば 8 行目の規則によってこれまで読んだ部分がまとめられる。しかし、ここでは次の入力が * なので、12 行目の規則でまとめるために先に * を読んでその次の 3 まで処理を進める。3 まで読んだ段階で 12 行目の規則を適用し、右端の T のノードを作成する。そしてこの段階では次の入力はないため、E + T というノードたちを 8 行目の規則でまとめる。これにより一番上の E のノードが作られる。

言葉で説明するのは少し難しいが、ここで注意すべき点は非終端記号 T を導入したことで、解析木を作る過程に段階を設けたことである。演算子として順位が強いものを先に括れるようにそれを部分木となるようにするのである。言葉での説明より、これまでの図による説明の方がわかりやすいのではないだろうか。

演習 2

1. ex4.y を使って構文解析器を作り、入力 1+2*3 などを実行してみよ。
2. ex4.y を改造し、演算子として- と / を導入せよ。改造する場合、Yacc 記述のファイル名を ex5.y とし、それ用の Lex 記述のファイルを ex5.l とせよ。実際に構文解析器を作り、入力を変えて実行せよ。
3. 上記の説明をもとに適当な入力 (例えば 2+3*4*5 など) に対する構文解析木を手で紙に描いてみよ。

3.5 繰り返しのテクニック – 左再帰と右再帰 –

これまでに例えば、足し算の式を表現するために生成規則として E : E '+' NUM のようなものを見てきた。この規則では左辺の記号 E が右辺にも現れる形になっている。このような生成規則を再帰的な規則といった。この再帰的な規則のおかげで繰り返しを表現できる。

再帰的な表現として E : NUM '+' E という書き方もできる。この規則によって 1+2+3 を構文解析したときの構文解析木は図 16 のようになる。

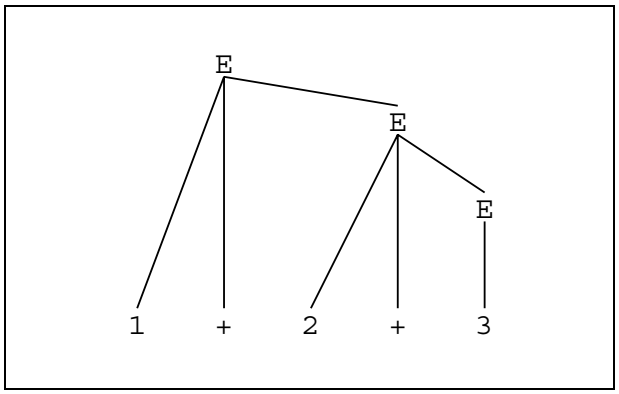

図 16: E : NUM '+' E による右再帰的な構文解析木

図 16 は図 11 と比べるとわかるが、最も右のものからまとめる形になっている。足し算の場合は問題ではないが引き算だと図 16 のまとめかたと図 11 のまとめ方では演算結果が変わってくる。

算術式などで演算子を適用する際に左から順に適用するようなものを左結合という。四則演算は左結合である。ところで、C 言語では a=b=c=3; のような文が書ける。この場合、c=3 が実行され、その結果 (3) が b に代入され、その結果が a に代入される。=は**代入演算子**と呼ばれるが、この演算は右結合ということになる。

先ほども述べたように - などは結合の仕方によって演算結果が異なるので留意する必要がある。

演習 3

1. C 言語の変数宣言は次のように書ける。

```
int i, j, k;
```

このi,j,kの部分を表す文法を左再帰を用いて記述せよ。また、同様にこれを右再帰を用いて記述せよ。なお、最低でも1個の識別子があることに注意せよ。図5のEなどを参考にせよ。ここではiやjなどを示すものは識別子としてIDという終端記号を使い、再帰的に書くための非終端記号をIDLISTとし、区切りであるコンマはCOMMAとする。

2. C言語の宣言部分は例えば次のように宣言をする文が何行か並んだものである。

```
int i;
int j;
int k;
```

また、初心者用の最初のプログラムであるhello, worldを表示するプログラムには宣言はない。この宣言部分を表す文法を記述せよ。ここでは、1つの宣言の文はDECLという文法記号で表現することにする。つまりDECLは1つも出てこなくてもよいし、1つ以上何回も出てきてもよい。図5のLISTなどを参考にせよ。この文法についても左再帰および右再帰の両方を書いてみよ。DECL の繰返しは DECLSという日終端記号で表現するとする。

3.6 あいまいな文法

Yacc入門の最後としてあいまいな文法について述べておく。あいまいな文法とは、同じ入力に対して2つ以上の異なる解析木が得られるような文法のことを言う。まず、文法の例を示そう。

```
E   : E + E
    | NUM
    ;
```

足し算の文法をこのように書いたとしよう。そして1+2+3に対して解析木を作ろうとすると図17のように2通りできる。

このような文法が与えられた場合、Yaccとしては図17の(a)もしくは(b)のどちらの解析木を作るような解析器を生成してよいかわからない。

しかし、文法を記述しているときには知らずにあいまいな文法を書いている場合も多い。Yacc記述としてはあいまいではない文法を与えることが基本である。しかし、場合によってはあいまいな文法を用いる方が効率的である場合もある。ここではYacc記述中にあいまいな文法を書く方法については省略する。Yaccに関する参考書を参照されたい。

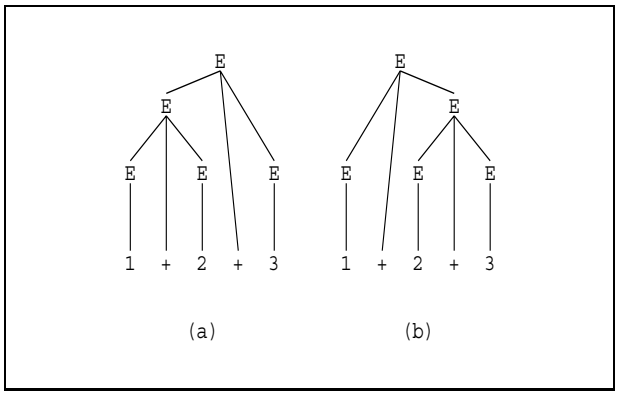

図 17: あいまいな文法よる1+2+3に対する構文解析木

4 コンパイラで用いるデータ構造

C言語を用いて実用的なプログラムを作るほとんどの場合、動的なメモリ確保を用いる。リンクトリスト(linked list)やツリー(tree)がその例である。特にツリーはコンパイラとは切っても切れない。これまでに見たように構文解析木を構築し、その上で意味解析以降の操作をするコンパイラも多い。

構造体とポインタ、そしてそれを用いたリンクトリストについては各自で復習しておいてもらいたい。

演習 4

1. 次のプログラムを作れ。
   (a) main関数の中に2つの変数aとbを用意する。
   (b) それぞれ、1と2に初期化する。
   (c) 関数hogeを呼び出し、その2つの変数の値を入れ替える。(関数内で入れ替えた結果としてmainのaとbが入れ替わるようにする)
   (d) mainプログラム内ではhogeを呼ぶ前後でaとbの値を表示し、変化が確認できるようにせよ。

2(a) main関数の中に2つの変数aとbを用意する。
   (b) それぞれ、5と3に初期化する。

(c) 関数 hoge で a+b の結果を a に、a-b の結果を b に入れる。(この場合の a と b は main のものを意味する)

(d) main プログラム内では hoge を呼ぶ前後で a と b の値を表示し、変化が確認できるようにせよ。

5 宿題

前回同様、宿題の提出義務はない。

現在のところ、Lex または Yacc を使ってプログラムを作成した際は、入力は標準入力からしか受け付けない。このため、作成した実行形式にファイルの内容を与えるにはリダイレクトを使うこと。

[**例**]

```
dream% a.out < file
```

1. 標準入力から受け取った文字列をリストにつないでいくプログラムを作成せよ。ただし、入力として owari がきたら、つなぐ作業を中止し、これまでにつないだリストの要素を 1 行に 1 要素出力するようにせよ。
2. 前回の宿題の C 言語の識別子を認識するプログラム次のように改造せよ。
   (a) 識別子を認識した段階では線形リストに登録するだけで、ファイルからの読み込みが終了した段階でそのリストを先頭からたどることで識別子を出力する。
   (b) 上記の問題で、リスト中の各識別子はアルファベット順に格納されているようにせよ[6] 。
   (c) 上記をさらに改造し、同じ識別子が出てきたら重複してつながないようにせよ。
3. 前回の宿題の C 言語の識別子を認識するプログラムを改造してクロスリファレンサを作ろう。クロスリファレンサとはある変数がそのソース中の何行目で出てきたかを示すプログラムである。リストの各要素には識別子を保存するほかにその識別子が出現する全ての行番号を保存する必要がある。しかし、その識別子が何回出現するかはわからない。このため、各要素はその出現した行番号のリストを持っていることになる[7] 。
4. 演習 2-2 の拡張として (と) を導入し、括弧で括った式を先に演算するようにせよ。

[6] **ヒント:** リストにつなぐ際に適切な位置に挿入せよ。

[7] **ヒント:** リストが二重になっている点に注意。まず、識別子のリストがある。そのリストの各要素は次の識別子セルへのポインタとその識別子の出現行番号リストの先頭ポインタからなる。新規にその識別子が現れた場合は、そのためのセルを作り、また、その識別子が現れた行番号をしまうためのセルも作り、その出現行番号リストの先頭とする。その識別子の 2 度目以降の出現では、その識別子のセルにたどり着いた後、行番号リストをたどり、その末尾に現在の行番号のセルをつなげる。

6 コンパイラの参考文献

コンパイラの参考文献を巻末に挙げておく。

7 演習問題の解答

演習 1 省略

演習 2

1. 省略
2. 図 18 参照

```
 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 { $$ = $1; }
11      ;
12
13 T    : T '*' NUM { $$ = $1 * $3; }
14      | T '/' NUM { $$ = $1 / $3; }
15      | NUM { $$ = $1; }
16      ;
17
18 %%
19
20 #include "lex.yy.c"
21
22 main(){
23   yyparse();
24 }
```

図 18: 演習 2-2 の解答例

3. 省略

演習 3

1. 左再帰の文法は次のようになる。

```
IDLIST : IDLIST COMMA ID
       | ID
       ;
```

右再帰の文法は次のようになる。

```
IDLIST : ID COMMA IDLIST
       | ID
       ;
```

2. 左再帰の文法は次のようになる。

```
DECLLIST : DECLLIST DECL
         | /* empty */
         ;
```

右再帰の文法は次のようになる。

```
DECLLIST : DECL DECLLIST
         | /* empty */
         ;
```

演習 4 1. は図 19、2. は図 20 を参照。

```
 1 #include <stdio.h>
 2
 3 void hoge(int*, int*);
 4
 5 main(){
 6   int a=1;
 7   int b=2;
 8
 9   printf("a = %d\nb = %d\n", a, b);
10   hoge(&a, &b);
11   printf("a = %d\nb = %d\n", a, b);
12 }
13
14 void hoge(int *x, int *y){
15   int tmp;
16
17   tmp = *x;
18   *x  = *y;
19   *y  = tmp;
20 }
```

図 19: 演習 4 の 1 の解答例

```
 1 #include <stdio.h>
 2
 3 void hoge(int*, int*);
 4
 5 main(){
 6   int a=5;
 7   int b=3;
 8
 9   printf("a = %d\nb = %d\n", a, b);
10   hoge(&a, &b);
11   printf("a = %d\nb = %d\n", a, b);
12 }
13
14 void hoge(int *x, int *y){
15   int tmp1, tmp2;
16
17   tmp1 = *x + *y;
18   tmp2 = *x - *y;
19   *x   = tmp1;
20   *y   = tmp2;
21 }
```

図 20: 演習 4 の 2 の解答例

参考文献

[1] Levine, J. R., Mason, T., and Brown, D.: *lex & yacc*, O'Reilly & Associates, Inc., 1991.
(邦訳) 村上列:『lex & yacc』, アスキー出版局 (1994)
備考: ISBN4-7561-0297-2
定価 本体 3600 円＋税.

[2] 中田育男: **コンパイラ**, オーム社, 1995.

[3] 佐々政孝: **プログラミング言語処理系**, 岩波書店, 1989.

[4] 五月女健治: *yacc/lex* **プログラムジェネレータ** *on UNIX*, テクノプレス, 1996. 備考: ISBN4-924998-14-1
定価 本体 3400 円＋税.