記号処理・第 4 回

2017 年 5 月 2 日

今回の内容

4.1 構文図からのパーザの生成

この節では、単に、プログラムの標準入力から入力された文字列が与えられた文法に合致しているかどうかを判定することだけを目標に、構文解析を行うプログラムを構築する方法を考えます。構文解析を行うプログラムを一般に**パーザ** (parser) と呼びますが、この節で考えるパーザは、入力があらかじめ与えられた文法に合致していれば何もせずに終了し、入力に文法エラーが含まれていればそのことを通知して終了するだけの機能を持ちます。

入力をトークン (終端記号) に分解する (字句解析を行う) プログラムが必要となりますが、ここでは、文字 1 つ 1 つをトークンとし、次のような C プログラムによって字句解析部が実現されているものとします。

```
#include <stdio.h>

int token;

void gettoken(void)
{
    do {
        token = getchar();
    } while (token == ' ' || token == '\n' || token == '\t');
}
```

この関数 `gettoken()` は、標準入力から (空白類を無視しながら) 1 文字を読み取り、読み取った文字の文字コードを大域変数 `token` に代入してリターンするものです。パーザは、`gettoken()` の呼び出しごとに `token` にセットされる文字コードの列としてソースプログラムを認識することになります。

メモ

また、構文解析中に文法エラーを検出した際には、次のような関数 `error()` を呼び出して全体の処理を終了するものとしておきます。

```
#include <stdlib.h>

void error(void)
{
    printf ("文法エラーを検出しました\n");
```

```
    exit(1);
}
```

メモ

4.2 パーザの構造

決定的な構文図が与えられている場合、その構文図の構造に従ってパーザを構築することができます。それぞれの非終端記号ごとに、その非終端記号が生成するトークンの列を認識する 1 つの関数を定義することによって、パーザ全体を構築します。非終端記号 N の認識を担当する関数 N は、

1. N の構文図の先頭のトークン (終端記号) が 1 つ読み込まれた状態で呼び出され、
2. 読み込んだトークンに応じて分岐を選びながら、構文図の入り口から出口までの流れを辿り、
3. N の構文図の出口の先にあるトークン (終端記号) を 1 つ先読みした状態で呼び出し元に戻る

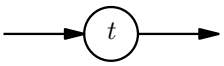
という仕事を行います。このような関数 N は、

```
void N(void)
{
    Nの構文図
}
```

のように定義しますが、N の本体のプログラム [N の構文図] の部分は、非終端記号 N の構文図の形に応じて、以下のように (機械的に) 定めることができます。

メモ

1. 構文図が

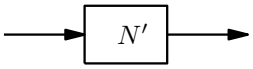


の形をしている場合、この構文図に対応するプログラムは、

```
if (token == 't')
    gettoken();
else
    error();
```

のようにします。つまり、読み込んだトークンが t と等しければ次のトークンを読み込み、さもなければ文法エラーとします。

2. 構文図が

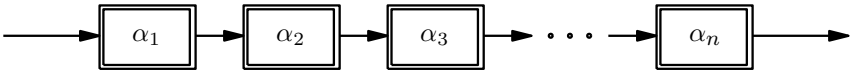


の形をしている場合、この構文図に対応するプログラムは、

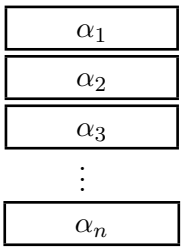
```
N'();
```

のように、その非終端記号 $\langle N' \rangle$ に対応する関数の呼び出しになります。$\langle N' \rangle$ が $\langle N \rangle$ と等しい場合、つまり、$\langle N \rangle$ の構文図に $\langle N \rangle$ 自身が現れていた場合、この関数呼び出しは、現在定義しようとしている関数 N に対する再帰呼び出しになります。

3. 構文図が

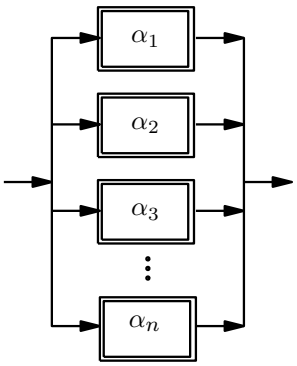


の形をしている場合、この構文図に対応するプログラムは、



のように、この構文図を構成している各 α_i の形によってに定まるプログラム $\boxed{\alpha_i}$ を順に並べたものになります。

4. 構文図が



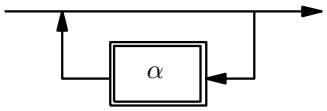
の形をしている場合は、それぞれの構文図 $\boxed{\boxed{\alpha_i}}$ の先頭に現れる可能性のあるトークンを c_{i1}, ..., c_{im_i} とすると、この構文図に対応するプログラムは、

```
switch (token) {
case 'c11': case 'c12': ... case 'c1m1':
    α1
    break;
case 'c21': case 'c22': ... case 'c2m2':
    α2
    break;
```

```
                    ⋮
case 'cn1': case 'cn2': ... case 'cnmn':
    [ αn ]
    break;
default:
    error();
}
```

のようにします。

5. 構文図が

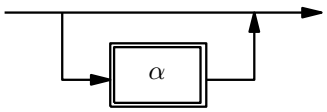


の形をしている時、構文図 $\boxed{\alpha}$ の先頭に現れる可能性のあるトークンを $c_1, \ldots, c_n$ とすると、この構文図に対応するプログラムは、

```
while (token == 'c1' || token == 'c2' || ... || token == 'cn') {
    [ α ]
}
```

のようにします。

6. 構文図が



の形をしている場合、構文図 $\boxed{\alpha}$ の先頭に現れる可能性のあるトークンが $c_1, \ldots c_n$ であるとすると、この構文図に対応するプログラムは、

```
if (token == 'c1' || token == 'c2' || ...|| token == 'cn') {
    [ α ]
}
```

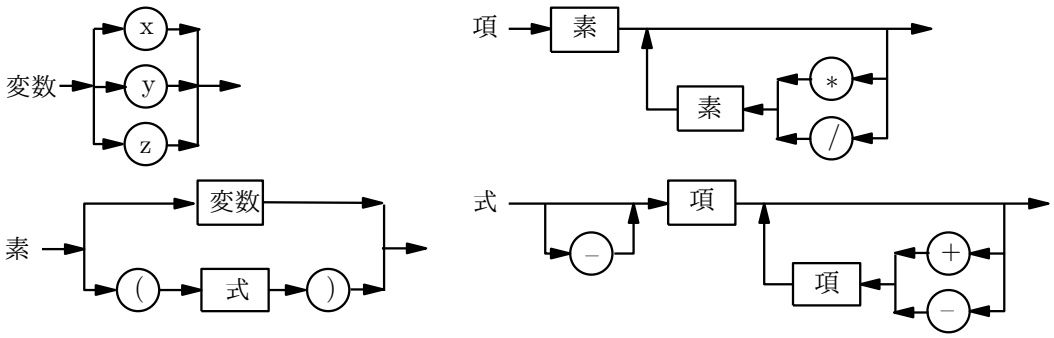
とします。

以上のような方法で、機械的にパーザのプログラムを作成することができます。プログラムの実行の流れが、構文図の中の流れに対応していることに注意してください。構文図の中の分岐は、パーザのプログラムに現れている `if` 文や `switch` 文、`while` 文 での分岐に対応することになります。このプログラムの実行の分岐がうまく働くためには、構文図が決定的であることが必要です。

メモ

4.3 パーザの例

前節に示した手順に従うと、例えば



というような構文図に対して次のような関数群を、ほとんど機械的に生成することができます。

```
 1 extern void Hensuu(void);
 2 extern void So(void);
 3 extern void Kou(void);
 4 extern void Shiki(void);
 5
 6 void Hensuu(void)
 7 {
 8     switch (token) {
 9     case 'x':
10         if (token == 'x')
11             gettoken();
12         else
13             error();
14         break;
15     case 'y':
16         if (token == 'y')
17             gettoken();
18         else
19             error();
20         break;
21     case 'z':
22         if (token == 'z')
23             gettoken();
24         else
25             error();
26         break;
27     default:
28         error();
29     }
30 }
31
32 void So(void)
33 {
34     switch (token) {
35     case 'x': case 'y': case 'z':
36         Hensuu();
37         break;
38     case '(':
39         if (token == '(')
40             gettoken();
```

```
 41         else
 42             error();
 43         Shiki();
 44         if (token == ')')
 45             gettoken();
 46         else
 47             error();
 48         break;
 49     default:
 50         error();
 51     }
 52 }
 53
 54 void Kou(void)
 55 {
 56     So();
 57     while (token == '*' || token == '/') {
 58         switch (token) {
 59         case '*':
 60             if (token == '*')
 61                 gettoken();
 62             else
 63                 error();
 64             break;
 65         case '/':
 66             if (token == '/')
 67                 gettoken();
 68             else
 69                 error();
 70             break;
 71         default:
 72             error();
 73         }
 74         So();
 75     }
 76 }
 77
 78 void Shiki(void)
 79 {
 80     if (token == '-') {
 81         if (token == '-')
 82             gettoken();
 83         else
 84             error();
 85     }
 86     Kou();
 87     while (token == '+' || token == '-') {
 88         switch (token) {
 89         case '+':
 90             if (token == '+')
 91                 gettoken();
 92             else
 93                 error();
 94             break;
 95         case '-':
 96             if (token == '-')
 97                 gettoken();
 98             else
 99                 error();
100             break;
```

```
101         default:
102             error();
103         }
104         Kou();
105     }
106 }
```

以上のように各関数を定義しておけば、たとえば非終端記号 ⟨式⟩ の構文解析を行いたい場合、この仕事を行う関数 parse として、次のようなものを用意すればよいことになります。

```
void parse(void)
{
    gettoken();
    Shiki();
    if (token != EOF)
        error();
}
```

構築されたパーザは、常にトークンを 1 つ先読みして、構文図の各部分に対応する関数を呼び出していることに注意して下さい。全体の構文解析を行う関数 parse は、読み残しのトークンが残っていないことを最後に確認して終了しています。

以上のような方針で行う構文解析を、**トップダウンの構文解析**と呼びます。トップダウンの構文解析は、入力を読み込みながら、構文木を上 (根) から下 (葉) に向かって認識していきます。これとは逆に、構文木を下から上に向って認識するような構文解析の方法を採用することも可能で、こちらを**ボトムアップの構文解析**と呼びます[1]。

メモ

[1] この授業では説明はしません。

4.4 演習問題

1. 今回パーザの例として挙げたプログラムには、冗長な処理をする部分が多く含まれています。このような部分をより簡潔で効率の良いプログラムとなるように書き換えなさい。

2. 前回の演習問題 3 で作った決定的な構文図に対して、関数 parse が、非終端記号 ⟨実数⟩ の文法チェックを行うような構文解析プログラムを構築しなさい。ただし、関数 gettoken() や error() は 1 ページのように定義されているものとして構いません。

3. BNF 記法で書かれた次のような文法について考えます。

$$
\begin{aligned}
\langle M \rangle &::= \text{“*”} \mid \text{“/”} . \\
\langle A \rangle &::= \text{“+”} \mid \text{“-”} . \\
\langle D \rangle &::= \text{“0”} \mid \text{“1”} \mid \text{“2”} \mid \text{“3”} \mid \text{“4”} \mid \text{“5”} \mid \text{“6”} \mid \text{“7”} \mid \text{“8”} \mid \text{“9”} . \\
\langle V \rangle &::= \langle D \rangle \{ \langle D \rangle \} [\text{“.”} \langle D \rangle \{ \langle D \rangle \}] . \\
\langle F \rangle &::= \langle V \rangle \mid \text{“(”} \langle E \rangle \text{“)”} . \\
\langle T \rangle &::= \langle F \rangle \{ \langle M \rangle \langle F \rangle \} . \\
\langle E \rangle &::= [\langle A \rangle] \langle T \rangle \{ \langle A \rangle \langle T \rangle \} . \\
\langle L \rangle &::= \{ \langle E \rangle \text{“=”} \} .
\end{aligned}
$$

   (1) 非終端記号 V、F、T、E、L の構文図をそれぞれ書きなさい。

   (2) 下の C プログラムは、標準入力 (キーボード) から、非終端記号 E の文法に従った数式に続いて = が入力される度に、その数式の計算結果を、次の実行例のように出力するプログラム calc.c の一部である。関数 V、F、T、E は、それぞれ非終端記号 V、F、T、E が表す集合に含まれる終端記号列を読み込み、その終端記号列を定数や数式と見なしたときの計算結果を返り値として戻すようにしている。関数 L も同様であるが、この関数は返り値を持たない。関数 F、T、E、L のプログラム (関数定義) を書いてこのプログラムを完成させなさい。ただし、これらの関数が文法エラーを発見した場合には、関数 error を呼ぶようにしなさい。

calc.c のコンパイルと実行例

```
$ cc -o calc calc.c
$ ./calc
1.23 =
1.230000
1.5+2.5*3 =
9.000000
1/(1+2) =
0.333333
(1.5+2.5)*(3/(1.2-2.5)) =
-9.230769
.
$
```

calc.c の一部

```
#include <stdio.h>
#include <stdlib.h>
```

```
extern double F(void);
extern double T(void);
extern double E(void);
extern void L(void);

int t;

void error(void)
{
    printf("Syntax error\n");
    exit(1);
}

void gettoken(void)
{
    do {
        t = getchar();
    } while (t == ' ' || t == '\t' || t == '\n');
}

double D(void)
{
    double v;

    if (t < '0' || '9' < t)
        error();
    v = (double)(t - '0');
    gettoken();
    return v;
}

double V(void)
{
    double v = 0.0, b = 1.0;

    do {
        v *= 10.0;
        v += D();
    } while ('0' <= t && t <= '9');
    if (t == '.') {
        gettoken();
        do {
            b /= 10.0;
            v += b*D();
        } while ('0' <= t && t <= '9');
    }
    return v;
}

double F(void)
{
    ⋮
}

double T(void)
{
    ⋮
}

double E(void)
```

```
{
    ⋮
}

void L(void)
{
    ⋮
}

int main(void)
{
    gettoken();
    L();
    return 0;
}
```

記号処理・第 4 回・終わり

付録：第 2 回演習問題の解答例

1. 導出できることを○で、できないことを×で表すと、次の表のようになる。

記号列	⟨変数⟩	⟨項⟩	⟨式⟩
(1) `y`	○	○	○
(2) `+ x`	×	×	×
(3) `y * z`	×	○	○
(4) `x + x`	×	×	○
(5) `x + y + z`	×	×	○
(6) `x * y * z`	×	○	○
(7) `x + y * z`	×	×	○
(8) `x * y + z`	×	×	○
(9) `( z )`	×	○	○
(10) `( y * z )`	×	○	○
(11) `x + ( y * z )`	×	×	○
(12) `( x + y ) * z`	×	○	○

(構文木は省略)

2. ⟨英文字⟩ ::= “a” | “b” | . . . | “z” | “A” | “B” | . . . | “Z” .
   ⟨数字⟩ ::= “0” | “1” | . . . | “9” .
   ⟨変数名⟩ ::= ⟨英文字⟩ { ⟨英文字⟩ | ⟨数字⟩ } .

3. ⟨正整数⟩ ::= ⟨非ゼロ数字⟩ { ⟨数字⟩ } .

4. ⟨整数⟩ ::= “0” | [“-”] ⟨非ゼロ数字⟩ { ⟨数字⟩ } .

5. ⟨実数⟩ ::= ⟨整数⟩ [⟨小数部⟩] | “-” “0” ⟨小数部⟩ .
   ⟨小数部⟩ ::= “.” { ⟨数字⟩ } ⟨非ゼロ数字⟩ .

6. ⟨式頭⟩ ::= ε | ⟨演算子⟩ .
   ⟨式尾⟩ ::= ε | ⟨演算子⟩ ⟨変数⟩ ⟨式尾⟩ .
   ⟨式⟩ ::= ⟨式頭⟩ ⟨変数⟩ ⟨式尾⟩ .

7. 例えば、終端記号列 000 は非終端記号 ⟨2進数⟩ に対して、以下の 2 つの構文木を持つ。