BNF の基本

BNF は言語の構文規則(syntax rule)を記述するための表記法です。この構文規則とは言語の大まかな「書き方」を定めるもので、一般的なプログラミング言語ではこの構文規則がかなり厳密に定められています。ここでは、この BNF の読み方や基本的な考え方についての解説を行います。

なお、このテキストで使っている表記法は BNF を少し変えたもので、オリジナルの BNF と異なります。

BNF の読み方

BNF は「記号 ::= 右辺」の形式で記述します。たとえば次のように書けます。

```
Expression ::= '+' '1'
```

ここでの BNF は、左辺に CamelCase で記号を書き、右辺には'～'で囲んだ文字列の並びを書くものとします。

これは「Expression は、まず'+'が来てその次に'1'が来る」と解釈します。そのため、ある言語が Expression から始まる場合、その言語は"+1"という文字列からなることになります。逆に、"-1"や"1+1"はこの言語で表せません。

ここで、上記の構文規則中に出現する Expression, '+', '1' をそれぞれ「記号」とよびます。また、Expression から"+1"という文字列をつくることを「Expression から"+1"を導出する(derive)」といい、逆に"+1"を Expression という記号に置き換えることを「"+1"を Expression に還元する(reduce)」といいます。

練習

上記の BNF を書き換えて、Expression が"1+2"という文字列を導出するようにせよ。ただし、記号には { '+', '1', '2', Expression } のみを利用すること。

選言

先ほどは 1 種類の文字列にしかならない構文規則を作りました。次のように書くことで 2 種類以上の文字列になる可能性がある構文規則を表現できます。

```
Expression ::= '+' '1'
             | '-' '1'
```

上記は「Expression は、'+', '1'が順に来る、または'-', '1'が順に来る」と解釈します。つまり、この Expression は"+1"と"-1"の文字列を導出する事ができます。このように、規則の右辺を「|」(選言, Alternation)で区切ることによって、左辺が導出できる文字列の可能性を複数書けます。

なお、一つの規則に対して複数の選言を書くこともできます。

```
Expression ::= '+' '1'
             | '-' '1'
             | '1' '+' '1'
             | '1' '-' '1'
```

練習

「Expression ::= '+' '1' | '-' '1'」を書き換えて、さらに文字列"1"も導出できるようにせよ。

終端記号と非終端記号

次に、少し複雑な規則を書いてみましょう。

```
Expression ::= Value '+' Value
     Value ::= NUMBER
             | '-' NUMBER
```

上記の BNF には Expression を左辺にもつ規則と、Value を左辺に持つ規則の 2 つが出現しています。これは次のように解釈します。

- Expression から、 Value, '+', Value を導出できる
- Value から、NUMBER、または'-', NUMBER を導出できる

ここで、NUMBER は任意の数字列とします。これまでは、Expression から文字列を直接導出することができました。しかし、規則の右辺内に他の規則の左辺記号が出現している場合、「右辺に出現する記号をさらに導出して、その文字列に置き換える」という動作が必要となります。つまり、"1+-2"という文字列は、図 1 のように Expression から導出されます。なお、このような構文解析の結果を木構造で表現した図を「構文木(Syntax Tree)」とよびます。

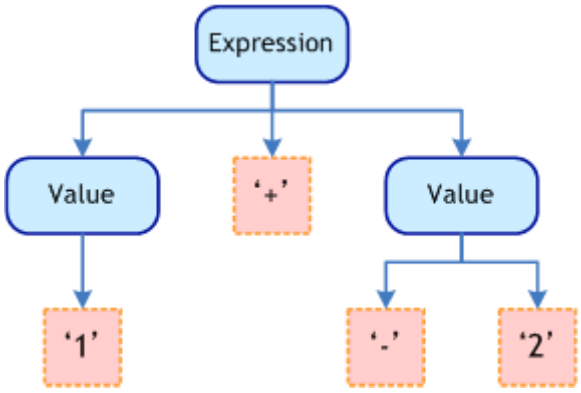

図 1 "1+-2"の構文木

なお、左辺に出現する記号を「非終端記号(non-terminal symbol)」、それ以外の記号を「終端記号(terminal-symbol)」とよびます。左辺は必ず 1 つの非終端記号のみからなり、右辺は終端記号や非終端記号の列からなります。上記の BNF では、非終端記号が{Expression, Value}で、終端記号が{'+', '-', NUMBER}でした。非終端記号は次々と導出していくことで最終的に非終端記号(文字)の列で表現することができ、言語全体で導出の入り口となる非終端記号を「開始記号」または「ゴール記号」とよびます。

ちなみに、このテキストでは、非終端記号を CamelCase で表し、終端記号を UPPER_CASE や'..'で表すものとします。このあたりが実際の BNF と異なるので、RFC などを読む際には気をつけてください。

練習

上記の BNF の各右辺から、非終端記号をすべて取り除け。ただし、Expression から導出可能な文字列は変えてはならない。

0 文字の文字列

BNF の中には「0 文字の文字列」が出現する場合があります。

```
Expression ::= 'f' '(' Argument ')'
  Argument ::= NUMBER
             |  empty
```

上記の Argument の右辺には、*empty* という表記が出現しています。これは「0 個の記号の列」を分かりやすく書いたもので「Argument は NUMBER を導出できる、またはなにも導出しない」というように解釈できます。つまり、この Expression は "f(10)"や"f()"などの文字列を導出できます。

なお、*empty* は書かなくても同じ意味となります。ですが「Argument ::= NUMBER | 」のように書くと見落としてしまいがちなので、このテキストでは明記するようにします。

練習

上記の BNF に出現するすべての非終端記号と終端記号をそれぞれ示せ。

上記の Expression から導出可能なすべての文字列を正規表現で示せ。ただし、NUMBER を表す正規表現は"0|[1-9][0-9]*"であるとする。

無限長の文字列

最後に、無限長の文字列を生成可能な例です。

```
Expression ::= '-' Expression
             |  'a'
```

この Expression は"a", "-a", "------a"などの文字列を導出可能で、'-'を無限に含めることができます。このように、非終端記号を導出する際に自分自身を出現させることで、いくらでも大きな言語を作成することができます。

練習

上記の Expression から導出可能なすべての文字列を正規表現で示せ。

上記の Expression から導出可能な文字列を変えずに、Expression に出現するすべての非終端記号を取り除くことは可能か？考察せよ。

練習問題

問題 1

```
String ::= 'a' String 'a'
         | 'b' String 'b'
         | 'c' String 'c'
         | 'a'
         | 'b'
         | 'c'
         | empty
```

構文規則が上記の BNF で表現され、開始記号が String である言語について、

(1) 上記の言語がとりうる文字列は、どのような特徴を持つか、答えよ。

(2) 上記の言語がとりうる文字列を、正規表現で記述する事は可能か、答えよ。

問題 2

「'a'^m 'b'^(m+n) 'c'^n」で表される言語がある。ただし、X^k は文字 X を k 回繰り返すことを表すものとする。また、m, n は 0 以上の任意の整数とする。

この言語について、

(1) この言語の構文規則を BNF で記述せよ。

(2) 上記の言語がとりうる文字列を、正規表現で記述する事は可能か、答えよ。

問題 3

```
List ::= List ',' 'a'
       | 'a'
```

構文規則が上記の BNF で表現され、開始記号が List である言語について、

(1) 上記の言語の構文規則を修正し、"" (空の文字列)を許すようにせよ。

(2) 上記の言語がとりうる文字列を、正規表現で記述する事は可能か、答えよ。

構文アクション付き BNF

BNF は言語の構文規則を表記するためのもので、「言語がとりうる文字列の種類」しか表現することができません。つまり、BNF でわかるのは「この文字列は対

象の言語に属するかどうか」ということだけでした。しかし、今回作成したいのは「構文解析器(Syntax Analyzer, Parser)」であり、「対象の言語に属するかどうか」ということよりも多くの情報を文字列から取り出す必要があります。

ここでは BNF を拡張し、「それぞれの構文規則に合致する文字列を見つけた際に、何らかの処理を行う」ということを記述できるようにしてみます。また、これらの処理を「構文アクション」とよぶことにします。

BNF と属性

構文アクションは、それぞれの構文規則の右側に";"を置き、その後に任意の Java の式を書けるものとします。つまり、BNF は次のようになります。

```
Expression ::= '+' '1' ; 1
             | '-' '1' ; -1
```

これは、「'+', '1'を Expression に還元する際に{ 1 }という値を割り当てる、または'-', '1'を Expression に還元する際に{ -1 }という値を割り当てる」と解釈します。つまり、Expression から文字列"+1"を導出した場合、割り当てられた値は Java の 1 という値になります。同様に、Expression から文字列"-1"を導出した場合、割り当てられた値は Java の-1 という値になります。

このように、文字列をある非終端記号に還元する(=非終端記号から文字列を導出する)際に、その非終端記号に何らかの値を割り当てています。この割り当てられた値をその非終端記号の「属性」とよび、特に還元の際に割り当てられる属性を「合成属性」とよびます。また、このように属性を利用する文法を「属性文法」とよびます。

合成属性の利用

構文アクションの中で、右辺に出現する非終端記号の合成属性を利用したい場合、次のように書くことにします。

```
Expression ::= Value$a '+' Value$b ; a + b
             | Value$a '-' Value$b ; a - b
     Value ::= '1'                 ; 1
             | '2'                 ; 2
```

上記は右辺に出現する記号に対して$とアルファベット 1 文字をつけ、右辺でそのアルファベットを利用して非終端記号の属性を参照しています。たとえば

Expression から"1+2"という文字列を導出する場合、「Value$a '+' Value$b」の a, b はそれぞれ 1, 2 という値を参照します。この場合、Expression の属性は{ 1 + 2 }というアクションが評価されて{ 3 }となります。同様に、Expression から"1-2"という文字列を導出した場合、その属性は{ -1 }となります。

練習

Expression から"2-1"という文字列を導出した場合、その Expression の属性を示せ。

上記は加算と減算の記号のみを取り扱えるが、さらに乗算記号'*'を扱えるように書き換えよ。

非終端記号の属性

また、終端記号の属性を参照した場合、次のような trait のインスタンスを利用可能であるとします。

```
trait Token {
    /** このトークンの種類 */
    val kind: TokenKind
    /** このトークンを構成する文字列 */
    val image: String

    // ...
}
```

つまり、終端記号の属性から文字列を取り出して利用する場合、次のように書くことができます。

```
Value ::= NUMBER$t ; t.image.toInt
```

上記は数値を表す終端記号から文字列を取り出し、それを Java の int 型の値に変換しています。

練習

終端記号 NUMBER は 0 以上の数値を表現する文字列である。上記の規則を書き換え、Value を導出した際に'-'から始まる負の値を取り扱えるようにせよ。また、その属性に文字列で表現された数値が与えられるように構文アクションを記述せよ。

抽象構文木

構文アクションには任意の Java の式を書けることにしたので、次のようなものも書けます。

```
Expression ::= Expression$a '+' Value$b ; new AddExpr(a, b)
             | Value$a                  ; a
    Value ::= NUMBER$t                  ; new ValueExpr(BigDecimal(t.image))
```

上記の Expression が"1"を導出する場合、その属性は{ new ValueExpr("1") }となります。同様に、"1+2"を導出した場合は{ new AddExpr(new ValueExpr("1"), new ValueExpr("2")) }となります。この値は Add オブジェクトを根にもつツリー構造を表現していますが、"1+2"を導出した際の構文木(図 2)よりも簡単な構造になっています(図 3)。これは構文木を簡単にしたものであるため、抽象構文木(Abstract Syntax Tree, AST)とよばれます。

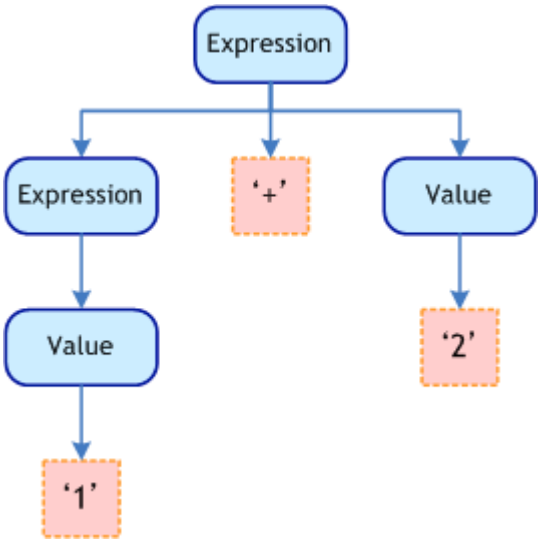

図 2 "1+2"の構文木

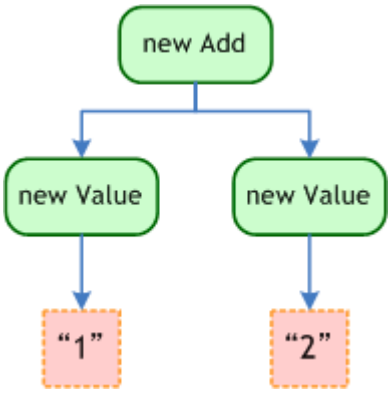

図 3 "1+2"の抽象構文木

多くの構文解析器は、テキストの構文を解析した結果として、上記のような抽象構文木を構築するように作られています。今回の授業でも、テキストを抽象構文木に変換する手法を中心に紹介していく予定です。

練習

Expression から"1+2+3"という文字列を導出した場合の、その Expression の属性示せ。

上記のアクション付き BNF は加算式を処理して AST に変換するためのものであった。これを書き換えて、減算"-"を取り扱えるようにせよ。ただし、減算記号は二項の左結合演算子とし、結合優先度は加算演算子と同等とする。また、AST 上での表現は{ new SubExpr(a, b) }とする。

練習問題

```
Expression ::= Expression '+' Term
             | Term
      Term ::= Term '*' Factor
             | Factor
    Factor ::= '-' Factor
             | Primary
   Primary ::= NUMBER
             | '(' Expression ')'
```

上記は加算、乗算、符号反転、式のグループ化を行える構文の構文規則を表現している。また、開始記号は Expression であるとする。

(1) 上記の BNF に構文アクションを付与し、入力された式の演算結果を計算するようにせよ。

(2) 上記の BNF に先ほどとは別の構文アクションを付与し、抽象構文木を生成するようにせよ。ただし、加算は{new AddExpr(a, b)}、乗算は{new MultiExpr(a, b)}、符号反転は{new MinusExpr(a)}、グループ化は{new ParenthesizedExpr(a)}、数値は{new ValueExpr(s)}で表現されるものとする。このとき、構文アクション中に出現する a, b は抽象構文木のノードとし、s は数値を表現する文字列とする。

(3) (2)で作成した構文アクション付き BNF を拡張し、減算{new SubExpr (a, b)}、除算{new DivExpr(a, b)}を処理できるようにせよ。ただし、減算、除算はいずれも左結合の二項演算で、減算の結合優先度は加算のそれと等しく、除算の結合優先度は乗算のそれと等しいものとする。

(4) 上記の BNF は、「Expression ::= Expression '+' Term」のように右辺の最初と左辺で同じ記号を使っている(左再帰とよぶ)ものがある。これを、「Expression ::= Term '+' Expression」のように右辺の最後に同じ記号が来るように書き換えた場合にどのような問題が起こるか、考察せよ。