構文解析表の作成

講義でも少し触れましたが、「各選言で先頭に出現する可能性がある終端記号の集合」のことを DIRECTOR 集合とよびます。DIRECTOR は direction(方向)を決定するという意味を持っており、LL(k)構文解析器が非終端記号を解析する際に、そのうちどの選言を利用するかを決めるためにこの DIRECTOR 集合を利用します。構文解析表もこの DIRECTOR 集合を元に作成しますが、講義では時間の関係でこの詳しい計算方法を紹介していませんでしたので、ここでは DIRECTOR 集合を機械的に計算する方法を紹介します。

DIRECTOR 集合はふつう、FIRST 集合と FOLLOW 集合とよばれるものを利用して計算します。前者は「ある記号列から導出可能な終端記号列の先頭(first)に出現する可能性がある終端記号列」を表し、後者は「ある非終端記号に後続(follow)する可能性がある終端記号列」を表します。

以降では実際の算出のためのアルゴリズムを紹介します。なお、アルゴリズムは擬似コードで解説しますが、その中では、記号を Vocab 型[1]、記号の列を List<Vocab>型、記号の集合を Set<Vocab>型で表現します。また、List<Vocab>型の値は list[0]で記号列の先頭を表現し、list[1..]で記号列の先頭を除いた残りの記号列を表現します。Set<Vocab>型では、+で 2 つの集合の和集合(左の集合と右の集合の要素をすべて含む集合)を表現し、-で差集合(左の集合から右の集合に含まれる要素を取り除いた集合)を表現するものとします。また、[...], { ... }はそれぞれ List<Vocab>, Set<Vocab>を構築するリテラルで、指定された要素からなる List<Vocab>や Set<Vocab>型の値を構築するものとします。なお、Set<Vocab>には 0 個の記号列を表す EMPTY や、ファイル終端を表す EOF が含まれる場合があります[2]。

FIRST 集合の計算

FIRST 集合は、「ある記号列から導出可能な終端記号列の先頭(first)に出現する可能性がある終端記号列」を表しています。これを計算する関数を「First(記号列)」で表した場合、この関数は次のようなアルゴリズムで実現できます。

---

[1] Vocabulary(語彙)の略で、「文法内で利用可能な記号」の意味です。

[2] FIRST, FOLLOW, DIRECTOR 集合は、本来「記号の集合」ではなく「記号**列**の集合」を表します。ただし、LL(1)においてはこれらの記号列は最大 1 つまでとなるため、Set<List<Vocab>>で表現すべきところを Set<Vocab>で表現しています。そのときに出現する「0 個の記号列」をこのテキストでは EMPTY という疑似記号で表現していますが、本来は[ ] (空のリスト)で表現します。

```
First(List<Vocab> list):
  if (list が空):
    return { EMPTY }
  if (list の先頭が終端記号):
    return { list[0] }
  if (list の先頭が非終端記号):
    Set<Vocab> result = {}
    for (List<Vocab> alt in list[0]の各選言):
      if (First(alt)に EMPTY が含まれない):
        result += First(alt)
      else:
        result += First(alt) - { EMPTY } + First(list[1..])
    return result
```

上記のように、First の計算には First を再帰的に利用するため、手で計算するには収斂計算(繰り返して結果が収束するまで行う)が必要になります。

各ステップの詳しい説明は、例を元に行います。

```
Expr    ::=   Sign Value
Sign    ::=   '-'
            |   empty
Value   ::=   NUM
            |   '(' Expr ')'
```

ここでは、各非終端記号の FIRST 集合を計算してみましょう。つまり、First([Expr]), First([Sign]), First([Value])を計算することになります。このとき、**BNF の後の方に出現するものから順に計算する**ことがポイントです。

FIRST 集合 step1: 終端記号

まず、First([Value])を計算します。これは、その右辺にあるそれぞれの選言に対する FIRST 集合の和を計算することになります。つまり、

```
First([Value])
 -> First([NUM]) + First(['(', Expr ')'])
```

ということになります。この First([NUM]), First(['(', ...])の最初の記号はいずれも終端記号ですので、先ほどの式は次のように簡約されます。

```
First([Value])
  -> { NUM } + { '(' } = { NUM, '(' }
```

この値は後で利用しますので、それぞれの非終端記号に対する FIRST 集合の表を作ってまとめておきましょう。

非終端記号	確定	FIRST 集合
Expr	NO	{ }
Sign	NO	{ }
Value	**YES**	**{ NUM, '(' }**

FIRST 集合 step2: empty

次に、First([Sign])を計算します。この右辺には *empty* が出現しますが、これは「一つも記号がない」ことを表していますので、疑似コードでは [] (空の記号列)で表現することになります。このため、First([Sign])は次の手順で計算します。

```
First([Sign])
 -> First(['-']) + First([])
```

このとき、First([])の結果に EMPTY が出現しますので、それを取り除いたあとに[Sign]の 1 つ目の記号を読み飛ばしたリスト([])に対する First を計算します。つまり、簡約の手順は下記のようになります。

```
First([Sign])
 -> First(['-']) + First([])
 -> { '-' } + ( { EMPTY } - { EMPTY } + First([]) )
 -> { '-' } + ( {} + First([]) )
 -> { '-' } + ( {} + { EMPTY } )
 = { '-', EMPTY }
```

となります。この結果も後で使いますので、先ほど作成した FIRST 集合の表に追加しておきます。

非終端記号	確定	FIRST 集合
Expr	NO	{ }
Sign	**YES**	**{ '-', EMPTY }**
Value	YES	{ NUM, '(' }

FIRST 集合 step3: 非終端記号

次に、First([Expr])を計算します。

```
First([Expr])
 -> First([Sign, Value])
```

今回は、記号列の先頭に非終端記号が出現しました。アルゴリズムの手順に従うと、これは First([Sign])の結果を利用します。この結果は作成中の FIRST 集合の表に含まれていますので、それを利用します。

```
First([Expr])
 -> First([Sign, Value])
 -> { '-', EMPTY } ...
```

このとき、結果の集合に EMPTY が出現したので、それを取り除いて[Sign, Value]の 1 つ目を取り除いたリストに対して再帰的な計算を行います。

```
First([Expr])
 -> First([Sign, Value]) ...
 -> { '-', EMPTY } - { EMPTY } + First([Value])
 -> { '-' } + First([Value])
```

同様に、すでに計算した Value の FIRST 集合を利用します。

```
First([Expr])
 -> First([Sign, Value]) ...
 -> { '-', EMPTY } - { EMPTY } + First([Value])
 -> { '-' } + First([Value])
 -> { '-' } + { NUM, '(' }
 = { '-', NUM, '(' }
```

結果として上記のようになります。これも FIRST 集合の表に追加しておきましょう。

非終端記号	確定	FIRST 集合
Expr	**YES**	**{ '-', NUM, '(' }**
Sign	YES	{ '-', EMPTY }
Value	YES	{ NUM, '(' }

気づいた方もいるかと思いますが、EMPTY が出現する場合の処理「result += First(alt) - { EMPTY } + First(list[1..])」は、講義でも紹介した「*empty* が来たら読み飛ばす」という操作を表したものです。この First(alt)は選言に含まれる非終端記号から導出した記号列ですので、その先頭が *empty* になるということは、この記号列全体が *empty* になることと同じです。この場合、その記号列は無視して、その非終端記号の次の記号(list[1..])に対してさらに FIRST 集合を計算するという操作が必要になります。

FIRST 集合 step4: 収斂

FIRST 集合の表をみて、すべての項目が確定していれば計算は終わりです。今回は計算中に FIRST 集合の表を利用しましたが、いずれも計算がすでに確定していたものだけを利用していました。これは、「BNF の後の方に出現するものから順に計算する」と

いう方法を採ったためで、たとえば逆の順番で計算を行ったりすると、First([Expr])の計算に未確定の First([Sign])の結果が必要となり、First([Expr])の結果も未確定となります。

ここまでで FIRST 集合の表に未確定の項目が残っていれば、その表を利用してもう一度最初から計算を行います。そして、すべての項目が確定するか、変化がなくなるまで計算したら、FIRST 集合の表が「収斂した」ことになり、計算は完了です(再帰がある場合には確定しませんので、変化がなくなるまで計算します)。

以上で FIRST 集合の計算手順の紹介は終了です。なお、今回の FIRST 集合では「先頭に来る可能性のある終端記号 **1 つ分**」を計算しています。このため、FIRST 集合に含まれる終端記号列は、常に単一の終端記号か、EMPTY(0 個の終端記号列)で表現されています。このような FIRST 集合を、特に **FIRST1 集合**とよぶことがあります。

練習(FIRST 集合)

下記の BNF の各非終端記号に対し、FIRST 集合の表を作成せよ。

```
Expr    ::=   Sign Value
Sign    ::=   '+'
            |   '-'
            |   empty
Value   ::=   NUM
            |   ID
```

FOLLOW 集合の計算

FOLLOW 集合は、「ある非終端記号に後続(follow)する可能性がある終端記号列」を表しています。これを計算する関数を「Follow(非終端記号)」で表した場合、この関数は次のようなアルゴリズムで実現できます。

```
Follow(Vocab vnt):
  Set<Vocab> result = {}
  if (vnt が開始記号):
    result += { EOF }
  for (List<Vocab> right in vnt に後続する記号列):
    if (First(right)に EMPTY が含まれない):
      result += First(right)
    else:
      result += First(right) - { EMPTY } + Follow(right の左辺記号)
  return result
```

FIRST 集合の計算と同様で、Follow の計算でも自己再帰呼出しが出現していますので、こちらも収斂計算が必要になります。また、Follow の計算には First の結果を利用しますので、先に FIRST 集合の表を作っておくのがよいでしょう。

これも、例を紹介しながら解説して行きます。

```
      Expr   ::=   Value ExprRest
  ExprRest   ::=   '-' Value
              |    empty
     Value   ::=   NUM
              |    '(' Expr ')'
```

これに対する FIRST 集合の表は、次のようになります。

非終端記号	FIRST 集合
Expr	{ NUM, '(' }
ExprRest	{ '-', EMPTY }
Value	{ NUM, '(' }

ここでは、各非終端記号の FOLLOW 集合を計算してみましょう。つまり、Follow(Expr), Follow(ExprRest), Follow(Value)を計算することになります。計算順序は First のときとは逆で、**BNF の前の方に出現するものから順に計算する**ことがポイントです。

FOLLOW 集合 step1: FIRST 集合の利用

最初に、Follow(Expr)を計算します。最初にやることは、「Expr が右辺に出現する箇所を探す」ということです。見つけたら、「左辺記号 => 選言の内容」の表記方法でリストアップしておきます。

- Value => '(' **Expr** ')'

この「vnt(= Expr)に後続する記号列」は[')']です。また、Expr は開始記号としますので、初期の集合は { EOF } になり、次のように進めて行きます。

```
Follow(Expr)
 -> { EOF } + First([')'])
 -> { EOF } + { ')' }
 = { ')', EOF }
```

これで Follow(Expr)が確定しましたので、FIRST 集合のときと同様に FOLLOW 集合の表も作りましょう。

非終端記号	確定	FIRST 集合	FOLLOW 集合
Expr	**YES**	**{ NUM, '(' }**	**{ ')', EOF }**

非終端記号	確定	FIRST 集合	FOLLOW 集合
ExprRest	NO	{ '-', EMPTY }	{ }
Value	NO	{ NUM, '(' }	{ }

FOLLOW 集合 step2: 左辺の利用

次に、順番通り Follow(ExprRest)を計算します。これも同様に「ExprRest が右辺に出現する箇所」をリストアップしてみましょう。

- Expr => Value **ExprRest**

上記は ExprRest が記号列の右端にあるため、そのさらに右側には一つも記号列がありません。この場合、右側に来る記号列は空、つまり **[]** として表現します。つまり、次のように計算して行きます。

```
Follow(ExprRest)
 -> {} + First([]) ...
```

このとき、First([])には EMPTY が出現しますので、EMPTY を取り除いたあとに「Follow(左辺記号)」を加えてやります。今回は「Expr => Value ExprRest」について計算していましたので、左辺記号は Expr となります。

```
Follow(ExprRest)
 -> {} + ( First([]) - { EMPTY } + Follow(Expr) )
 -> {} + ( { EMPTY } - { EMPTY } + Follow(Expr) )
 -> {} + ( {} + Follow(Expr) )
```

この Follow(Expr)は先ほど作成し始めた FOLLOW 集合の表で計算が確定していますので、その値に置き換えます。

```
Follow(ExprRest)
 -> {} + ( First([]) - { EMPTY } + Follow(Expr) )
 -> {} + ( { EMPTY } - { EMPTY } + Follow(Expr) )
 -> {} + ( {} + Follow(Expr) )
 -> {} + ( {} + { ')', EOF } )
 = { ')', EOF }
```

これで Follow(ExprRest)も確定しましたので、FOLLOW 集合の表に書き加えてやります。

非終端記号	確定	FIRST 集合	FOLLOW 集合
Expr	YES	{ NUM, '(' }	{ ')', EOF }
ExprRest	**YES**	**{ '-', EMPTY }**	**{ ')', EOF }**
Value	NO	{ NUM, '(' }	{ }

FOLLOW 集合 step3: empty

次に、Follow(Value)を計算します。これも同様に出現箇所をリストアップするところから始めます。

- Expr => **Value** ExprRest
- ExprRest => '-' **Value**

少し説明を省略すると、2 つ目は Value が右端に出現しているため、先ほどの Follow(ExprRest)と似たような計算になります。これは左辺記号が ExprRest であるため、次のように計算できます。

```
-> First([]) - { EMPTY } + Follow(ExprRest)
-> { EMPTY } - { EMPTY } + { ')', EOF }
-> { ')', EOF }
```

残るは 1 つ目の「Expr => Value ExprRest」の計算ですが、これは Value の右側に ExprRest が出現しているため、次のように計算します。

```
-> First([ExprRest]) ...
```

FIRST 集合の表を参照すると、上記の式は{ '-', EMPTY }となり EMPTY が出現します。そのため、EMPTY を取り除いて「Expr => Value ExprRest」の左辺記号である Expr の FOLLOW 集合を加えてやります。つまり、次のようになります。

```
-> First([ExprRest]) - { EMPTY } + Follow(Expr)
-> { '-' } + Follow(Expr)
```

Follow(Expr)はすでに計算してありますので、その結果に置換します。

```
-> First([ExprRest]) - { EMPTY } + Follow(Expr)
-> { '-' } + Follow(Expr)
-> { '-' } + { ')', EOF }
-> { '-', ')', EOF }
```

これでそれぞれの選言に対する計算が終わりましたので、これらの和集合が Follow(Value)の計算結果になります。

```
Follow(Value)
 = { '-', ')', EOF } + { ')', EOF }
 = { '-', ')', EOF }
```

この結果も FOLLOW 集合の表に追加しておきましょう。

非終端記号	確定	FIRST 集合	FOLLOW 集合
Expr	YES	{ NUM, '(' }	{ ')', EOF }
ExprRest	YES	{ '-', EMPTY }	{ ')', EOF }
Value	**YES**	**{ NUM, '(' }**	**{ '-', ')', EOF }**

FOLLOW 集合 step4: 収斂

最後に、FIRST 集合の際と同様に、FOLLOW 集合の表に対しても収斂を行います。収斂の終了条件は FIRST 集合のそれと同様で、「すべての項目が確定する」か「表が変化しなくなる」までです。今回はすべての項目が確定していますので、この時点で FOLLOW 集合の計算は終了です。なお、今回は BNF の前から順に計算を行いましたが、逆順でやると収斂までに時間がかかります。

なお、ここで計算した FOLLOW 集合は、先ほどの FIRST 集合と同様に「後続する可能性のある終端記号 **1** つ分」を計算しています。このため、このような FOLLOW 集合は特に **FOLLOW1** 集合とよぶことがあります。

練習(FOLLOW 集合)

下記の BNF の各非終端記号に対し、FOLLOW 集合の表を作成せよ。

```
      Expr   ::=   Value ExprRest
  ExprRest   ::=   '+' Value
              |    '-' Value
              |    empty
     Value   ::=   NUM
              |    '(' Expr ')'
```

DIRECTOR 集合の計算

非常に前置きが長くなりましたが、やっと当初の目的である DIRECTOR 集合の算出方法です。この DIRECTOR 集合は「各選言で先頭に出現する可能性がある終端記号の集合」を表現しますが、これは先ほどまでに算出した FIRST, FOLLOW 集合を元に計算できます。この Director 集合は選言に対して計算するため、「Director(選言の左辺記号, 選言の右辺記号列)」で表し、この関数は次のようなアルゴリズムで実現できます。

```
Director(Vocab lhs, List<Vocab> rhs):
  Set<Vocab> set = First(rhs)
  if (First(rhs)が EMPTY を含まない):
    return First(rhs)
  else:
    return First(rhs) - { EMPTY } + Follow(lhs)
```

今回は再帰呼出しが出現せず、First, Follow だけを利用して計算ができます。なお、疑似コード中に出現する lhs, rhs はそれぞれ Left Hand Side(左辺)、Right Hand Side(右辺)の略です。日常的に利用する略記なので、気づかず使っていたら脳内で置き換えてください。

この DIRECTOR 集合の計算も、例を紹介しながら解説して行きます。

```
      Expr   ::=   Value ExprRest
  ExprRest   ::=   '-' Value
              |    empty
     Value   ::=   NUM
              |    '(' Expr ')'
```

これに対する FIRST, FOLLOW 集合の表は、次のようになります。

非終端記号	FIRST 集合	FOLLOW 集合
Expr	{ NUM, '(' }	{ ')', EOF }
ExprRest	{ '-', EMPTY }	{ ')', EOF }
Value	{ NUM, '(' }	{ '-', ')', EOF }

DIRECTOR 集合 step1: First(右辺)

計算の対象となる選言は、下記の５個です。

- Expr => Value ExprRest
- ExprRest => '-' Value
- ExprRest => *empty*
- Value => NUM
- Value => '(' Expr ')'

まず、右辺記号列の FIRST 集合を計算するところから始めます。

```
First([Value, ExprRest]) = { NUM, '(' }
First(['-', Value]) = { '-' }
First([]) = { EMPTY }
First([NUM]) = { NUM }
First(['(', Expr, ')']) = { '(' }
```

この操作は、FIRST 集合を元に「各選言の最初に出現する終端記号」を計算しようとしています。ただし、「ExprRest => *empty*」のように結果に EMPTY が出現する場合がありますので、これだけでは計算が終わりません。それ以外の選言では、この右辺記号列の FIRST 集合がそのまま DIRECTOR 集合となります。

```
Director(Expr, [Value, ExprRest])
 -> First([Value, ExprRest])
 = { NUM, '(' }
Director(ExprRest, ['-', Value])
 -> First(['-', Value])
 = { '-' }
Director(Value, [NUM])
 -> First([NUM])
 = { NUM }
Director(Value, ['(', Expr, ')'])
 -> First(['(', Expr, ')'])
 = { '(' }
```

DIRECTOR 集合 step2: Follow(左辺)

「ExprRest => *empty*」では右辺記号列の FIRST 集合に EMPTY が出現しましたので、アルゴリズムの手順通りに EMPTY を除去して左辺記号の FOLLOW 集合を追加してやります。

```
Director(ExprRest, [])
 -> First([]) - { EMPTY } + Follow(ExprRest)
```

今度は FOLLOW 集合の表を元に、Follow(ExprRest)を置換してやります。

```
Director(ExprRest, [])
 -> First([]) - { EMPTY } + Follow(ExprRest)
 -> { EMPTY } - { EMPTY } + { ')', EOF }
 = { ')', EOF }
```

なお、講義で紹介した「*empty* が来たら読み飛ばして、導出元の次の終端記号を探す」(p43-p47)というのは「FIRST 集合に EMPTY が来たらそれを除去して、左辺の FOLLOW 集合を追加する」ということをおおざっぱに説明したものでした。講義では直接 *empty* が来る場合のみを考慮していましたが、今回の手順では右辺から結果的に *empty* が導出されたとしても、同じ手順で DIRECTOR 集合を算出することができます。

DIRECTOR 集合 step3: DIRECTOR 集合の表

最後に、これまでに計算した DIRECTOR 集合を表にしてみましょう。DIRECTOR 集合は選言ごとに計算するため、元の BNF の右側に列挙してやります。

```
      Expr   ::=  Value ExprRest              { NUM, '(' }
  ExprRest   ::=  '-' Value                   { '-' }
```

```
            |    empty                          { ')', EOF }
    Value   ::=  NUM                              { NUM }
            |    '(' Expr ')'                     { '(' }
```

講義では力技で上記の表を算出していましたが、実は舞台裏ではこのような計算を行っていました。これまでに紹介したように、アルゴリズムを元に機械的に算出でき、パーサ・ジェネレータなどはこの操作を忠実に再現して、DIRECTOR 集合を算出しています。DIRECTOR 集合が算出できれば、そこから構文解析表も作成でき、LL(1)文法については機械的に構文解析器を生成することができます。

なお、この表は FIRST1 集合と FOLLOW1 集合を元に計算し、「各選言で先頭に出現する可能性がある終端記号１つ分の集合」を計算していました。これを特に **DIRECTOR1** 集合とよぶことがあり、DIRECTOR1 集合を元に作成された構文解析器を LL(1)構文解析器とよびます。LL(**k**)構文解析器では、これまで 1 文字だったすべての計算を **k** 文字に拡張しなければならず、文法の規模を n とした場合にこれらの計算量が $O(n^k)$程度に膨れ上がります。

練習(DIRECTOR 集合)

下記の BNF(FOLLOW 集合の練習と同様)に対し、DIRECTOR 集合の表を作成せよ。

```
      Expr    ::=   Value ExprRest
  ExprRest    ::=   '+' Value
              |     '-' Value
              |     empty
     Value    ::=   NUM
              |     '(' Expr ')'
      Expr    ::=   Sign Value
```

LL(1)構文解析の技法

LL(1)文法でない代表的な例として、次の 2 つを挙げました。

- 選言の左端で、同じ記号列が出現する
- 選言の左端で、左辺と同じ非終端記号が出現する

これらに対して、講義では次の技法を紹介しました。

- 左括り出し
- 左再帰除去

この章では、上記の技法の詳しい紹介と、それに関連する技法を紹介します。

左括り出し

講義では左括り出しの意義と簡単な適用例を紹介しましたが、一般的な適用手順については省略していました。ここでは、左括り出しの一般的なやり方について紹介します。

まず、左括り出しの対象となる BNF は「同一の非終端記号に対する左辺の各選言で、先頭に同一の記号列が出現する」という場合です。この「非終端記号」を Vnt とおき[3]、「同一の記号列」を<left-common>とおいた場合、BNF は次のような形式になります。

```
Vnt   ::=   <left-common> ...
        |   <left-common> ...
        |   <left-common> ...
        |   ...
```

これではなにもわからないので、各<left-common>の右側を別々の記号列へと書き直します。それぞれ<right1>, <right2>, <right3>とします。

```
Vnt   ::=   <left-common> <right1>
        |   <left-common> <right2>
        |   <left-common> <right3>
        |   ...
```

左括り出しでは、先頭に共通する記号列を見つけた場合、その右側をまとめて別の非終端記号 VntRest に括り出してやります[4]。

---

[3] Vnt は Vocabulary of Non-Terminal のつもりで書いています。

[4] 慣例では括り出した非終端記号は、元の非終端記号の末尾に"Rest"や"Tail"を続けたものにすることが多いため、ここでは"Vnt"に対して"VntRest"という名前の非終端記号を導入しています。

```
        Vnt    ::=    <left-common> VntRest
                |     ...
```

この VntRest は<right1>, <right2>, <right3>を括り出したものなので、それらを選言でつないでやります。すると、BNF 全体は次のようになります。

```
        Vnt    ::=    <left-common> VntRest
                |     ...
    VntRest    ::=    <right1>
                |     <right2>
                |     <right3>
```

なお、<right*>は 0 個以上の記号列を表していますので、0 個の場合は *empty* で表現します。一方、<left-common>が 0 個の記号列では、VntRest を括り出してもあまり意味がないため、通常は 1 個以上の記号列に対して左括り出しを適用しましょう。

練習(左括り出し)

次の BNF を変換し、LL(1)文法にせよ。

```
       Expr    ::=    Value
                |     '-' Value
                |     '-' Value Value
      Value    ::=    NUM
```

左再帰除去

左括り出しと同様に、左再帰除去についても一般的な変換方法を紹介しておきます。左再帰除去は名前の通り「左再帰」を含む BNF を対象としていますが、これは「ある右辺の左端に、左辺と同じ非終端記号が出現する」という場合です。この「非終端記号」を Vnt とおいた場合、BNF は次のような形式になります。

```
        Vnt    ::=    Vnt ...
                |     ...
```

これも先ほど同様、もう少し記号列を増やしてみます。

```
        Vnt    ::=    Vnt <recursive-rest>
                |     <no-recursive1>
                |     <no-recursive2>
                |     ...
```

これは Moore の左再帰除去を利用すると、次のように変形できます。

```
        Vnt    ::=    <no-recursive1> VntRest
                |     <no-recursive2> VntRest
```

```
                |     ...
     VntRest    ::=   empty
                |     <recursive-rest> VntRest
```

このように、選言の左端で自己再帰を行っているものを、「左再帰」とよびます。これは左再帰除去を適用することで、右端で自己再帰を行う「右再帰」の形に変換することができます。なお、左再帰の中でも自己再帰を行っているものを「直接左再帰」とよびますが、それよりも深い位置でループする「間接左再帰」とよばれるケースもあります。

```
       First    ::=   Second ...
                |     ...
      Second    ::=   First ...
                |     ...
```

これが出現するケースはあまりありません。もしこのケースに遭遇した場合にも機械的に除去する方法はありますが、ここでは紹介しません。

また、左再帰が同一の非終端記号の右辺に 2 つ以上出現する場合、先に左括り出しを行っておきます。たとえば、

```
        Expr    ::=   Expr '+' NUM
                |     Expr '-' NUM
                |     NUM
```

という BNF は、Expr の各選言で Expr が共通して左端に出現しますので、次のように書き換えます。

```
        Expr    ::=   Expr ExprRest
                |     NUM
    ExprRest    ::=   '+' NUM
                |     '-' NUM
```

これの Expr に対し、さらに左再帰除去を適用します。

```
        Expr    ::=   NUM ExprRest2
   ExprRest2    ::=   empty
                |     ExprRest ExprRest2
    ExprRest    ::=   '+' NUM
                |     '-' NUM
```

これでは多少読みにくいので、ExprRest2 に出現する ExprRest を展開してみましょう。

```
        Expr    ::=   NUM ExprRest2
   ExprRest2    ::=   empty
```

```
                |     '+' NUM ExprRest2
                |     '-' NUM ExprRest2
```

この操作はコツがつかめたら「左括り出し→左再帰除去→左括り出しの展開」という一連の作業を一括して適用してもかまいません。

練習(左再帰除去)

次の BNF を変換し、LL(1)文法にせよ。

```
        Expr    ::=    Expr '+' NUM
                |      Expr '+' ID
                |      NUM
                |      ID
```

末尾再帰除去

講義でも多少触れましたが、コンパイラの最適化の一つに「末尾再帰除去」というものがあります。これは、次のような関数に対して適用します。

```
f(a):
  ...
  return f(x)
  ...
```

このような「return で自己再帰した結果を返す関数」は、次のように書き換えることで再帰呼出しを除去することができます。

```
f(a):
  while (true):
    ...
    a = x
    continue
    ...
```

自己再帰は「引数の値を書き換えて、もう一度自分自身を実行する」ということを行いますので、全体をループで括って、その仮引数の値を再帰呼出しの実引数に変えたのち、全体を括ったループを繰り返して実行してやれば、元の再帰呼出しと同じ動作をするプログラムになります。

通常の手続き型言語を利用している場合、このような最適化を適用できるケースはあまりありません[5]。しかし、今回のように LL(k)パーサを作成している場合などの(奇特な)ケースでは、よくこのようなパターンに遭遇します。たとえば、左再帰除去を適用したあとの BNF は次のような形式になります。

```
          Add   ::=   Value$a AddRest(a)$r              ; r
  AddRest(a)    ::=   empty                             ; a
                 |    '+' Value$b AddRest(a+b)$r        ; r
```

この AddRest を素直にプログラム化すると、次のような疑似コードになります。

```
AddRest(a):
  if (next.kind == EOF):
    return a
  else if (next.kind == '+'):
    consume('+')
    var b = Value()
    return AddRest(a + b)
  else
    raise error
```

これには「return AddRest(...)」という再帰呼出しがあります。今回、末尾再帰の「x」の部分は「a + b」となっていますので、これを考慮して末尾再帰除去を適用してみると次のようになります。

```
AddRest(a):
  while (true):
    if (next.kind == EOF):
      return a
    else if (next.kind == '+'):
      consume('+')
      var b = Value()
      a = a + b
      continue
    else:
      raise error
```

---

[5] 関数型言語に代表される繰り返しの概念が弱い言語では、この末尾再帰除去の考え方は一般的です。多くの関数型言語の処理系(Emacs Lisp をのぞく)では、この末尾再帰除去の最適化が含まれています。

つまり、末尾再帰のパターンでは、

- 関数全体を while(true)で囲む
- 「return+再帰呼出し」の形式があれば、再帰呼出しの実引数を仮引数に代入する
- 代入後に continue で while(true)の先頭に戻る

という書換えを行うことができます。また、2 つ以上の末尾再帰が存在する場合、末尾再帰除去を順次適用して行くと関数全体が何度も while(true)で囲まれることになりますが、while(true)は 2 つ以上あっても 1 つの場合と同じ意味なので、1 つに省略できます。

なお、BNF での話に戻りますと、LL(k)構文解析を行う際には左再帰を除去しなければならないというのが前回の講義の内容でした。そして、Moore の左再帰除去を適用した場合、右側で再帰を行う「右再帰」という形式になります。この右再帰を含む BNF から講義で紹介した方法で LL(1)パーサを作成すると、プログラムは必ず末尾再帰を含む関数が作られます。この末尾再帰を含む関数は、末尾再帰除去を適用することによって、通常のループで表現することができ、プログラムは非常に効率の良いものとなります。

練習(末尾再帰除去)

次の構文アクション付き BNF に対応する LL(1)パーサのプログラムを疑似コードで記述せよ。ただし、再帰呼出しを含んではならない。

```
       Expr  ::=  Value$a ExprRest(a)$r       ; r
ExprRest(a)  ::=  empty                       ; a
              |   '+' Expr$b ExprRest(a+b)$r  ; r
              |   '-' Expr$b ExprRest(a-b)$r  ; r
      Value  ::=  NUM$t                       ; Integer.parseInt(t.image)
```