

Parser勉強会(1)

2010/08/25

あらかわ (@ashigeru)

Table of Contents

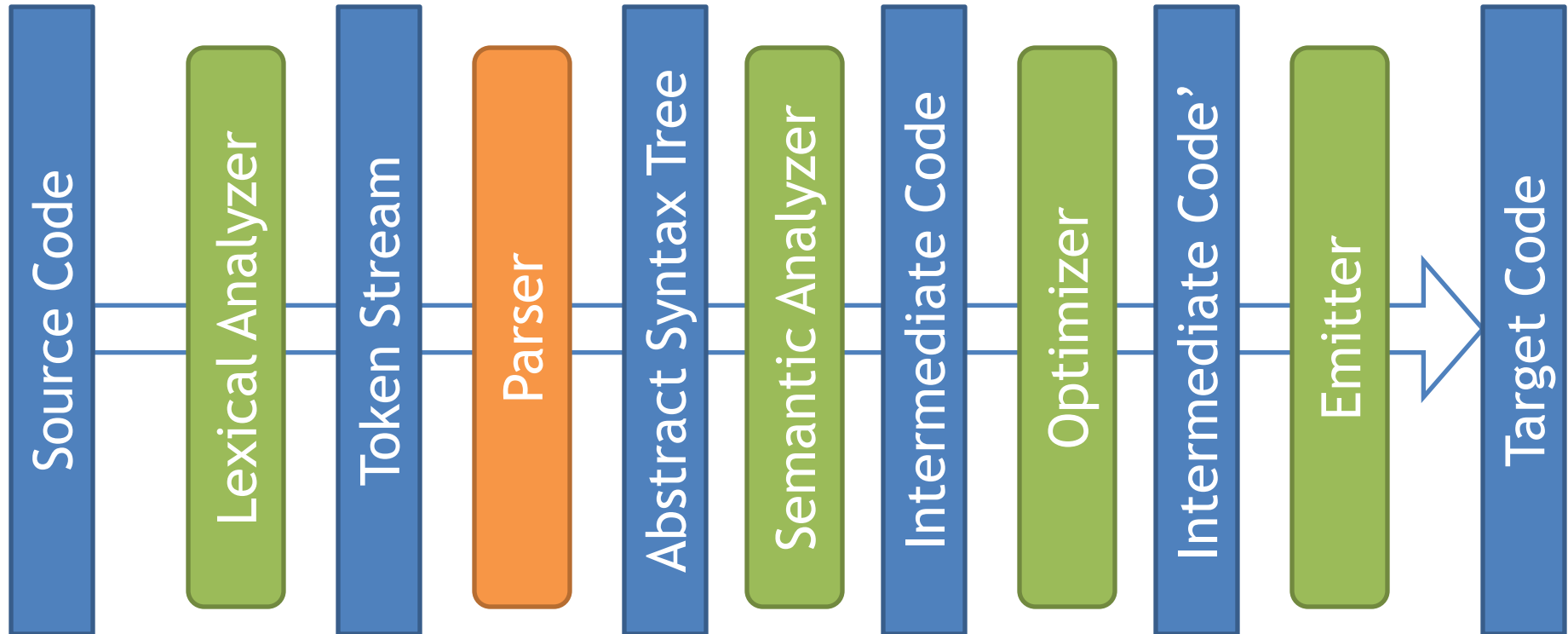
- Parserとは何か
- LL(1) Parserの作り方
- BNFのLL(1)化
- パーサ・ジェネレータの機能

Parserとはなにか

- 構文解析器(Syntax Analyzer)
 - 文字列の構文を解析して、構造化する
- 次の部分として利用
 - コンパイラ
 - トランスレータ
 - インタープリタ

コンパイラの構造

- Parserはコンパイラのほんの一部



新しい言語

- 新しい言語を作る問題
 - 習得コストが掛かる
 - 保守コストが掛かる
 - XMLと同程度の表現力しかない
 - プロダクトレベルに達するまでが非常に遠い
 - 理解可能なエラーメッセージ
 - エディタの提供
 - デバッガの提供
- パーサを1時間で書ける言語で、上記をそろえる
とだいたい1週間くらい必要

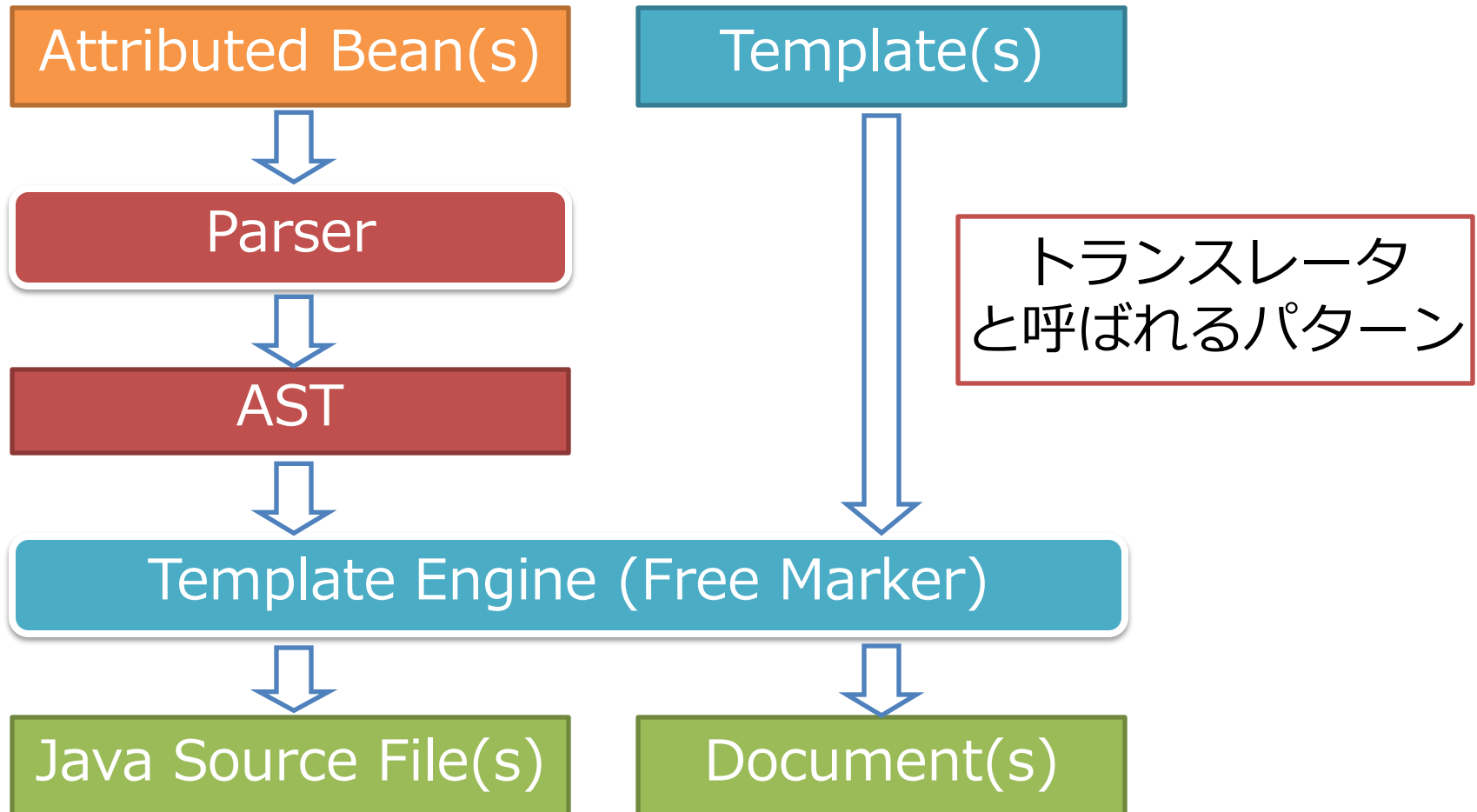
Attributed Bean (1)

```
[Doc="{@code if}文"]  
IfStatement extends Statement {  
    [Doc="条件式"]  
    [NotNull]  
    condition : Expression;  
    [Doc="条件成立時に実行される文"]  
    [NotNull]  
    thenStatement : Statement;  
    [Doc="条件不成立時に実行される文"]  
    [CanNull="この文が{@code if-then}文である場合"]  
    elseStatement : Statement;  
}
```

Attributed Bean (2)

- 自動生成されるもの
 - Beanのインターフェース
 - Beanの実装
 - Beanのメタクラス(type-safe reflection)
 - (Abstract) Factory Patternのサポート
 - Visitor Patternのサポート
 - 比較器系
 - ドキュメンテーション

Attributed Bean (3)



Attributed Bean (4)

- コード生成効率
 - 記述したDSL : 9k
 - 生成したJava : 66k
- 利点
 - 凝縮したDSLで、Java側のエラーはほぼない
 - Java側の横断的な変更もコストが極小に
- 欠点
 - 秀丸でがんばった
 - ほかの人が利用するのはたぶん無理

どうやって作るか

- ツールの利用
 - パーサ・ジェネレータ
 - JavaCC, ANTLR, SableCC, CUP, ...
 - パーサ・コンビネータ
 - Scala, JParsec, ...
- ツールがやってくれること
 - 字句解析器の生成
 - 構文解析器の生成
 - (抽象構文木モデルの生成)

パーサ・ジェネレータ/JavaCC

- 入力
 - 字句規則
 - 構文規則
- 出力
 - 字句解析と構文解析を行うJavaのプログラム
- 今回はパーサ・ジェネレータの気分になってプログラムを書いてみる

まずはBNFをプログラムに変換する方法

LL(1) PARSERの作り方

LL(1)パーサの作り方

- 説明が難しいので、少しずつ進めていく
 - 最後まで行かないとLL(1)という用語の説明すらできない

終端記号 (1)

Expr ::= '+' NUM

- このパーサのプログラムは？
 - 正しい順番でトークンが来るかチェックするだけのパーサ

終端記号 (2)

```
Expr ::= '+' NUM
```

```
Expr():  
    if (next token is '+')  
        consume next token  
    else  
        raise error  
  
    if (next token is NUM)  
        consume next token  
    else  
        raise error
```

省略記法

- 以後、終端記号の消費は次のメソッドを利用

```
consume(SOMETHING):  
    if (next token is SOMETHING)  
        return (consume next token)  
    else  
        raise error
```


終端記号 (3)

Expr():

consume(' + ')

consume(NUM)

- 終端記号の列を解析するパーサ
 - 終端記号の出現順にトークンを消費するだけ

構文アクション (1)

```
Expr ::= '+' NUM$t ; Integer.parseInt(t.image)
```

- このパーサのプログラムは？

構文アクション (2)

```
Expr ::= '+' NUM$t ; Integer.parseInt(t.image)
```

```
Expr():
```

```
    consume('+')
```

```
    var t = consume(NUM)
```

```
    return Integer.parseInt(t.image)
```

構文アクション (2.1)

```
Expr ::= '+' NUM$t ; Integer.parseInt(t.image)
```

```
Expr():  
    consume('+')  
    var t = consume(NUM)  
    return Integer.parseInt(t.image)
```

合成属性を変数へ

構文アクション (2.2)

```
Expr ::= '+' NUM$t ; Integer.parseInt(t.image)
```

```
Expr():  
    consume('+')  
    var t = consume(NUM)  
    return Integer.parseInt(t.image)
```

returnでアクションを書く

構文アクション (3)

Expr():

```
consume('+')
```

```
var t = consume(NUM)
```

```
return Integer.parseInt(t.image)
```

- 構文アクションを含むパーサ
 - 合成属性を変数に保存
 - returnで構文アクションを実行

選言 (1)

```
Expr ::= '+' NUM$t ; + Integer.parseInt(t.image)
      | '-' NUM$t ; - Integer.parseInt(t.image)
```

- このパーサのプログラムは？

選言 (2)

```
Expr ::= '+' NUM$t ; + Integer.parseInt(t.image)
      | '-' NUM$t ; - Integer.parseInt(t.image)
```

- 最初の記号に注目する
 - '+' ならば？
 - '-' ならば？
 - それ以外ならば？

選言 (3)

```
Expr ::= '+' NUM$t ; + Integer.parseInt(t.image)
      | '-' NUM$t ; - Integer.parseInt(t.image)
```

```
Expr():
```

```
    if (next token is '+')
```

```
        consume('+')
```

```
        var t = consume(NUM)
```

```
        return + Integer.parseInt(t.image)
```

```
    else if (next token is '-')
```

```
        consume('-')
```

```
        var t = consume(NUM)
```

```
        return - Integer.parseInt(t.image)
```

```
    else
```

```
        raise error
```

選言 (3.1)

```
Expr ::= '+' NUM$t ; + Integer.parseInt(t.image)
       | '-' NUM$t ; - Integer.parseInt(t.image)
```

Expr():

if (next token is '+')

consume('+')

var t = consume(NUM)

return + Integer.parseInt(t.image)

else if (next token is '-')

consume('-')

var t = consume(NUM)

return - Integer.parseInt(t.image)

else

raise error

最初の記号で分岐

選言 (3.2)

```
Expr ::= '+' NUM$t ; + Integer.parseInt(t.image)
       | '-' NUM$t ; - Integer.parseInt(t.image)
```

```
Expr():
```

```
    if (next token is '+')
```

```
        consume('+')
```

```
        var t = consume(NUM)
```

```
        return + Integer.parseInt(t.image)
```

```
    else if (next token is '-')
```

```
        co
```

```
        v
```

```
        r
```

```
    else
```

```
        raise error
```

ここだけ注目すると
'+' NUM\$t ; ...

選言 (3.3)

```
Expr ::= '+' NUM$t ; + Integer.parseInt(t.image)
      | '-' NUM$t ; - Integer.parseInt(t.image)
```

Expr():

ここだけ注目すると

'-' NUM\$t ; ...

else (next token is '-')

```
consume('-')
var t = consume(NUM)
return - Integer.parseInt(t.image)
```

else
raise error

選言 (4)

```
Expr ::= '+' NUM$t ; Integer.parseInt(t.image)
```

```
Expr():
```

```
    if (next token is '+')  
        consume('+')  
        var t = consume(NUM)  
        return Integer.parseInt(t.image)  
    else raise error
```

- 選言がないのもこの略記
 - 同じプログラムになるはず

選言 (5)

Expr():

if (next token is '+') ...

else if (next token is '-') ...

else raise error

- 選言を含むパーサ
 - 最初の記号で分岐するだけ
 - 分岐先ではこれまでの書き方と同様
 - エラーも忘れずに

非終端記号 (1)

```
Expr ::= '-' value$v ; -v
```

```
value ::= NUM$t ; Integer.parseInt(t.image)
```

- このパーサのプログラムは？

非終端記号 (2)

```
Expr ::= '-' value$v ; -v
```

```
value ::= NUM$t ; Integer.parseInt(t.image)
```

- 先にValueだけ考えてみる
 - 終端記号だけなら簡単

非終端記号 (3)

```
Expr ::= '-' value$v ; -v
```

```
value ::= NUM$t ; Integer.parseInt(t.image)
```

```
value():
```

```
    if (next token is NUM)
        var t = consume(NUM)
        return Integer.parseInt(t.image)
    else
        raise error
```

非終端記号 (4)

```
Expr ::= '-' value$v ; -v
```

```
value ::= NUM$t ; Integer.parseInt(t.image)
```

```
Expr():
```

```
    if (next token is '-')
```

```
        consume('-')
```

```
        var v = ???
```

```
        return -v
```

```
    else
```

```
        raise error
```

ここには何を入れる？

非終端記号 (4)

```
Expr ::= ' - ' value$ v ; -v
```

```
value ::= NUM$ t ; Integer.parseInt(t.image)
```

```
Expr():
```

```
    if (next token is ' - ')
```

```
        consume(' - ')
```

```
        var v = ???
```

```
        return -v
```

```
    else
```

```
        raise error
```

ここには何を入れる？
→Valueと同じものを
導出できる何か

非終端記号 (5)

```
Expr ::= '-' value$ v ; -v
```

```
value ::= NUM$ t ; Integer.parseInt(t.image)
```

```
Expr():
```

```
    if (next token is '-')  
        consume('-')  
        var v = value()  
        return -v  
    else  
        raise error
```

先程作ったValue()が
まさにその機能を持つ

```
value():
```

```
    if (next token is NUM)  
        var t = consume(NUM)  
        return Integer.parseInt(t.image)  
    else  
        raise error
```

非終端記号 (6)

```
Expr():  
    if (next token is '-')  
        consume('-')  
        var v = value()  
        return -v  
    else  
        raise error
```

- 右辺に非終端記号を含むパーサ
 - 非終端記号を解析するメソッドを実行するだけ

ひとやすみ (1)

- ここまでの知識で解ける課題
 - Q0Parser
- おさらい
 - 終端記号はトークンの消費
 - 構文アクションはreturn文
 - 合成属性は変数代入
 - 選言は最初の終端記号で分岐
 - 非終端記号はメソッド呼び出し

非終端記号+選言 (1)

```
Expr ::= value  
      | '-' value  
value ::= NUM  
      | 'a'
```

- このパーサのプログラムは？

非終端記号+選言 (2)

```
Expr ::= value  
      | '-' value
```

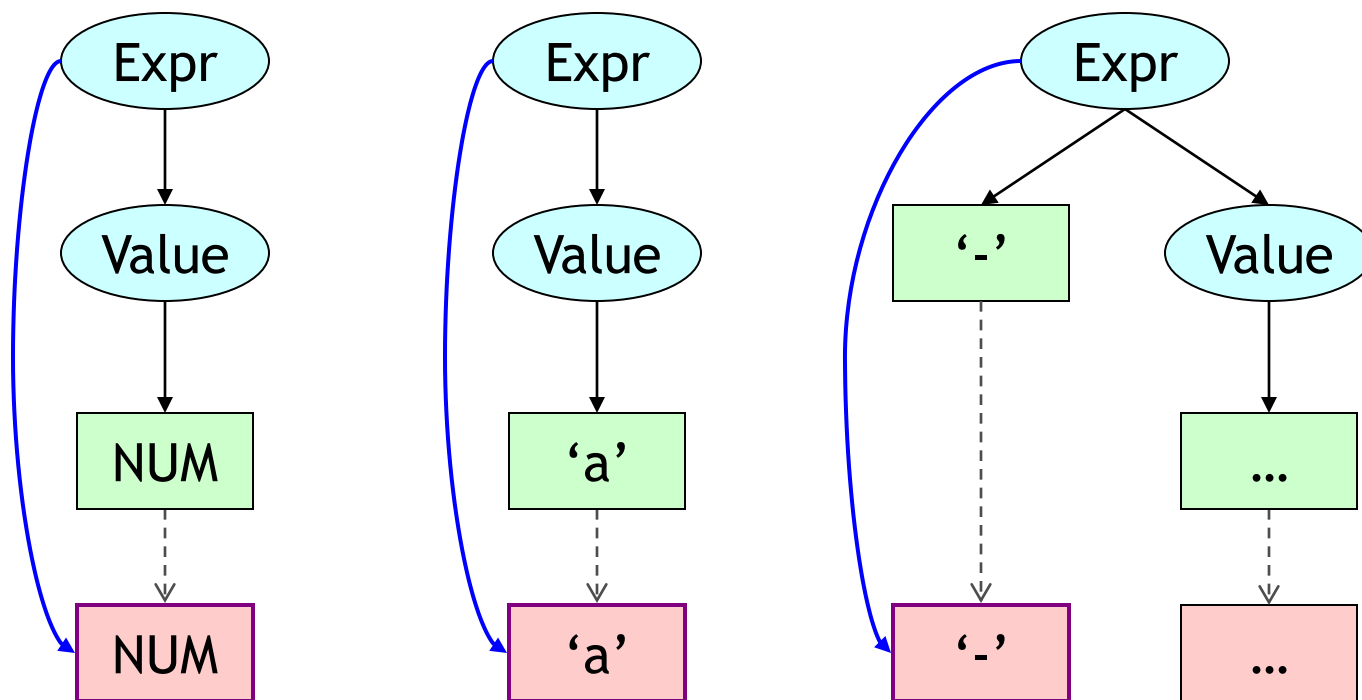
```
value ::= NUM  
       | 'a'
```

```
Expr():  
    if (next token is value) ...  
    else if (next token is '-') ...  
    else raise error
```

- 最初の記号に注目する
 - Valueは非終端記号なのでうまくいかない
 - では、最初の終端記号はどうなる？

非終端記号+選言 (3)

```
Expr ::= value  
      | '-' value  
  
value ::= NUM  
      | 'a'
```



- 導出した先頭に出現する終端記号を探せばいい
 - 非終端記号があれば再帰的に導出

非終端記号+選言 (4)

Expr ::= value
 | '-' value
value ::= NUM
 | 'a'

```
Expr():  
    if (next token is NUM | 'a') ...  
    else if (next token is '-') ...  
    else raise error
```

- 終端記号にぶつかるまで探していく
 - 結果として、次の終端記号が2種類以上になる場合も

非終端記号+選言 (5)

```
Expr ::= value  
      | '-' value
```

```
value ::= NUM  
       | 'a'
```

```
Expr():  
    if (next token is NUM | 'a') ...  
    else if (next token is '-') ...  
    else raise error
```

- 選言の先頭に非終端記号を含むパーサ
 - 非終端記号を辿って先頭の終端記号を探す

練習 (1)

```
Expr ::= ' - ' Expr  
      |  Prim
```

```
Prim  ::= '( Expr )'  
      |  value  
      |  STRING
```

```
value ::= NUM
```

- それぞれの選言で最初に来る可能性がある終端記号は？

練習の解答 (1.1)

Expr	::=	'-' Expr	{ '-' }
		Prim	?
Prim	::=	'(' Expr ')'	{ '(' }
		value	?
		STRING	{ STRING }
value	::=	NUM	{ NUM }

- まずは明らかなものを埋めて行く
 - 先頭に終端記号が来たら何も考えない

練習の解答 (1.2)

Expr	::=	'-' Expr	{ '-' }
		Prim	?
Prim	::=	(' Expr ')	{ '(' }
		value	{ NUM }
		STRING	{ STRING }
value	::=	NUM	{ NUM }

- 非終端記号のルールを展開
 - 下からやると楽

練習の解答 (1.3)

Expr	::=	'-' Expr	{ '-' }
		Prim	{ '(', NUM, STRING }
Prim	::=	'(' Expr ')'	{ '(' }
		value	{ NUM }
		STRING	{ STRING }
value	::=	NUM	{ NUM }

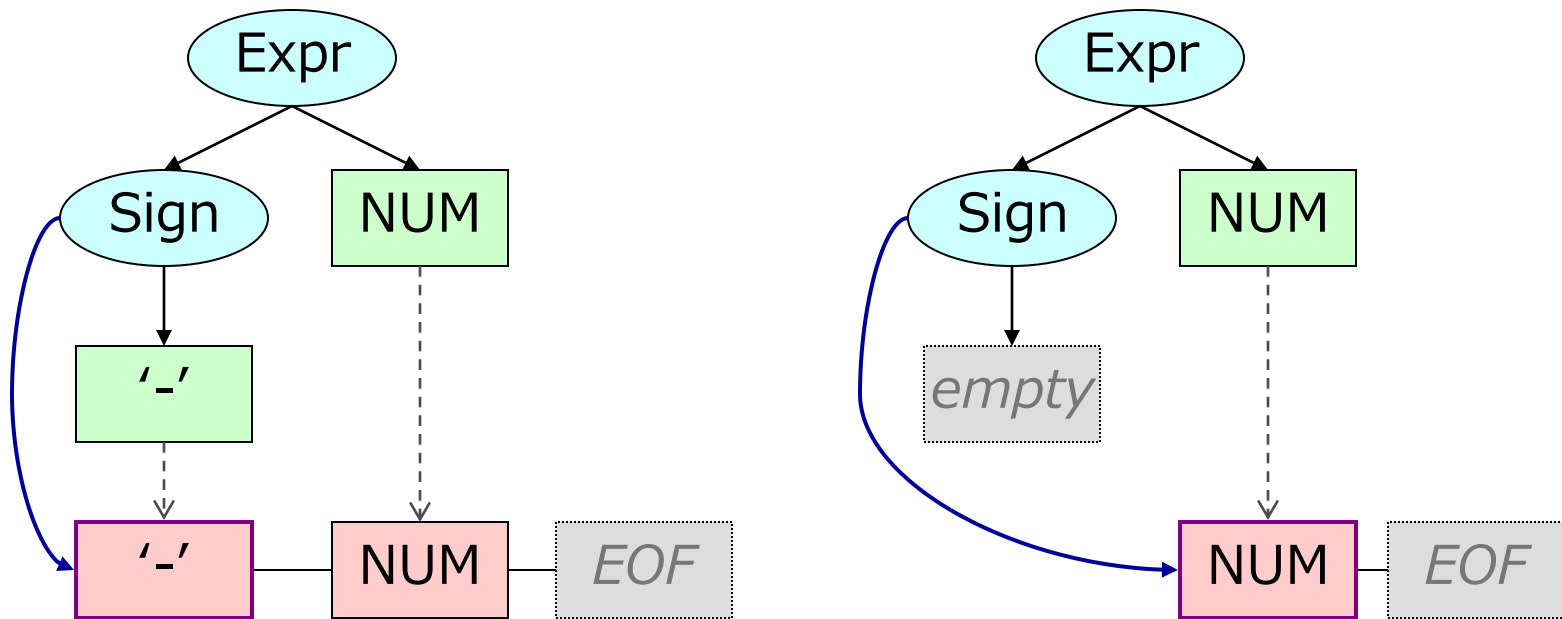
- 非終端記号に宣言が含まれたら、集合の和を使う

Empty (1)

```
Expr ::= Sign NUM  
Sign ::= '-'  
       | empty
```

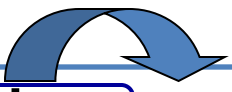
- それぞれの選言で先頭に来る可能性がある終端記号は？
- なお、*empty*は「なにもない」を表現するので、先頭に来ることはありえない
– /* *empty* */

Empty (2)



- *empty*は読み飛ばしてNUMがくる
 - *empty*でもちゃんと次の非終端記号がある
 - 最後の最後にはEOFというダミーの終端記号がいる

Empty (3)



```
Expr ::= Sign NUM  
Sign ::= '-'  
       | empty
```

- *empty*という記号は存在しない
 - *empty*と書いてある場所の「先頭の終端記号」は、Signの直後にくる可能性のある終端記号になる
- 上記の場合、さらにNUMもなければ、Exprの次に来る終端記号(=EOF)となる
 - いつまでたっても*empty*は「次の終端記号」にはならない

Empty (4)

```
Expr ::= Sign NUM  
Sign ::= '-'  
       | empty
```

Signがなかったことになる場合、その次はNUM

```
sign():  
    if (next token is '-')  
        consume('-')  
        return  
  
    else if (next token is NUM)  
        /* empty */  
        return  
  
    else  
        raise error
```

Empty (5)

```
Expr ::= Sign NUM  
Sign ::= '-'  
      | empty
```

```
Sign():  
  if (next token is '-') ...  
  else if (next token is NUM) ...  
  else ...
```

- *empty*を含むパーサ
 - 見なかったことにして、導出元の次の終端記号を探す

練習 (2)

Expr ::= 'f' '(' Args ')'

Args ::= List
| *empty*

List ::= STRING
| NUM

- それぞれの選言で最初に来る可能性がある終端記号は？

練習の解答 (2.1)

Expr ::= 'f' '(' Args ')'	{ 'f' }
Args ::= List	?
<i>empty</i>	?
List ::= STRING	{ STRING }
NUM	{ NUM }

- 先ほど同様に、わかるところを埋める

練習の解答 (2.2)

Expr ::= 'f' '(' Args ')'	{ 'f' }
Args ::= List	{ NUM, STRING }
<i>empty</i>	?
List ::= STRING	{ STRING }
NUM	{ NUM }

- 非終端記号への伝搬も同様

練習の解答 (2.3)

Expr ::= 'f' '(' Args ')'	{ 'f' }
Args ::= List	{ NUM, STRING }
<i>empty</i>	{ ')'
List ::= STRING	{ STRING }
NUM	{ NUM }

- *empty*は呼び出し元の次の記号
 - Argsの右側の記号

構文解析表 (1)

- それぞれの選言の開始直後に来る可能性
がある終端記号を列挙してみる
 - 練習と同じ

Expr ::= Sign NUM	{ '−', NUM }
Sign ::= '−'	{ '−' }
<i>empty</i>	{ NUM }

構文解析表 (2)

- この集合で選言を自動的に判別可能

Expr ::= Sign NUM	{ '−', NUM }
Sign ::= '−' NUM	{ '−' }
empty	{ NUM }

```
Expr():  
  if (next is '−' | NUM)  
    Sign()  
    consume(NUM)  
  else  
    raise error
```

```
Sign():  
  if (next is '−')  
    consume('−')  
    consume(NUM)  
  else if (next is NUM)  
    return  
  else  
    raise error
```

構文解析表 (3)

- 表形式にすると、構文解析表になる
 - 非終端記号\終端記号 でまとめる
 - 各カラムの内容は選ばれる選言の内容
 - 非終端記号の導出開始時に、次に来る終端記号を表現

```
Expr ::= Sign NUM { ' - ', NUM }  
Sign ::= ' - ' NUM { ' - ' }  
      | empty      { NUM }
```

	' - '	NUM
Expr	Sign NUM	Sign NUM
Sign	' - ' NUM	<i>empty</i>

構文解析表 (4)

- このような表をLL(1)構文解析表とよぶ
 - 入力を左(Left)から読み、選言の左端(Left-most)の点で次の1文字を先読みしている
- さらに、LL(1)構文解析表を元に作成されたパーサをLL(1)パーサと呼ぶ

	' _ '	NUM
Expr	Sign NUM	Sign NUM
Sign	' _ ' NUM	<i>empty</i>

練習 (3)

Expr ::= 'f' '(' List ')'

List ::= NUM Rest

Rest ::= ',' NUM Rest
| *empty*

- この構文解析表は？
 - ちょっと考えないと難しいかも
- 先頭の終端記号を探して右端にたどり着いたら、左辺の非終端記号の右から再開する
 - Rest → List → Expr となる箇所がある

練習の解答 (3.1)

Expr ::= 'f' '(' List ')'	{ 'f' }
List ::= NUM Rest	{ NUM }
Rest ::= ',' NUM Rest <i>empty</i>	{ ',' } ?

- ともあれ、わかるところは先に

練習の解答 (3.2)

Expr ::= 'f' '(' List ')'	{ 'f' }
List ::= NUM Rest	{ NUM }
Rest ::= ',' NUM Rest <i>empty</i>	{ ',' } ?

- まずは呼び出し元を探す
 - が、呼び出し元の右側に何も無い

練習の解答 (3.3)

Expr ::= 'f' '(' List ')'	{ 'f' }
List ::= NUM Rest	{ NUM }
Rest ::= ',' NUM Rest	{ ',' }
<i>empty</i>	{ ')' }

- さらにその呼び出し元を探す

ひとやすみ (2)

- ここまでの知識で解ける課題
 - Q1Parser
- おさらい
 - 非終端記号が先頭にあったら展開して、先頭に来る終端記号を探す
 - *empty*は読み飛ばしてその次に来る終端記号を探す
 - 非終端記号\終端記号で表を作ると、選言を自動的に判別可能

ここまでの方法でうまくいかないパターン

BNFのLL(1)化

LL(1)? (1)

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

- 上記の構文解析表は？
 - Q3Parserを簡単にしたBNF

LL(1)? (2)

```
Expr ::= value '+' value { NUM }  
      | value '-' value { NUM }  
      | value             { NUM }
```

```
Expr():  
    if (next token is NUM) ...  
    else if (next token is NUM) ...  
    else if (next token is NUM) ...  
    else raise error
```

- どう見てもLL(1)じゃない
 - 左端が同じだと、LL(1)とはならない

左括り出し (1)

```
Expr ::= Value Rest { NUM }  
Rest ::= '+' Value { '+' }  
       | '-' Value { '-' }  
       | empty      { EOF }
```

- 左が同じなら、まとめて括り出す
 - みんな大好きRefactoring (正しくはfactoring)
- 「左括り出し (Left-Factoring)」とよぶ
 - LL(1)では標準的な技法
 - 多少読みにくくなるのが難点

左括り出し (2)

- 構文アクションはどうしよう
 - 明らかにいろいろ足りない

```
Expr ::= value$a '+' value$b ; a + b
      | value$a '-' value$b ; a - b
      | value$a ; a
```

```
Expr ::= value$a Rest ; ?
Rest ::= '+' value$b ; a + b
      | '-' value$b ; a - b
      | empty ; a
```

左括り出し (3.1)

- まず、ExprでのaをRestで使っている
 - Restに渡してやればいい？

Expr ::= Value	a	Rest		;	?
Rest ::=	'+'	Value	b	;	$a + b$
	'-'	Value	b	;	$a - b$
		<i>empty</i>		;	a

左括り出し (3.2)

- 渡してみた
 - プログラムはどうなる？

```
Expr ::= Value$a Rest(a) ; ?
Rest(a) ::= '+' Value$b ; a + b
          | '-' Value$b ; a - b
          | empty ; a
```


左括り出し (4)

```
Expr ::= Value$a Rest(a) ; ?  
Rest(a) ::= '+' Value$b ; a + b  
           | '-' Value$b ; a - b  
           | empty ; a
```

```
Expr():  
  var a = Value()  
  Rest(a)  
  return ...
```

```
Rest(a):  
  if (next is '+')  
    ... return a + b  
  else if (next is '-')  
    ... return a - b  
  else if (next is EOF)  
    ... return a  
  else raise error
```

左括り出し (5.1)

- あとは、Restのアクション結果がExprのアクション結果として返ればいい

```
Expr ::= value$a '+' value$b ; a + b
      | value$a '-' value$b ; a - b
      | value$a ; a
```

```
Expr ::= value$a Rest(a) ; ?
Rest(a) ::= '+' value$b ; a + b
          | '-' value$b ; a - b
          | empty ; a
```

左括り出し (5.2)

```
Expr ::= Value$a Rest(a)$r ; r
Rest(a) ::= '+' Value$b ; a + b
           | '-' Value$b ; a - b
           | empty ; a
```

- Restの合成属性をExprの合成属性に渡す
- なお、(x)のような属性は継承属性とよぶ
 - パーサの合成属性 → プログラムの戻り値
 - パーサの継承属性 → プログラムの引数

左括り出し (6)

```
Expr():  
    var a = value()  
    var r = Rest(a)  
    return r
```

```
Rest(a):  
    if (next is '+')  
        ... return a + b  
    else if (next is '-')  
        ... return a - b  
    else if (next is EOF)  
        ... return a  
    else raise error
```

- 最初の記号が重複しているパーサ
 - 左括り出しで重複部分をまとめる
 - 属性が分断されたら引数で渡す(継承属性)

練習 (4)

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

- 左括り出しでLL(1)化できる？

練習の解答 (4.1)

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

- まずは括り出す部分を考える
 - その右側を非終端記号におく

練習の解答 (4.2)

```
Expr ::= '-' value Rest  
      | value  
  
Rest ::= empty  
      | value  
  
value ::= NUM
```

- 「何もない」という宣言は*empty*で書く

練習の解答 (4.3)

Expr ::= '−' value Rest	{ '−' }
value	{ NUM }
Rest ::= empty	{ EOF }
value	{ NUM }
value ::= NUM	{ NUM }

- LL(1)かどうか確かめる
 - LL(1)構文解析表を作ればいい
 - このルールではEOFが出現

練習の解答 (4.4)

Start ::= Expr EOF	...
Expr ::= '−' value Rest	{ '−' }
value	{ NUM }
Rest ::= <i>empty</i>	{ EOF }
value	{ NUM }
value ::= NUM	{ NUM }

- 暗黙のルールStartがあると考える
 - 開始記号を導出してそのあとにEOF
 - 予習課題でもそんな感じ

ひとやすみ (3)

- ここまでの知識で解ける課題
 - Q2Parser
 - Q3Parser
- おさらい
 - 選言の先頭で記号が重複していたら左括りだし
 - そのままではLL(1)にならない
 - 左括り出しで属性が分断されたら、継承属性で分断先に引き渡す
 - 継承属性はメソッドの引数で実現できる
 - 合成属性はメソッドの戻り値だった

LL(1)? (3)

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

- 上記の構文解析表は？
 - Q4Parserの構文アクションをはずしたBNF

LL(1)? (4)

Expr ::= Expr '+' value	{ NUM }
value	{ NUM }
value ::= NUM	{ NUM }

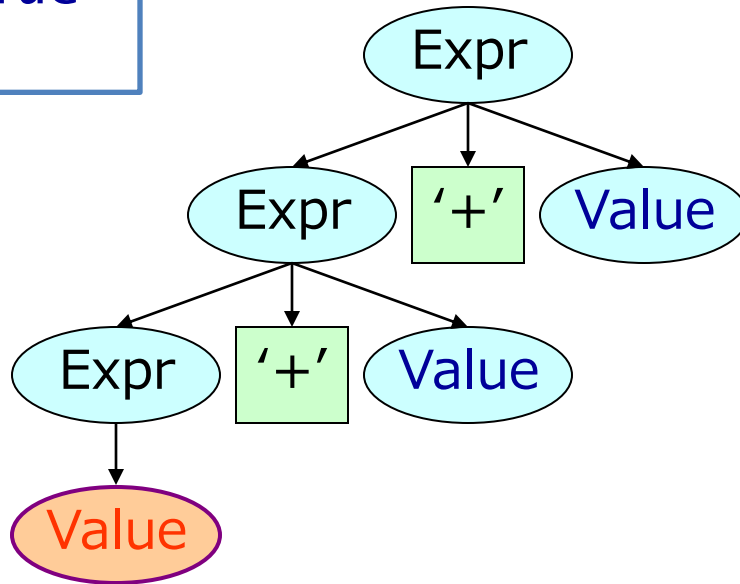
- ExprでNUMが衝突する
 - 左端で再帰していると、LL(1)にならない
 - LL(1)は左端で選言を選ぶため、このような左再帰形の構文はものすごく相性が悪い

左再帰除去 (1)

- 時間の関係でMooreの「左再帰除去」のテクニックをそのまま利用

左再帰除去 (2)

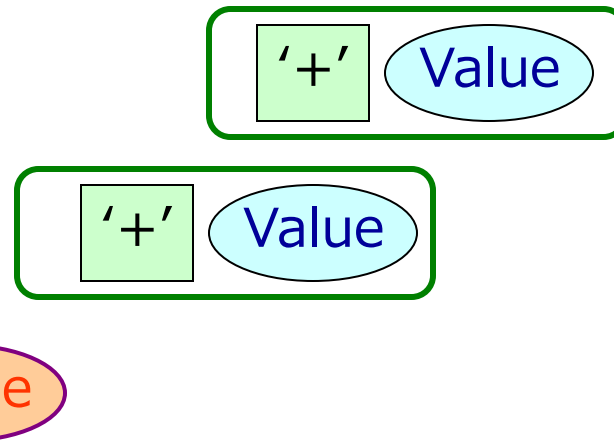
Expr ::= Expr '+' value
| value



- 左再帰でない規則を選ばないと終わらない
 - つまり、いつか左端に左再帰以外の規則が来る

左再帰除去 (3)

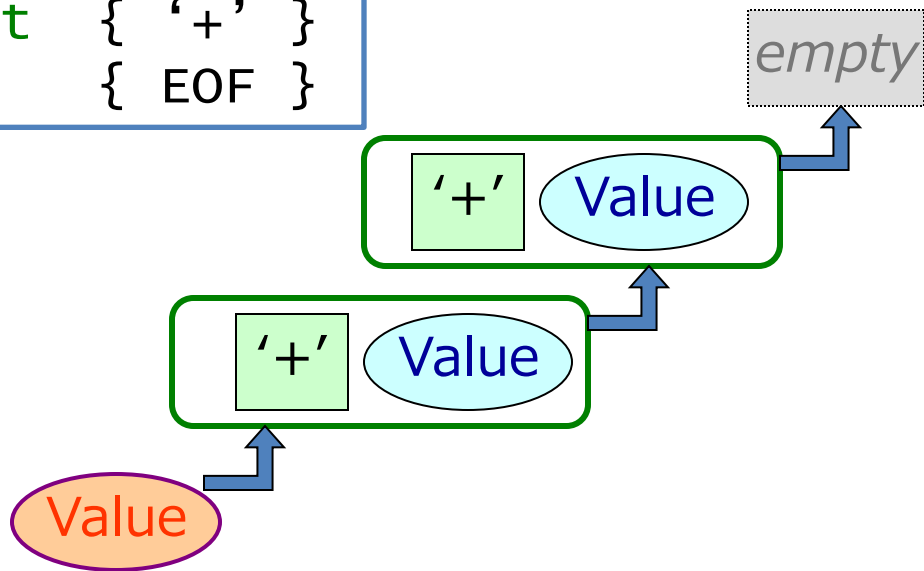
Expr ::= Expr '+' value
| value



- 左再帰の右にあるものが繰り返される
– 繰り返しのなので、再帰は必要になる

左再帰除去 (4)

Expr ::= value Rest	{ NUM }
Rest ::= '+' value Rest	{ '+' }
empty	{ EOF }

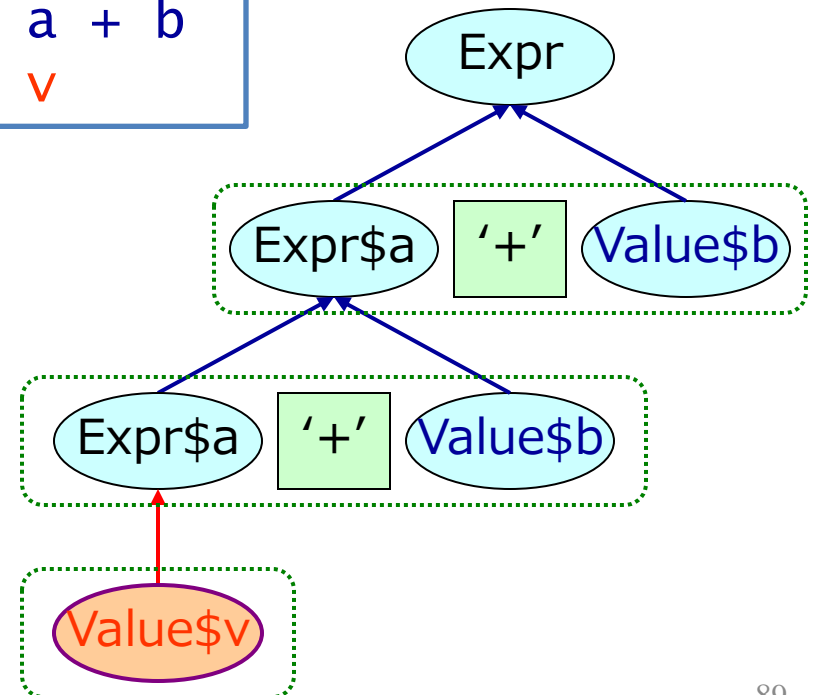


- 右再帰の形式に書き換えられる
 - 多少難しいので、パターンで覚えるとよさげ

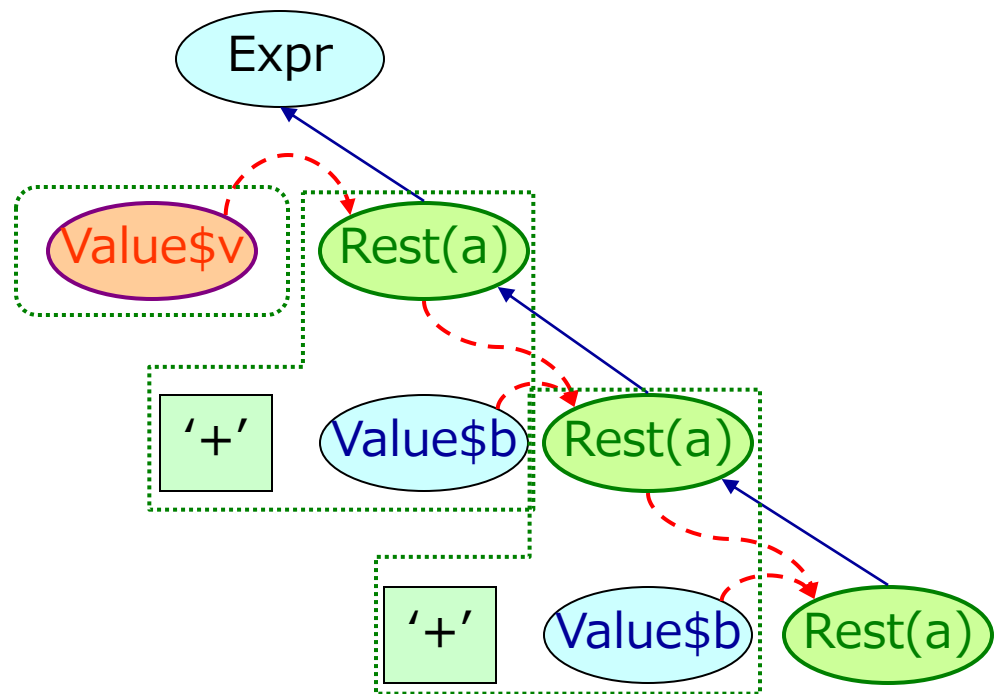
左再帰除去 (5)

- 構文アクションも丁寧にリファクタリングすれば実現可能

Expr ::= Expr\$a '+' value\$b ; a + b
| value\$v ; v



左再帰除去 (6)



Expr ::=	value\$v	Rest(v)\$r	; r
Rest(a) ::=	'+'	value\$b	Rest(a+b)\$r ; r
		empty	; a

empty

左再帰除去 (7)

```
Expr ::= Expr$a '+' value$b ; a + b  
      | value$v ; v
```

```
Expr ::= value$v Rest(v)$r ; r  
Rest(a) ::= '+' value$b Rest(a+b)$r ; r  
          | empty ; a
```

- 左再帰しているパーサ
 - Mooreの左再帰除去を利用して右再帰に変換
 - 丁寧にリファクタリングして属性を正しく渡す
- LL(1)化への鬼門
 - 某大学の講義で正答率が5割切るレベル

練習 (5)

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

```
Prim ::= '(' Expr ')'  
      | value
```

```
value ::= NUM
```

- 左再帰除去でLL(1)化できる？
 - それ以前に左括り出しが必要

練習の解答 (5.1)

```
Expr ::= Expr Right  
      | Prim  
Right ::= '+' Prim  
      | '-' Prim  
Prim  ::= '(' Expr ')'  
      | value  
value ::= NUM
```

- まずは左括り出し

練習の解答 (5.2)

```
Expr ::= Expr Right
      | Prim

Right ::= '+' Prim
      | '-' Prim

Prim  ::= '(' Expr ')'
      | value

value ::= NUM
```

- 左再帰部分と、左端に来る記号を考える

練習の解答 (5.3)

```
Expr ::= Prim Rest
Rest  ::= Right Rest
        | empty
Right ::= '+' Prim
        | '-' Prim
Prim  ::= '(' Expr ')'
        | value
value ::= NUM
```

- Moore

```
Expr ::= Expr Right
        | Prim
```

練習の解答 (5.4)

```
Expr ::= Prim Rest
Rest ::= '+' Prim Rest
       | '-' Prim Rest
       | empty
Right ::= '+' Prim
          | '-' Prim
Prim ::= '(' Expr ')'
       | value
value ::= NUM
```

- Rightをインライン展開すると見やすい？

練習の解答 (5.5)

Expr ::= Prim Rest	{ '(', NUM }
Rest ::= '+' Prim Rest	{ '+' }
'-' Prim Rest	{ '-' }
<i>empty</i>	{ EOF }
Prim ::= '(' Expr ')'	{ '(' }
value	{ NUM }
value ::= NUM	{ NUM }

- LL(1)の確認をしておく

ひとやすみ (4)

- ここまでの知識で解ける課題
 - Q4Parser
 - Q5Parser
- おさらい
 - 左再帰していたら左再帰除去
 - Mooreの左再帰除去で右再帰形に変換
 - そのままではLL(1)にならない
 - 難しいのでパターン化しておく
 - 復習資料に記載

JavaCCを使うともう少し楽ができます

パーサ・ジェネレータの機能

パーサ・ジェネレータの機能 (1)

- 今日これまでにやったこと
 - 終端記号の処理
 - 非終端記号の処理
 - 選言の処理
 - 構文アクションの処理
 - LL(1)構文解析表の作成
 - 左括り出し
 - 左再帰除去

パーサ・ジェネレータの機能 (2)

- JavaCCがやってくれること
 - 終端記号の処理
 - 非終端記号の処理
 - 選言の処理
 - 構文アクションの処理
 - LL(**k**)構文解析表の作成
 - 左括り出し
 - 左再帰除去

終端記号の処理

```
Expr():  
    consume('+')  
    consume(NUM)
```

```
void Expr() : {} {  
    "+" <NUM>  
}
```

非終端記号の処理

```
Expr():  
    value()
```

```
void Expr() : {} {  
    value()  
}
```

選言の処理

```
Expr():  
    if (next token is '-')  
        consume('-')  
        consume(NUMBER)  
    if (next token is '+')  
        consume('+')  
        consume(NUMBER)  
    else raise error
```

最初に来る文字を
勝手に計算してくれる

```
void Expr() : {} {  
    “-” <NUMBER> | “+” <NUMBER>  
}
```


構文アクションの処理

```
Expr():  
    consume('+')  
    var t = consume(NUM)  
    return Integer.parseInt(t.image)
```

```
void Expr() : { Token t; } {  
    “+” t = <NUM>  
    { return Integer.parseInt(t.image); }  
}
```

構文解析表

- 選言を書くだけで、パーサ・ジェネレータは構文解析表を自動で生成
 - LL(1)のように1文字先読みだけでなく、任意の文字数 k を先読みする表も作れる
- ただし、LL文法でない場合にはパーサの生成に失敗する
 - 左括り出し、左再帰除去は自分でやる
 - 構文解析表を意識しながらやる

まとめ

- LL(1)パーサは人間でもかける
 - 手続き型のプログラミング言語と相性がよい
 - ただし、構文解析表を作るなどの負担が大きい
- JavaCCを使うとLL(k)パーサを自動生成
 - 構文解析表を勝手に作ってくれる
 - ただし、LL文法でない場合は失敗
 - 左括り出し
 - 左再帰除去