言語プロセッサ2014
No.13

東京工科大学
コンピュータサイエンス学部
亀田弘之

平成27年1月5日（月）3限

今日の内容

1. ANTLRWorks の紹介
2. ANTLR v4 の紹介
3. プログラミング言語設計へ向けて

学問的キーワード

- 文法
- BNF（Backus-Naur Form）
- 字句解析器（lexer）
- 構文解析器（parser）
- 構文図

Antlrworkの紹介

- もう一つのツール
  （JavaCCもあるよ！）

ANTLRWorksの紹介

1. ANTLRWorks とは何？
2. ANTLRWorks インストール
3. 起動と終了
4. 使い方
   ① 文法作成
   ② 文字解析と構文解析
   ③ 実行

文法作成例

1. 文法名の設定.
   - 文法の名前とファイル名は揃える.
   - 今の場合，文法の名前を **Expr** とするので，文法を格納するファイル名は，**Expr.g** とする.
2. ANTLRWorksをエディタとして使い，文法の設計をする.
   - 文法記述を順次説明する.

Expr. gの仕様

- “120+45*2” や “x = 5” などのプログラム文1つ上が，改行で区切られている．これを順番に読み込んで処理する．
- 例えば，

  1+4

  a = 8

  a+a*2

  というプログラムを読み込ませると，

  5

  24

  と表示される．

文法の作成(1):名前付与

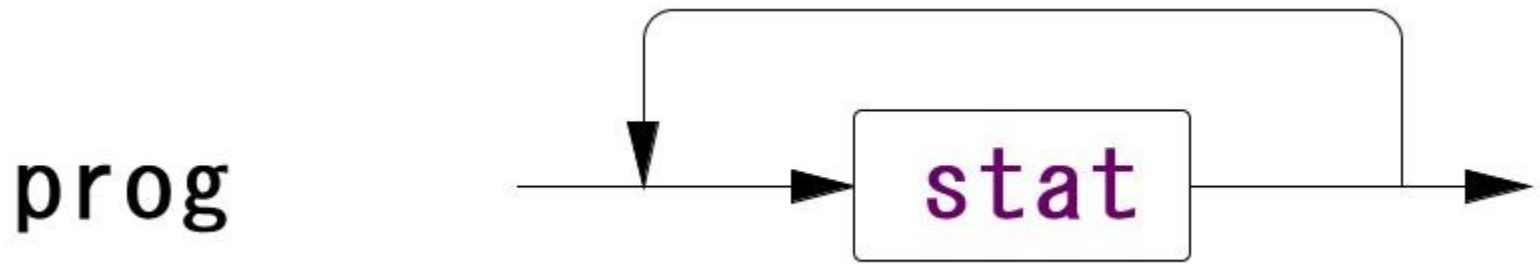
grammar Expr;

文法の名前の定義

文法の作成(2):prog要素定義

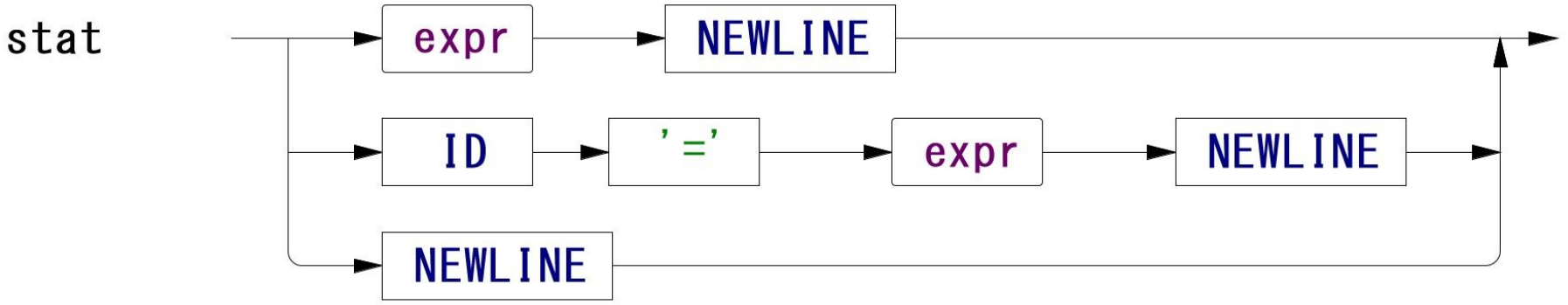
grammar Expr;

prog:　stat+ ;



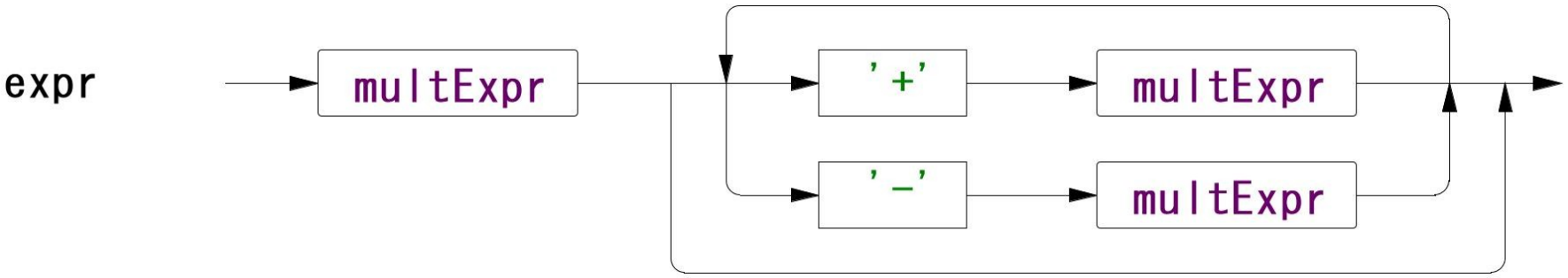
文法の作成(3):stat要素定義

```
grammar    Expr.g ;
prog:   stat+ ;
stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE ;
```



文法の作成(4):expr要素定義

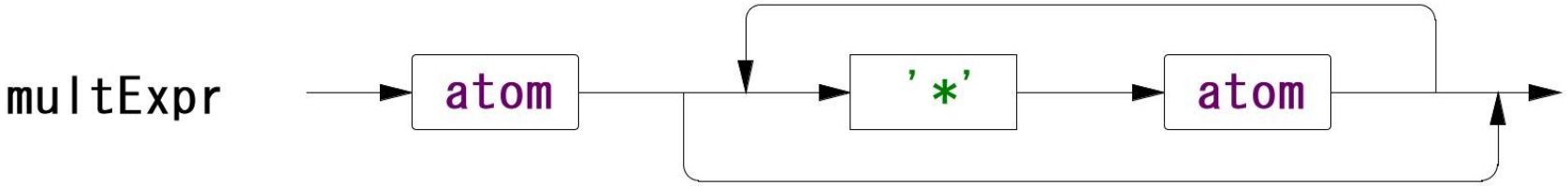
```
expr:
        multExpr
        (   '+'    multExpr    |     '-'    multExpr   )* ;
```



文法の作成(5):multExpr要素定義

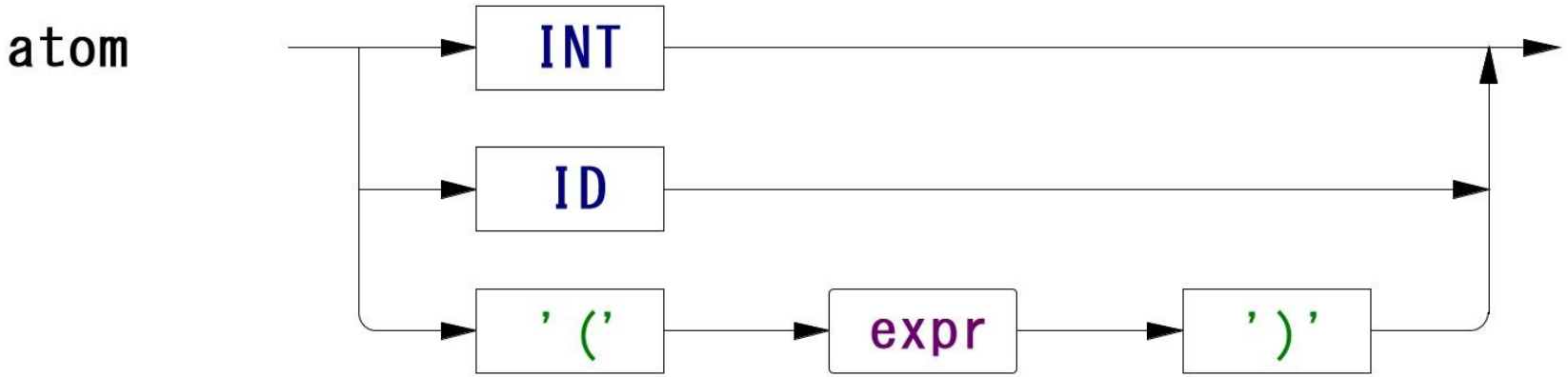
multExpr:

atom ('*‘ atom)* ;



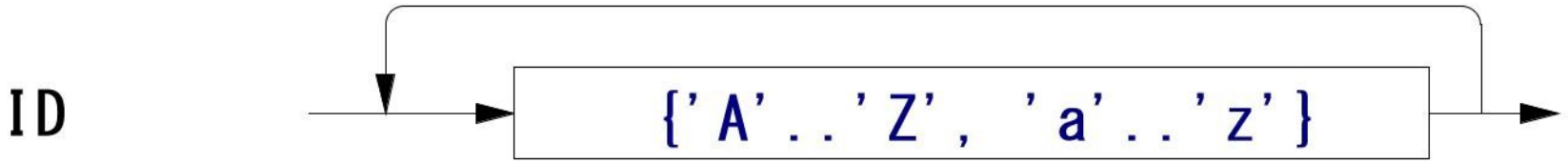
文法の作成(6):atom要素定義

atom: INT | ID | '(' expr ')' ;



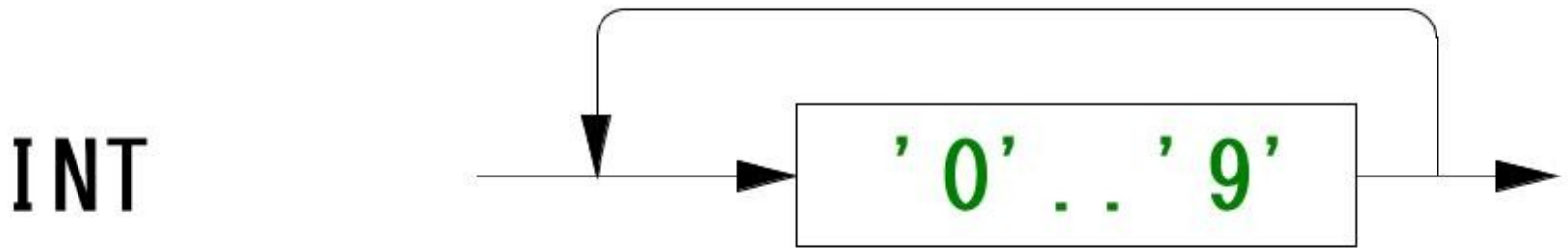
文法の作成(7):atom要素定義

ID : ('a'..'z'|'A'..'Z')+ ;



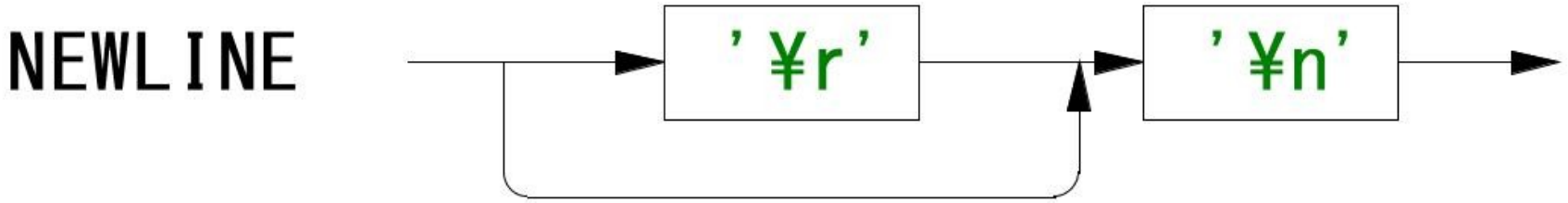
文法の作成(8):INT要素定義

INT : '0'..'9'+ ;



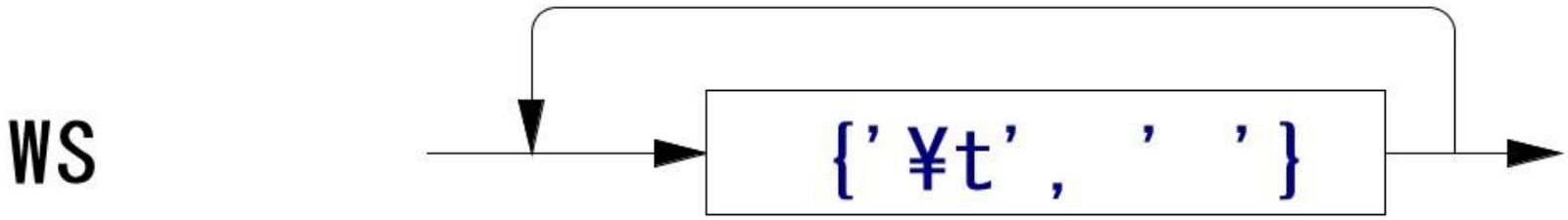
文法の作成(９):NEWLINE要素定義

NEWLINE:'¥r'? '¥n' ;

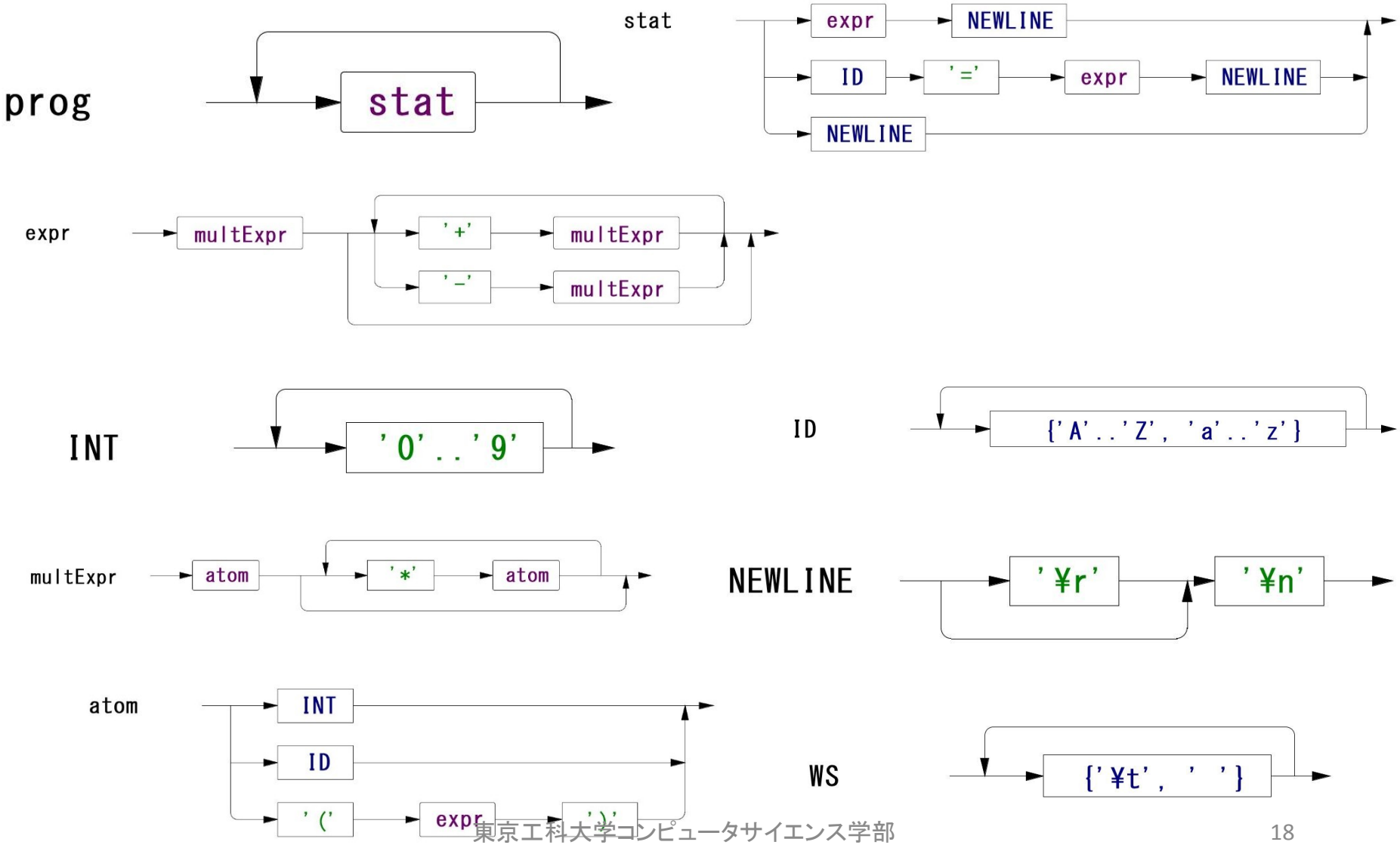


文法の作成(10):WS要素定義

WS : (' '|'¥t')+ {skip();} ;



構文図


prog
stat
stat
expr
NEWLINE
ID
'='
expr
NEWLINE
NEWLINE
expr
multExpr
'+'
multExpr
'-'
multExpr
INT
'0'..'9'
ID
{'A'..'Z', 'a'..'z'}
multExpr
atom
'*'
atom
NEWLINE
'¥r'
'¥n'
atom
INT
ID
'('
expr
')'
WS
{'¥t', ' '}

設計した文法の概要(Expr.g)

```
grammar Expr;
prog:   stat+ ;
stat:   expr  NEWLINE | ID  '=‘  expr NEWLINE   |   NEWLINE ;
expr:   multExpr (   '+'  multExpr |   '-'  multExpr )*;
multExpr:   atom    (  '*‘   atom  )*  ;
atom:   INT   |   ID   |   '(' expr ')' ;
ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'¥r'? '¥n' ;
WS  :   (' '|'¥t')+ {skip();} ;
```

説明できますか？

（注）

- ここまで一所懸命やっている作業は，
  「**文法の設計・作成**」
  ですが，これは
  「**言語の設計・作成**」
  でもあります．

形式言語とオートマトンの授業より
言語Lは文法Gにより定義され，L＝L(G).

このプログラミング言語のプログラム例は以下の通り.

```
A = 100
B = 300
C = A + B * 2
C
1+2+3+4
```

文法をもう少し詳しく見ると…

- 構文の記述だけではなく，動作も記述されている．

文法の動作部分の説明

```
grammar Expr;
@header {
import java.util.HashMap;
}
@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}
```

変数名とその値を保存する表

```
grammar Expr;
prog:   stat+ ;
stat:   expr   NEWLINE   { System.out.println($expr.value); }
    |   ID '=' expr NEWLINE
        { memory.put( $ID.text, new Integer($expr.value) ); }
    |   NEWLINE   ;
expr returns [int value]
    :   e=multExpr {$value = $e.value;}
        (   '+' e=multExpr {$value += $e.value;}
        |   '-' e=multExpr {$value -= $e.value;}
        )*  ;
```

```
grammar Expr;
  （省略）
multExpr returns [int value]
   :  e=atom {$value = $e.value;} (‘*’ e=atom {$value *= $e.value;} )* ;
atom returns [int value]
   :   INT {$value = Integer.parseInt($INT.text);}
   |    ID  {
      Integer v = (Integer)memory.get($ID.text);
      if ( v!=null ) $value = v.intValue();
      else System.err.println("undefined variable "+$ID.text);  }
   |    '(' expr ')' {$value = $expr.value;}  ;
```

```
grammar Expr;
  （省略）

ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'¥r'? '¥n' ;
WS  :   (' '|'¥t')+ {skip();} ;
```

動作させてみよう！

1. 構文着を作る（字句解析＋構文解析）
2. プログラムを実行する（実際に計算させる）

これ以降は，自分でプログラミング言語を設計することになります

```
program  tasu  {
        a = 4
        b = a +6
        a
        b
}
```

この言語の文法はどうなりますでしょうか？

最後に、ANTLR4について

まずは、開発・実行環境の整備

1. ANTLR v4 をダウンロードする。
2. CLASSPATH を設定する。
3. （任意）alias の設定をする。

まずは、開発・実行環境の整備

1. ANTLR v4 をダウンロードする。
   - 対象：Antlr-4.1-complete.jar
   - 設置先：
     - Linuxの場合： /usr/local/bin
     - Windowsの場合：c:¥Javalib
2. CLASSPATH を設定する。
   - Linuxの場合：

     export CLASSPATH=".:/usr/local/bin/antlr-4.1-complete.jar:$CLASSPATH"
   - Windowsの場合：

     SET CLASSPATH=.;c:¥Javalib¥antlr-4.1-complete.jar;%CLASSPATH%
3. （任意）alias の設定をする。
   - **alias antlr4='java –jar /usr/local/bin/antlr-4.1-complete.jar'**
   - **alias grun='java org.antlr.v4.runtime.misc.TestRig'**

練習1　次の文法を入力する

```
Grammar  Hello;
R    :    ‘hello’  ID ;
ID   :    [a-z]+ ;
WS :    [ ¥t¥r¥n]+  ->  skip ;
```

ANTLRWorkに入力し、入力ミスのチェックをする。

注意点

- 文法を格納するファイル名は、“**Hello.g**” です。
- ANTLRWorksを用いて、構文図を描画してみる。

練習1　コンパイルする

```
$ antlr4  Hello.g
$ javac Hello*.java
```

注意点

- リストコマンド（lsやdir）などで、どんな名前のJava言語コードがいくつ生成されたか確認してみよう。
- 自動生成されたプログラムの中を、エディタ等でのぞいてみるのも勉強になります。

練習１　実行する(CUI形式)

```
$ grun  Hello  r  -tree < input.txt
```

- コマンド grun は、alias で別名を与えたもの。
  その実体は以下のことを行ったのと同じ。
  $ java –jar /usr/local/bin/antlr-4.1-complete.jar
- Hello は、文法名がHello だから。
- 引数 r は、文法の開始記号が r だったから。
- オプション –tree は、木構造を出力したいから。
- 入力ファイル input.txt は次ページ参照。

（注） input.txt の中身（1つの例）

```
12+ 81
A = 6
B = A * 10
A * (A + B)
```

キーボード（標準入力）からの
入力の代わり。

実行結果(-treeで実行した場合)

```
$ grun Expr1 prog -tree < input.txt
(prog (stat a = (expr (multExpr (atom 4))) \r\
n) (stat b = (expr (multExpr (atom 7))) \r\n)
(stat (expr (multExpr (atom a) * (atom b)) + (
multExpr (atom a))) \r\n))
```

実行結果（-guiで実行した場合）