課題 3

下の文脈自由文法を構文解析し，抽象構文木を返すプログラムを作成せよ．

- id と num は，それぞれ識別子と整数を表すトークンとする．
- 左再帰を除去した文脈自由文法を考える必要がある．
- 消去可能な非終端記号, first, follow などを手で計算する必要がある．

$$
\begin{array}{rcl}
F & \rightarrow & \mathrm{id} \mid \mathrm{num} \mid (E) \mid \mathrm{Nil} \\
T & \rightarrow & T * F \mid T/F \mid F \\
E & \rightarrow & E + T \mid E - T \mid T \\
B & \rightarrow & E == E \mid E < E \\
C & \rightarrow & E :: C \mid E \\
I & \rightarrow & \mathrm{if}\,(B)\,I\,\mathrm{else}\,I \mid C
\end{array}
$$

follow の計算を行うときは，下の規則を追加して考えてください．

$$S \rightarrow I\$$$

抽象構文

抽象構文を表すための型は，`Abssyn.scala` に以下のように定義している．演算子の種類が多いので，二項演算子を表す型 `BOp` を用意している．

```
  trait BOp                          // 二項 (Binary) 演算子
  case object PlusOp extends BOp
  case object MinusOp extends BOp
  case object TimesOp extends BOp
  case object DivideOp extends BOp
  case object EqOp extends BOp       // E == E
  case object LessOp extends BOp     // E < E
  case object ConsOp extends BOp     // E :: C

  trait Exp
  case class VarExp(s: Var) extends Exp
  case class IntExp(i: Int) extends Exp
  case object NilExp extends Exp
  case class BOpExp(o: BOp, e1: Exp, e2: Exp) extends Exp
  case class IfExp(e: Exp, e1: Exp, e2: Exp) extends Exp
```

構文解析器

構文解析器は，以下のように JFlex で生成された字句解析器のクラス `Yylex` のオブジェクトを引数として，生成される．

- Parser のオブジェクトが new されると, `var tok: Token = src.yylex()` により最初のトークンが, 変数 `tok` に代入される．
- 関数 `advance()` が呼ばれると `src.yylex()` が呼ばれ, 次のトークンが変数 `tok` に代入される．

```
class Parser (val src: Yylex) {

  var tok: Token = src.yylex()

  def advance () =  tok = src.yylex()
  ...
}
```

実際に，再帰下降型構文解析を行うプログラムは，以下のようにスケルトンが与えられているので，関数 E などを実装すれば良い．

```
  def F(): Exp = ...

  def T(): Exp =
    tok match {
      case ID(_) | INT(_) | LPAREN | NIL =>  Tprime(F())
      case _ => error()
    }

  def Tprime(e: Exp): Exp =
    tok match {
      case TIMES => eat(TIMES); Tprime(BOpExp(TimesOp, e, F()))
      case DIV => eat(DIV); Tprime(BOpExp(DivideOp, e, F()))
      case PLUS | MINUS| RPAREN | EOF | ELSE | EQEQ | LESS | COLONCOLON => e
      case _ => error()
    }

  def E(): Exp = T()  // プログラムを書く．補助関数も必要

  def C(): Exp = E()  // プログラムを書く．補助関数も必要

  def B(): Exp = E()  // プログラムを書く．補助関数も必要

  def I(): Exp = C()  // プログラムを書く．
```

実行例

字句解析器と構文解析器を合わせて，文字列を構文解析するための関数が Main.scala に，以下のように実現されている．

```
object Main {
  def parseStr(s: String) = {
    val lexer = new Yylex (new java.io.StringReader(s))
    val parser = new Parser(lexer)
    parser.I()
  }
}
```

実行例を示す．

```
scala> Main.parseStr("x+10")
res1: Abssyn.Exp = BOpExp(PlusOp,VarExp(x),IntExp(10))

scala> Main.parseStr("if (x==10) x else y")
res2: Abssyn.Exp = IfExp(BOpExp(EqOp,VarExp(x),IntExp(10)),VarExp(x),VarExp(y))
```

sbt で実行できるテストを用意しているので，正しく実装できれば，以下のように ParserTest のすべてのテストが成功する．

```
> testOnly ParserTest
[info] Compiling 1 Scala source to /Users/minamide/lectures/compiler/kadai-ans/parsing-sbt/target/scala
[info] ParserTest:
[info] *
[info] - should 左に強く結合
[info] カッコ
```

```
[info] - should 正しく結合
...
[info] All tests passed.
[info] Passed: Total 9, Failed 0, Errors 0, Passed 9
[success] Total time: 1 s, completed 2016/12/07 10:42:13
```

課題: オプション課題

本講義では，以下の抽象構文を持つプログラム言語 (Nonscala) のコンパイラを作成する．この課題では，Nonscala の構文解析器を作成する．

E	$\rightarrow$	num	整数定数
		id	変数
		Nil	空リスト
		id$(E, \ldots, E)$	関数適用
		id.id	引数なしのメソッド呼び出し
		$E * E$	
		E / E	
		$E + E$	
		$E - E$	
		$E :: E$	
		$E == E$	
		$E < E$	
		(E)	
		if (E) E else E	
T	$\rightarrow$	id	
		id$[T]$	
D	$\rightarrow$	def id$(\text{id} : T, \ldots, \text{id} : T) : T = E$	

抽象構文に対応した型が，Abssyn.scala に定義してある．Nonscala の型 T としては，以下の 3 つの型のみを考える．

- Int, Boolean, List[Int]

そのため T に対応する型 `Ty` の定義は，以下のようになっている．

```
trait Ty
case object IntTy extends Ty
case object BoolTy extends Ty
case object IntListTy extends Ty
```

メソッド呼び出しは，x.isEmpty, x.head, x.tail のみを考え，1 引数の演算と解釈し下の形の抽象構文で表す．

```
UOpExp(isEmptyOp, x), UOpExp(headOp, x), UOpExp(tailOp, x)
```

曖昧さを除去した以下の文脈自由文法に基づき，構文解析するプログラムを作成せよ．構文解析を簡単にするために，構文をかなり制限している．例えば，

- ブール式は，if 式の条件文にしか書けない．
- コンス (::) の第 1 引数は E なので，第 1 引数にコンスを含む式を書けない．

$$
\begin{array}{rcl}
F & \to & \mathrm{id} \mid \mathrm{num} \mid \mathrm{Nil} \mid (E) \mid \mathrm{id}(C, \ldots, C) \mid \\
 & & \mathrm{id.isEmpty} \mid \mathrm{id.head} \mid \mathrm{id.tail} \\
T & \to & T * F \mid T/F \mid F \\
E & \to & E + T \mid E - T \mid T \\
C & \to & E :: C \mid E \\
B & \to & E == E \mid E < E \mid E \\
I & \to & \mathrm{if}\,(B)\,I\,\mathrm{else}\,I \mid C \\
D & \to & \mathrm{def}\,\mathrm{id}(\mathrm{id} : U, \ldots, \mathrm{id} : U) : U = I \\
U & \to & \mathrm{Int} \mid \mathrm{Boolean} \mid \mathrm{List[Int]}
\end{array}
$$

- 関数適用の ... の部分，関数定義の ... の部分は，文脈自由文法にする必要がある．

実行例

```
scala> Main.parseStrNonscalaDef("def f(x: Int, y: Boolean): List[Int] = Nil")
res0: Abssyn.Def = Def(f,List((x,IntTy), (y,BoolTy)),IntListTy,NilExp)
```

構文解析器の簡単なテストが用意してある．

```
> testOnly ParserNonscalaTest
[info] ParserNonscalaTest:
[info] *
[info] - should 左に強く結合
[info] カッコ
[info] - should 正しく結合
[info] *と+
...
[info] Tests: succeeded 14, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 14, Failed 0, Errors 0, Passed 14
[success] Total time: 0 s, completed 2016/12/07 10:43:19
```

- テストの中で，`examples/insert.scala` を構文解析するテストを実行している．

提出するファイル

以下のファイルの中で，変更したものを提出すること．

```
src/Parser.scala
src/ParserNonscala.scala
```

他にも変更したファイルや追加したファイルがあれば，提出してください．