# Software Tools
# ANTLR

Part II - Lecture 8

# Today's Outline

- Introduction to ANTLR
- Parsing Actions
- Generators

# Introduction to ANTLR

# ANTLR

- Parser/lexer generator: takes a grammar and generates a LL(k) lexer and/or parser for you
  - Written in Java, open source software
  - Can generate Java, C#, C++, Python, …
  - Uses the regular expression / grammar syntax that we have learned in the last lecture
  - Grammar files have suffix `.g`
- Besides simple LL(k), ANTLR supports backtracking:
  - If it is unclear which rule alternative to apply, alternatives are applied speculatively
  - If a choice turns out to be wrong, backtracking is used and another alternative is tried

# ANTLR Example: Java.g

```
grammar Java;
options { backtrack=true;  memoize=true; }

compilationUnit:
  ( (annotations)?  packageDeclaration )?
  (importDeclaration)* (typeDeclaration)*  ;

packageDeclaration: 'package' qualifiedName ';' ;

importDeclaration:
  'import' ('static')?  IDENTIFIER '.' '*' ';'
| 'import' ('static')?
  IDENTIFIER ('.' IDENTIFIER)+ ('.' '*')? ';' ;

qualifiedImportName: IDENTIFIER ('.' IDENTIFIER)* ; // ...
```

- The "Java" grammar uses backtracking
- Some grammar rules define simple tokens directly, e.g. `'import'`, `'static'`
- Grammar rules also refer to tokens of the lexer, which is defined later on in Java.g

# The Lexer in Java.g

```
LONGLITERAL: IntegerNumber LongSuffix ;

INTLITERAL: IntegerNumber ;

fragment IntegerNumber: '0'          // number zero
  |   '1'..'9' ('0'..'9')*           // decimal numbers
  |   '0' ('0'..'7')+                // octal numbers
  |   HexPrefix HexDigit+ ;          // hexadecimal numbers

fragment HexPrefix: '0x' | '0X' ;

fragment HexDigit: ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment LongSuffix: 'l' | 'L' ;

ABSTRACT: 'abstract' ;    // ...
```

- The lexer rules come right after the parser rules (some grammars have an optional `lexer` *lexerName*; )
- Lexer rules use essentially the same syntax as parser rules
- Lexer rules can use subrules (`fragment` rules) that do not define tokens themselves but are used by other rules

# Generating and Using Lexers and Parsers

- Generate Java classes for parser and lexer by executing class org.antlr.Tool with command line arguments:

```
-Xconversiontimeout 100000 -o src\pdstore\java Java.g
```

(timeout for backtracking) (output folder) (input)

- This generates classes JavaLexer and JavaParser, which can be used from other classes, e.g.

```
import org.antlr.runtime.*;   // ...
public class Import {
   public static void main(String[] args) {   // ...
      CharStream input = new ANTLRFileStream(args[0]);
      JavaLexer lexer = new JavaLexer(input);
      CommonTokenStream tokens = new CommonTokenStream();
      tokens.setTokenSource(lexer);
      JavaParser parser = new JavaParser(tokens);
      // start parsing at the compilationUnit rule
      parser.compilationUnit();   // ...
} }
```

# Parsing Actions

# Parsing Actions

- We want to do things with the source code we parse
- Idea: whenever we have recognized part of the language, execute some code ("action")
- Actions can be at beginning (`@init{ }`), end (`@after{ }`) or anywhere else in the rule body (`{ }`)

```
compilationUnit
@init { System.out.println("Rule application has begun");}
@after{ System.out.println("Rule application has ended");}
: ((annotations)? packageDeclaration
   { System.out.println("Parsed packageDeclaration");}
  )?
  (importDeclaration
   { System.out.println("Parsed importDeclaration"); }
  )*
  (typeDeclaration { ….println("Parsed typeDeclaration"); })*
;
```

# Accessing and Returning Values from Rules

- Rules are used to generate methods that can return values: add `returns [`*Type varName*`]` after rule name
- To access return values, assign a variable `var=ruleName` or `var=TOKEN` and access its fields
- The variable is declared for you by ANTLR and can be accessed in actions with `$var`
- Tokens have their text string in `$var.text`

```
packageDeclaration
: 'package' name=qualifiedName
  { System.out.println("qualifiedName="+$name.value); }
  ';' ;

qualifiedName returns [String value]
: id=IDENTIFIER  { $value = $id.text; }
  ('.' id=IDENTIFIER  { $value += "." + $id.text; } )* ;
```

# Example: Accessing and Returning Values

The following rule prints out the source code it parses:

```
importDeclaration
@init{ String s = "import "; }
:   'import'  ('static'  { s += "static "; } )?
    id=IDENTIFIER '.' '*' ';'
    { System.out.println(s + $id.text + ".*;"); }
|   'import'  ('static' { s += "static "; } )?
    id=IDENTIFIER
    { s += $id.text; }
    ('.' id=IDENTIFIER  { s += "." + $id.text; } )+
    ('.' '*' { s += ".*"; } )?
    ';'
    { System.out.println(s + ";"); }
;
```

# Debugging Parsing Actions

- ANTLR will not check the Java code in the actions, i.e. the generated class might contain errors
- Eclipse's compiler will show you syntax errors after reloading the generated .java file (F5 for reload)
- For each rule, ANTLR will generate a method with the rule name

```
importDeclaration returns
[PDJavaImport value] …
: 'import' ('static')?
  id=IDENTIFIER '.' '*' ';'
  {
    PDJavaPackage package = …
  }
  …
;
```

ANTLR

```
public final PDJavaImport
importDeclaration() throws
RecognitionException {
  …
  if (state.backtracking==0 )
  {
    PDJavaPackage package = …
  }
  …
}
```

Error: package is a Java keyword

# Example: Building an AST

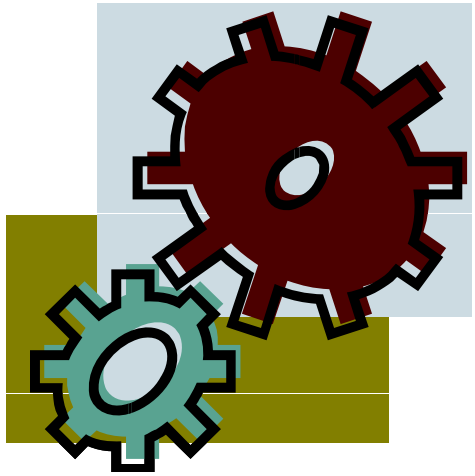
Idea: each rule returns AST node and gets the returned AST nodes of the rules it uses

```
compilationUnit returns [JavaCompilationUnit value]
@init {
   $value = new JavaCompilationUnit();
}  :   (   (annotations)?
            packageDecl=packageDeclaration
            { $value.setPackage($packageDecl.value); }
        )?
        (importDecl=importDeclaration
        { $value.addImports($importDecl.value); }
        )*
        (typeDecl=typeDeclaration
        { $value.addTypeDefinitions($typeDecl.value); }
        )* ;
```

# Building an AST Cont'd

```
importDeclaration returns [JavaImport value]
@init {
  $value = new JavaImport();
  String name = null;
  boolean isPackage = false;
} : …
  | 'import' ('static')? id=IDENTIFIER { name = $id.text; }
    ('.' id=IDENTIFIER { name += "." + $id.text; } )+
    ('.' '*' { isPackage = true; } )? ';'
    {
      if (isPackage) {
        JavaPackage p = new JavaPackage();
        p.setName(name); $value.setPackage(p);
      } else {
        JavaType type = new JavaType();
        type.setName(name); $value.setType(type);
    } } ;
```

# Generators



# Writing Generators

- Generators traverse the AST that was generated by the parser
- For each AST node, they generate some output
- Easy way to implement:
  - For important AST node types, write a generator method
  - Method for AST node type X calls other methods to do generation for child node types of X
- Examples:
  - Source code printer
  - Source code converter (i.e. print another language)

# Java Printer

```
public class JavaPrinter {
  PrintStream s;
  public JavaPrinter(OutputStream out) {
    s = new PrintStream(out);
  }
  public void printCompilationUnit(
                  JavaCompilationUnit compilationUnit) {
    s.println("package " +
       compilationUnit.getPackage().getName() + ";");
    s.println();  // use separate method to print imports
    for (JavaImport i : compilationUnit.getImports())
      printImport(i);
    s.println();  // use separate method to print types
    for (JavaType t : compilationUnit.getTypeDefinitions())
      printType(t);
  } …
```

# Java Printer Cont.

```
public void printImport(JavaImport javaImport) {
  if (javaImport.getPackage() != null)
    s.println("import " + javaImport.getPackage().getName()
       + ".*;");
  else if (javaImport.getType() != null)
    s.println("import " + javaImport.getType().getName()
       + ";");
}

public void printType(JavaType type) {
  if (type.getJavaInterface() != null)
    s.println("interface " + type.getJavaInterface().getName()
       + " { ... }");
  else if (type.getJavaClass() != null)
    s.println("class " + type.getJavaClass().getName()
       + " { ... }");
}
```

# Using the Java Printer

```
public class Import {
  public static void main(String[] args) { …
    // Create a parser that reads from the token stream
    JavaParser parser = new JavaParser(tokens);

    // start parsing at the compilationUnit rule
    JavaCompilationUnit compilationUnit =
        parser.compilationUnit();

    // set up a JavaPrinter that prints to the standard output
    JavaPrinter printer = new JavaPrinter(System.out);

    // print the AST
    printer.printCompilationUnit(compilationUnit);
    …
} }
```

# Summary

# Today's Summary

- ANTLR is a tool that can generate LL(k) parsers and lexers in Java
- By adding actions to a parser rule, Java code can be executed after something has been parsed
- Actions can create ASTs
- Generators traverse an AST and produce output recursively for each AST node

References:

- ANTLR Homepage with Online Documentation. http://www.antlr.org/
- Scott Stanchfield. An ANTLR 2.0 Tutorial. http://javadude.com/articles/antlrtut/