Parser Generation

Prof. Dr. Ralf Lämmel
Universität Koblenz-Landau
Software Languages Team

Mainly with applications to language processing
Program generation

Specification → Program generator → Program
Parser generation

Grammar with some code → Program generator → Parser
Generative programming is a style of computer programming that uses automated source code creation through generic frames, classes, prototypes, templates, aspects, and code generators to improve programmer productivity ... It is often related to code-reuse topics such as component-based software engineering and product family engineering.

Retrieved from Wikipedia on 3 May, 2011.
Language processing patterns

1. The Chopper Pattern
2. The Lexer Pattern
3. The Copy/Replace Pattern
4. The Acceptor Pattern
5. The Parser Pattern
6. The Lexer Generation Pattern
7. The Acceptor Generation Pattern
8. The Parser Generation Pattern
9. The Text-to-object Pattern
10. The Parser Generation$^2$ Pattern
11. (The Text-to-tree Pattern)
12. (The Tree-walk Pattern)
13. The Object-to-text Pattern

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
The *Lexer Generation Pattern*

See the *Lexer Pattern* for comparison
The **Lexer Generation Pattern**

- Intent:

  *Analyze text at the lexical level.*

  *Use token-level grammar as input.*

- Operational steps (compile time):

  1. Generate code from the lexer grammar.
  2. Write boilerplate code to drive the lexer.
  3. Write code to process token/lexeme stream.
  4. Build generated and hand-written code.
Token-level grammar for 101

```
COMPANY     : 'company';
DEPARTMENT  : 'department';
EMPLOYEE    : 'employee';
MANAGER     : 'manager';
ADDRESS     : 'address';
SALARY      : 'salary';
OPEN        : '{';
CLOSE       : '}';
STRING      : '"' (~""')* '"';
FLOAT       : ('0'..'9')+ ('.' ('0'..'9')+)?;
WS          : (\s|\r?\n|\t)+;
```

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Lexer generation

ANTLR

....g

....java
“ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. ANTLR provides excellent support for tree construction, tree walking, translation, error recovery, and error reporting.”

http://www.antlr.org/
Boilerplate code for lexer execution

```java
FileInputStream stream = new FileInputStream(file);
ANTLRInputStream antlr = new ANTLRInputStream(stream);
MyLexer lexer = new MyLexer(antlr);
Token token;
while ((token = lexer.nextToken()) != Token.EOF_TOKEN) {
    ...
}
```

Grammar-derived class

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
http://101companies.org/wiki/Contribution:antlrLexer
Summary of implementation aspects

- Define regular expressions for all tokens.
- Generate code for lexer.
- Set up input and token/lexeme stream.
- Implement operations by iteration over pairs.
- Build generated and hand-written code.
The Acceptor Generation Pattern
The Acceptor Generation Pattern

Intent:

*Verify syntactical correctness of input.*
*Use context-free grammar as input.*

Operational steps (compile time):

1. Generate code from the context-free grammar.
2. Write boilerplate code to drive the acceptor (parser).
3. Build generated and hand-written code.
Context-free grammar for 101

```plaintext
company :
   'company' STRING '{' department* '}' EOF;

department :
   'department' STRING '{'
      ('manager' employee)
      ('employee' employee)*
         department*
   '}'

employee :
   STRING '{'
      'address' STRING
      'salary' FLOAT
   '}'
```

Wanted: a generated acceptor
Parser generation

ANTLR

...Lexer.java

...Parser.java
Driver for acceptor

import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        SetyLexer lexer = new ...Lexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ...Parser parser = new ...Parser(tokens);
        parser....();
    }
}

This is basically boilerplate code. Copy and Paste!

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Driver for acceptor

```java
import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        SetyLexer lexer = new ...Lexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ...Parser parser = new ...Parser(tokens);
        parser....();
    }
}
```

Prepare System.in as ANTLR input stream
import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        SetyLexer lexer = new ...Lexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ...Parser parser = new ...Parser(tokens);
        parser....();
    }
}
Driver for acceptor

```java
import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        SetyLexer lexer = new ...Lexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ...Parser parser = new ...Parser(tokens);
        parser....();
    }
}
```

Obtain token stream from lexer
import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        SetyLexer lexer = new ...Lexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ...Parser parser = new ...Parser(tokens);
        parser....();
    }
}
Driver for acceptor

import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        SetyLexer lexer = new ...Lexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ...Parser parser = new ...Parser(tokens);
        parser....();
    }
}
http://101companies.org/wiki/
Contribution: antlrAcceptor
Summary of implementation aspects

- Define regular expressions for all tokens.
- Define context-free productions for all nonterminals.
- Generate code for acceptor (parser).
- Set up input and token/lexeme stream.
- Feed parser with token stream.
- Acceptor (parser) execution = call start symbol.
- Build generated and hand-written code.
The *Parser Generation Pattern*
The Parser Generation Pattern

- Intent:

  Enable structure-aware functionality.

  Use context-free grammar as the primary input.

- Operational steps (compile time):

  1. Include semantic actions into context-free grammar.
  2. Generate code from enhanced grammar.
  3. Write boilerplate code to drive the parser.
  4. Build generated and hand-written code.
A grammar rule with a semantic action

employee : STRING '{'
    'address' STRING
    'salary' s=FLOAT
    { total += Double.parseDouble($s.text); }
'}';

Semantic actions are enclosed between braces.

Variable local to parser execution

Retrieve and parse lexeme for the number
Demo

http://101companies.org/wiki/
Contribution:antlrParser

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Parser descriptions can also use “synthesized” attributes.

Add *synthesized attributes* to nonterminals. Add *semantic actions* to productions.

```plaintext
expr returns [int value]
  : e=multExpr {$value = $e.value;}
    ( '+' e=multExpr {$value += $e.value;}
    |  '-' e=multExpr {$value -= $e.value;}
    )*;
```

(Context-free part in bold face.)
The *Text-to-object* Pattern
The *Text-to-object* Pattern

**Intent:**

*Enable structure-aware OO programs.*

*Build AST objects along parsing.*

Operational steps (compile time):

1. Define object model for AST.
2. Instantiate *Parser Generation* pattern as follows:
   
   Use semantic actions for AST construction.
Acceptor---for comparison

company :
    'company' STRING '{'
    department*
    '
'}'
EOF;
With semantic actions for object construction included

company returns [Company c]:
{ $c = new Company(); }  
'company' STRING
{ $c.setName($STRING.text); }  
'{
( topdept= department
 { $c.getDepts().add($topdept.d); }
)*
'}
;  

Construct company object.  
Set name of company.  
Add all departments.
http://101companies.org/wiki/Contribution:antlrObjects
The *Parser Generation*\(^2\) Pattern

(That is, the *Parser Generation Generation* Pattern.)
The *Parser Generation*² Pattern

- **Intent:**
  
  *Enable structure-aware OO programs.*
  
  *Build AST objects along parsing.*
  
  *Use idiomatic context-free grammar as the only input.*

- **Operational steps (compile time):**
  
  1. Author idiomatic context-free grammar.
  2. Generate object model from context-free grammar.
  3. Generate enhanced grammar for AST construction.
  4. Generate parser and continue as before.
More declarative parser generation

idiomatic grammar

Program generator

.....g

.....java

object model
Idiomatic grammar for 101

Company: ...;

Department:
  'department' dname=QqString '{'
    'manager' manager=Employee
    subdepartments=Department*
    employees=NonManager*
  '}'
  
NonManager: ...;
Employee: ...;

Leverage predefined token-level grammar

Label all nonterminal occurrences
http://101companies.org/wiki/Contribution:yapg
Bootstrapping

1. Develop ANTLR-based parser for idiomatic grammars.
2. Develop object model for ASTs for idiomatic grammars.
3. Develop ANTLR generator.
4. Develop object model generator.
5. Define idiomatic grammar of idiomatic grammars.
6. Generate grammar parser from said grammar.
7. Generate object model from said grammar.
The grammar of grammars

Grammar     : prods=Production*;
Production   : lhs=Id ':' rhs=Expression ';';
Expression   : Sequence | Choice;
Sequence     : list=Atom*;
Choice       : first=Id rest=Option*;
Atom         : Terminal | Nonterminal | Many;
Terminal     : symbol=QString;
Nonterminal  : label=Id '=' symbol=Id;
Option       : '|' symbol=Id;
Many         : elem=Nonterminal '*';
The *Text-to-tree* Pattern

Use trees rather than POJOs for AST construction along parsing

Mentioned in passing
The *Text-to-tree Pattern*

- **Intent:**
  
  *Enable structure-aware grammar-based programs.*
  
  *Build AST trees along parsing.*
  
  *Use grammar with tree construction actions as input.*

- **Operational steps (compile time):**
  
  1. Author grammar with tree construction actions.
  2. Generate parser.
Tree construction with ANTLR

department :
  'department' name=STRING '{'
    manager
    ('employee' employee)*
    department*
  '}'
-> ^(DEPT $name manager employee* department*)
;
http://101companies.org/wiki/
Contribution: antlrTrees
The *Tree-walk* Pattern

*Mentioned in passing*
The *Tree-walk* Pattern

- **Intent:**
  
  *Operate on trees in a grammar-based manner.*

  *Productions describe tree structure.*

  *Semantic actions describe actions along tree walking.*

- **Operational steps (compile time):**
  
  1. Author tree grammar (as opposed to CFG).
  2. Add semantic actions.
  3. Generate and build tree walker.
Matching trees rather than parsing text.

Semantic actions as before.

company :
  ^(COMPANY STRING dept*)
  ;

department :
  ^(DEPT STRING manager employee* dept*)
  ;

manager :
  ^(MANAGER employee)
  ;

employee :
  ^(EMPLOYEE STRING STRING FLOAT)
  { total += Double.parseDouble($FLOAT.text); }
  ;
**Cut**: Tree walking with ANTLR

```
topdown : employee;

employee :
  ^(EMPLOYEE STRING STRING s=FLOAT)
  -> ^(EMPLOYEE
      STRING
      STRING
      FLOAT[
          Double.toString(
              Double.parseDouble(
                  $s.text) / 2.0d))
  ]
```

Tree walking strategy

Match trees

Rebuild tree
http://101companies.org/wiki/Contribution:antlrTrees
Summary

- Language processing is a programming domain.
- Grammars are program specifications in that domain.
- Those specifications may be enhanced semantically.
- Those specifications may also be constrained idiomatically.
- Manual implementation of such specifications is not productive.
- Parser generators automatically implement such specifications.
- Parser generation is an instance of generative programming.
A development cycle for language-based tools

1. Author language samples.
2. Complete samples into test cases.
3. Author the EBNF for the language.
4. Refine EBNF into acceptor (using generation or not).
5. Build and test the acceptor.
6. Extend the acceptor with semantic actions.
7. Build and test the language processor.

Some bits of software engineering