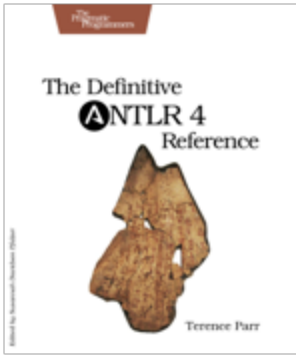
Introduction to ANTLR 4



The latest and last (sic!) revision

Oliver Zeigermann

Code Generation Cambridge 2013

Outline

- Introduction to parsing with ANTLR4
- Use case: Interpreter
- Use case: Code Generator
- The tiniest bit of theory: Adaptive LL(*)
- Comparing ANTLR4

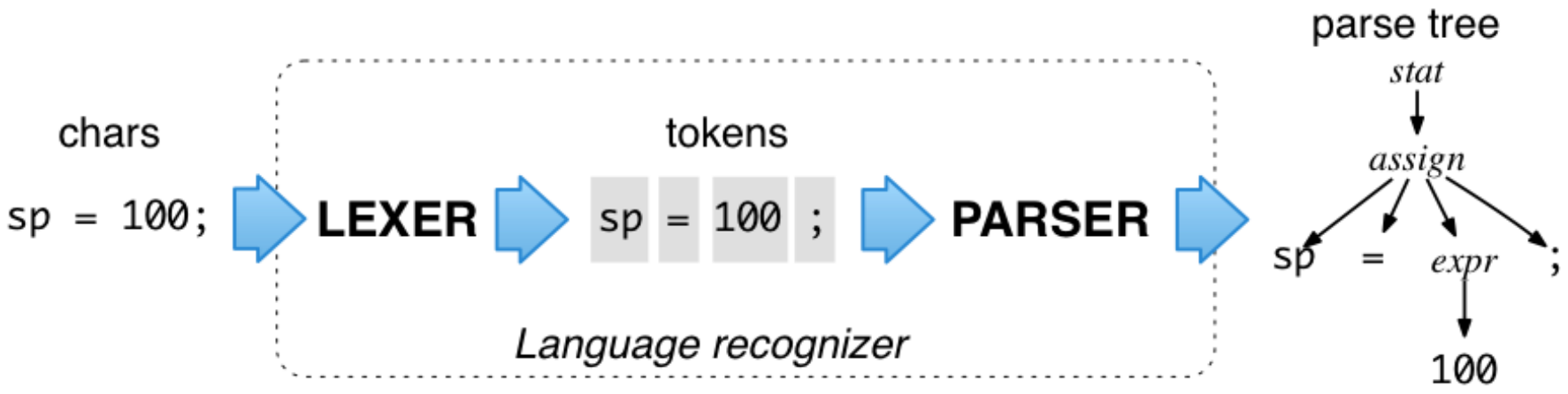
ANTLR 4: Management Summary

- Parser generator
- Language tool
- Design goals
  - ease-of-use over performance
  - simplicity over complexity
- Eats all context free grammars
  - with a small caveat

What does parsing with ANTLR4 mean?

Identify and reveal the implicit structure of an input in a parse tree

- Structure is described as a grammar
- Recognizer is divided into lexer and parser
- Parse tree can be auto generated



Practical example #1: An expression interpreter

Build an interpreter from scratch

- complete and working
- parses numerical expressions
- does the calculation

Lexical rule for integers

```
INT : ('0'..'9')+ ;
```

- Rule for token *INT*
- Composed of digits *0* to *9*
- At least one digit

Syntactic rule for expressions

```
expr : expr ('*'|'/') expr
     | expr ('+'|'-') expr
     | INT
     ;
```

- Three options
- Operator precedence by order of options
- Notice left recursion

```
grammar Expressions;

start : expr ;

expr : left=expr op=('*'|'/') right=expr #opExpr
     | left=expr op=('+'|'-') right=expr #opExpr
     | atom=INT #atomExpr
     ;

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

WS : [ \t\r\n]+ -> skip ;
```

Doing something based on input

Options for separation of actions from grammars in ANTLR4

1. Listeners (SAX style)
2. **Visitors (DOM style)**

Introducing Visitors

Yes, that well known pattern

- Actions are separated from the grammar
- Visitors actively control traversal of parse tree
  - if and
  - how
- Calls of visitor methods can return values

Visitor: actively visiting tree

```
int visitOpExpr(OpExprContext ctx) {
        int left = visit(ctx.left);
        int right = visit(ctx.right);
        switch (ctx.op) {
            case '*' : return left * right;
            case '/' : return left / right;
            case '+' : return left + right;
            case '-' : return left - right;
        }
}
```

Visitor methods pseudo code

```
int visitStart(StartContext ctx) {
 return visitChildren(ctx);
}
int visitAtomExpr(AtomExprContext ctx) {
 return Integer.parseInt(ctx.atom.getText());
}
int visitOpExpr(OpExprContext ctx) {
 // what we just saw
}
```

Live Demo

Practical example #2: Code generation from Java

Generate interface declarations for existing class definitions

- Adapted from ANTLR4 book
- Like a large scale refactoring or code generation
- Keep method signature *exactly* as it is
- Good news
  - You can use an existing Java grammar
  - You just have to figure out what is/are the rule(s) you are interested in: *methodDeclaration*

Doing something based on input

Options for separation of actions from grammars

1. **Listeners (SAX style)**
2. Visitors (DOM style)

Introducing listeners

Yes, that well known pattern (aka Observer)

- Automatic depth-first traversal of parse tree
- walker fires events that you can handle

Listener for method declaration, almost real code

```
void enterMethodDeclaration(
                MethodDeclarationContext ctx) {
    // gets me the text of all the tokens that have
    // been matched while parsing a
    // method declaration
    String fullSignature =
        parser.getTokenStream().getText(ctx);
    print(fullSignature + ";");
}
```

That's it!

(plus minor glue code)

Live Demo

ANTLR4 Adaptive LL(*) or ALL(*)

- Generates recursive descent parser
  - like what you would build by hand
- does dynamic grammar analysis while parsing
  - saves results like a JIT does
- allows for direct left recursion
  - indirect left recursion not supported
- parsing time typically near-linear

Comparing to ANTLR3

top-down SLL(*) parser with optional backtracking and memoization

- ANTLR4
  - can handler a larger set of grammars
  - supports lexer modes
- ANTLR3
  - relies on embedded actions
  - has tree parser and transformer
  - can't handle left recursive grammars

Comparision Summary

- PEG / packrat
  - can't do left recursion
  - similar to ANTLR3 with backtracking
  - error recovery or reporting hard
- GLR
  - bottom-up like yacc
  - can do all grammars
  - debugging on the declarative specification only
- GLL
  - top-down like ANTLR
  - can do all grammars

Comparing to PEG / packrat

top-down parsing with backtracking and memoization

- Sometimes scanner-less
- also can not do left recursion
- Uses the first option that matches
- Error recovery and proper error reporting very hard
- ANTLR4 much more efficient

Comparing to GLR

a generalized LR parser

- does breadth-first search when there is a conflict in the LR table
- can deliver all possible derivation on ambiguous input
- debugging on the declarative specification
  - no need to debug state machine
- can do indirect left recursion

Comparing to GLL

a generalized LL parser

- mapping to recursive descent parser not obvious (not possible?)
- can do indirect left recursion

Wrap-Up

- ANTLR 4 is easy to use
- You can use listeners or visitors for actions
- ANTLR 4 helps you solve **practical** problems

Why is ANTLR4 called "Honey Badger"?

- The *amimal* "Honey Badger" is the most fearless animal of all
- ANTLR4 named after it
  - http://www.youtube.com/watch?v=4r7wHMg5Yjg
- It takes any grammar you give it. “It just doesn’t give a damn.”

Resources

- Terence Parr speaking about ANTLR4
  - http://vimeo.com/m/59285751
- ANTLR4-Website
  - http://www.antlr.org
- Book
  - http://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference
- ANTLRWorks
  - http://tunnelvisionlabs.com/products/demo/antlrworks
- Examples of this talk on github
  - https://github.com/DJCordhose/antlr4-sandbox

Questions / Discussion

Oliver Zeigermann
zeigermann.eu

Java example and some illustrations provided as a courtesy of Terence Parr