Breizh C@mp
Mix de technologies

The Art of Parsing

#parsing

Evgeny Mandrikov - @_godin_
Nicolas Peru - @benzonico

Disclaimer

// TODO: don't forget to add huge disclaimer that all opinions hereinbelow are our own and not our employer (they wish they had them)

Evgeny Mandrikov
@_godin_

Nicolas Peru
@benzonico

sonarsource™

What is the plan?

Why

- javac, GCC, Roslyn are hand-written
- do we use parser-generators?

Together we will implement parser for

- arithmetic expressions
- constructions from Java
- C++ ;)

I want to create a parser, so easy...

- Yacc
- JavaCC
- CUP
- Bison
- ANTLR
- …

… or not

JLS7

Chapter 18. Syntax

This chapter presents a grammar for the Java programming language.

The grammar presented piecemeal in the preceding chapters (§2.3) is much better for exposition, but it is not well suited as a basis for a parser. The grammar presented in this chapter is the basis for the reference implementation. Note that it is not an LL(1) grammar, though in many cases it minimizes the necessary look ahead.

JLS8

Chapter 19. Syntax

This chapter repeats the syntactic grammar given in Chapters 4, 6-10, 14, and 15, as well as key parts of the lexical grammar from Chapter 3, using the notation from §2.4.

OpenJDK

OpenJDK FAQ
Installing
Contributing
Sponsoring
Developers' Guide

Mailing lists
IRC · Wiki

Bylaws · Census
Legal

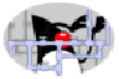
JEP Process

Compiler Grammar

Introduction

The goal of this Project is to develop an experimental version of the javac compiler based upon a grammar written in ANTLR.

The parser that is currently in the javac compiler is a hand-written LALR parser. It is somewhat fragile, and is not always easy to extend when working on potential new language features. In addition, it is not well-suited for analysis, such as comparison against the grammar rules in the Java Language Specification (JLS).



OpenJDK / compiler-grammar
log

log
graph
tags
bookmarks
branches

changeset
browse

age	author	description
2008-10-03	xdono	6754988: Update copyright year default tip
2008-09-25	xdono	Added tag jdk7-b36 for changeset 4b4f5fea8d7d
2008-09-17	xdono	Merge jdk7-b36
2008-09-17	ohair	6724787: OpenJDK README-builds.html

Answer is

42

Pill of theory

productions

$$\text{NUM} \rightarrow 42$$

nonterminals

terminals (tokens)

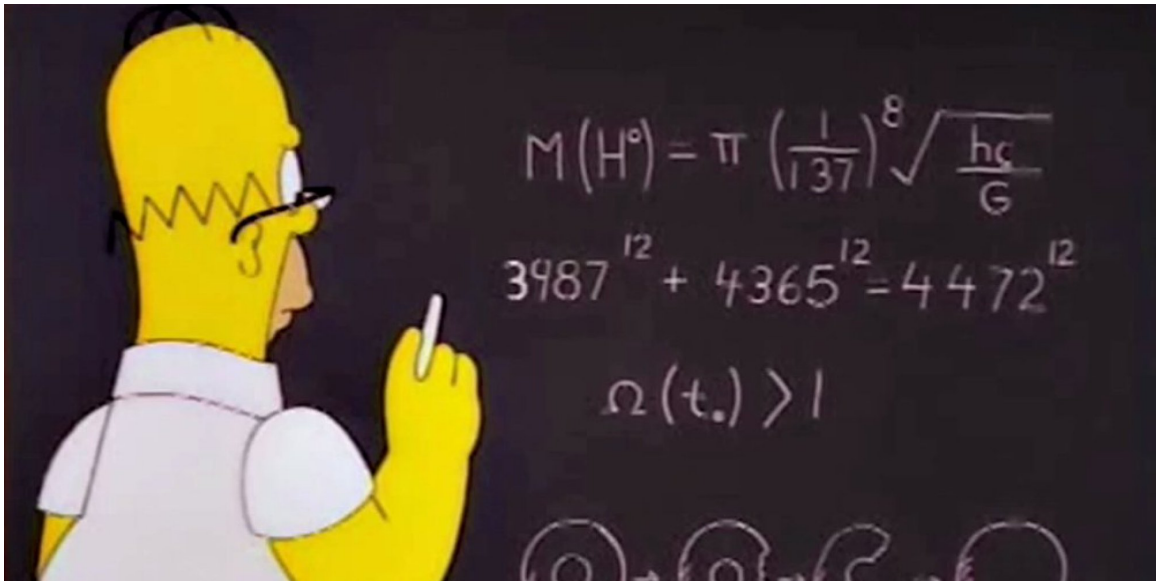
Pill of theory

alternatives

```
DIGIT→0
     | ...
     | 9
```

Arithmetic expressions

4 − 3 − 2 = ?



Arithmetic expressions

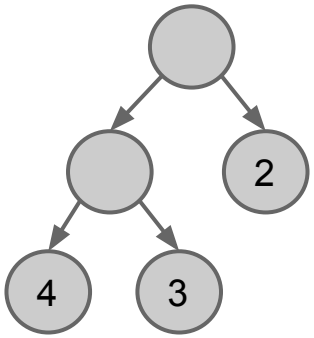
```
4 - 3 - 2 = ?
```

```
expr → expr - expr
     | NUM
```

Arithmetic expressions

```
4 - 3 - 2 = ?
```

```
(4 - 3)- 2 = -1
```



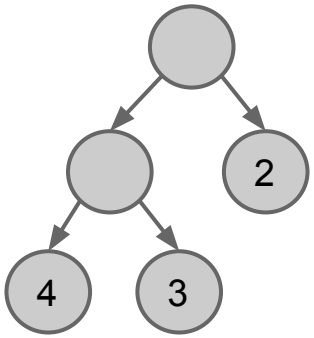

```
expr → expr - expr
     | NUM
```

Arithmetic expressions

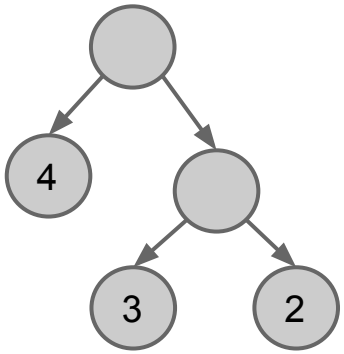
```
4 - 3 - 2 = ?
```

```
expr → expr - expr
     | NUM
```

```
(4 - 3)- 2 = -1
```




```
4 -(3 - 2)= 3
```

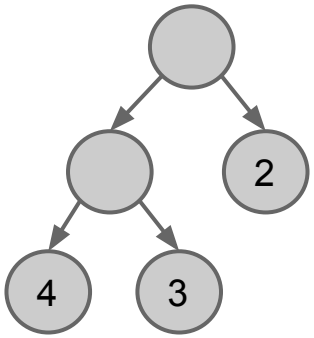


Arithmetic expressions

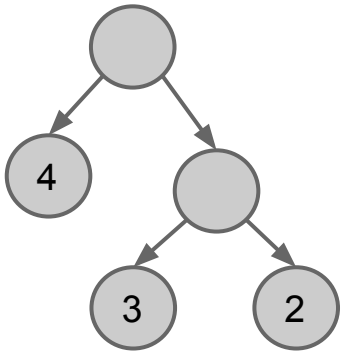
```
4 - 3 - 2 = ?
```

```
expr → expr - NUM
     | NUM
expr → expr - expr
     | NUM
expr → NUM - expr
     | NUM
```

(4 - 3) - 2 = **-1**



4 - (3 - 2) = **3**

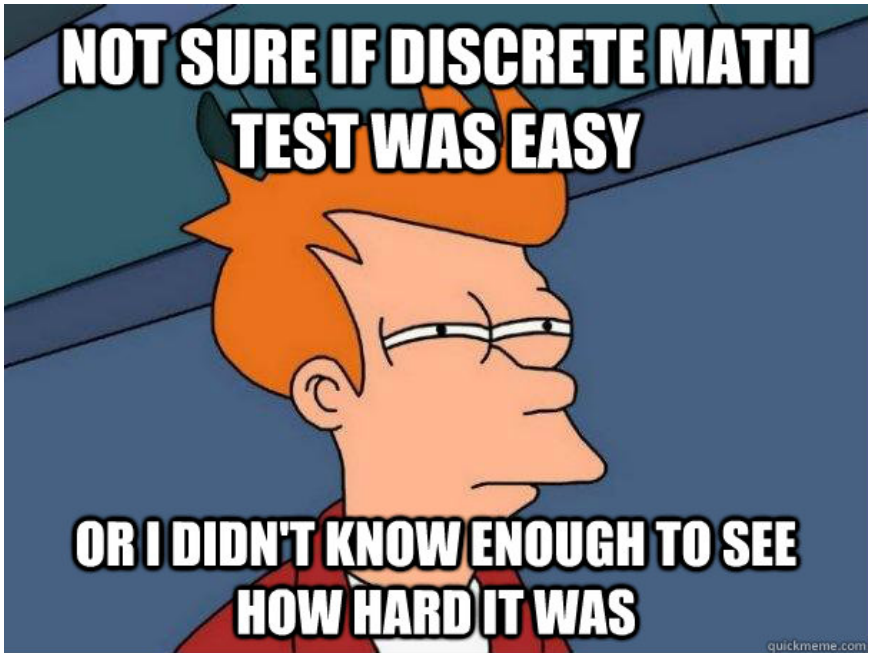


Arithmetic expressions

```
expr → expr - NUM
     | NUM

/* expr -> NUM{-NUM}*/
int expr() {
   int r = num();
   while (token == '-')
      r = r - num();
   return r;
}
```

Arithmetic expressions

4 - 3 * 2 = ?



Arithmetic expressions

```
4 - 3 * 2 = ?
```

```
expr → expr - NUM
     | expr * NUM
     | NUM
```

Arithmetic expressions

```
(4 - 3)* 2 = 2
```

```
expr → expr - NUM
     | expr * NUM
     | NUM
```

Arithmetic expressions

```
4 -(3 * 2)= -2
```

```
subs → subs - mult
     | mult
mult → mult * NUM
     | NUM
```

Arithmetic expressions

```
int subs() {
  int r = mult();
  while (token == '-')
    r = r - mult();
  return r;
}
int mult() {
  int r = num();
  while (token == '*')
    r = r * num();
  return r;
}
```

LL(1)

- back to 1969
- one token lookahead
- no left-recursion, no ambiguities
- trivial

The real deal

```
          expr → qualified-id
                | method-call
                | assignment
 method-call → qualified-id()
  assignment → qualified-id = expr
qualified-id → id
                | id . qualified-id
```

LL(1) way

```
          expr → qualified-id expr'
          expr'→
               | method-call
               | assignment
 method-call'→ ()
  assignment'→ = expr
qualified-id → id
               | id . qualified-id
```

LL(1) way

```
int expr() {
  String i = qualified_id();
  if (token == '(')
    return method_call(i);
  else if (token == '=')
    return assignment(i);
  else
    return field_access(i);
}
```

Reality

http://hg.openjdk.java.net/jdk8/jdk8/langtools/.../JavacParser.java

```
90      // Because of javac's limited lookahead, some contexts are ambiguous in
91      // the presence of type annotations even though they are not ambiguous
92      // in the absence of type annotations.  Consider this code:
93      //   void m(String [] m) { }
94      //   void m(String ... m) { }
95      // After parsing "String", javac calls bracketsOpt which immediately
96      // returns if the next character is not '['.  Similarly, javac can see
97      // if the next token is ... and in that case parse an ellipsis.  But in
98      // the presence of type annotations:
99      //   void m(String @A [] m) { }
100     //   void m(String @A ... m) { }
101     // no finite lookahead is enough to determine whether to read array
102     // levels or an ellipsis.  Furthermore, if you call bracketsOpt, then
103     // bracketsOpt first reads all the leading annotations and only then
104     // discovers that it needs to fail.  bracketsOpt needs a way to push
105     // back the extra annotations that it read.  (But, bracketsOpt should
106     // not *always* be allowed to push back extra annotations that it finds
107     // -- in most contexts, any such extra annotation is an error.
108     //
109     // The following two variables permit type annotations that have
110     // already been read to be stored for later use.  Alternate
111     // implementations are possible but would cause much larger changes to
112     // the parser.
```

Hand-written

- based on LL(1)
- precise error-reporting and recovery
- best performances
- maintenance hell

LL(1)

- back to 1969
- one token lookahead
- no left-recursion, no ambiguities
- trivial
- major grammar changes
- on steroids as in JavaCC usable for real languages

Better way - ordered choice

```
int expr() {
  try { return qualified_id(); }
  catch (PE e1) {
    try { return method_call(); }
    catch (PE e2) {
      return assignment();
    }
  }
}
```

Better way - ordered choice

```
int expr() {
  try { return qualified_id(); }
  catch (PE e1) {
    try { return method_call(); }
    catch (PE e2) {
      return assignment();
    }
  }
}
```

Better way - ordered choice

```
int expr() {
  try { return assignment(); }
  catch (PE e1) {
    try { return method_call(); }
    catch (PE e2) {
      return qualified_id();
    }
  }
}
```

Parsing Expression Grammars

- 2002
- ordered choice (operator “/” instead of “|”)
- backtracking
- no left-recursion, no ambiguities
- trivial
- fewer grammar changes
- usable for real languages
- dead-end for C/C++

SSLR DSL

```
enum Nonterminals {
  METHOD_CALL, ASSIGNMENT, ID
}
void defineGrammar() {
  rule(EXPR).is(
    firstOf(
      METHOD_CALL,
      ASSIGNMENT,
      ID));
}
```

Tea break?

Quiz

```
if (false)
   if (true)
      System.out.println(“foo”);
else
   System.out.println(“bar”);
```

“Dangling else”

```
if (false)
   if (true)
     System.out.println(“foo”);
   else
     System.out.println(“bar”);
```

```
if-stmt → if (cond) stmt else stmt
        / if (cond) stmt
```

Java is awesome

```
(A)*B
```

C++ all the pains of the world

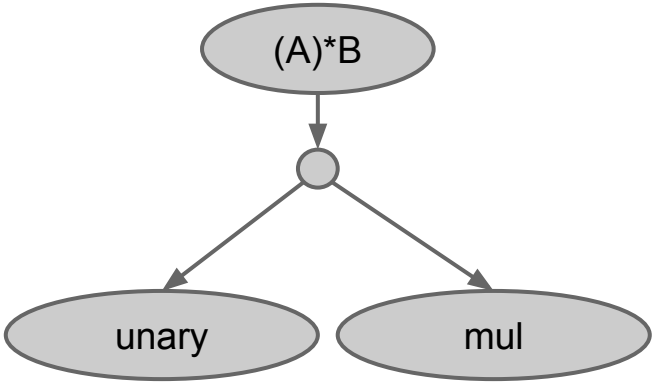
```
int A, B;
(A)*B // multiplication

int *B;
typedef int A;
(A)*B // cast to type ‘A’
      // of dereference of ‘B’
```

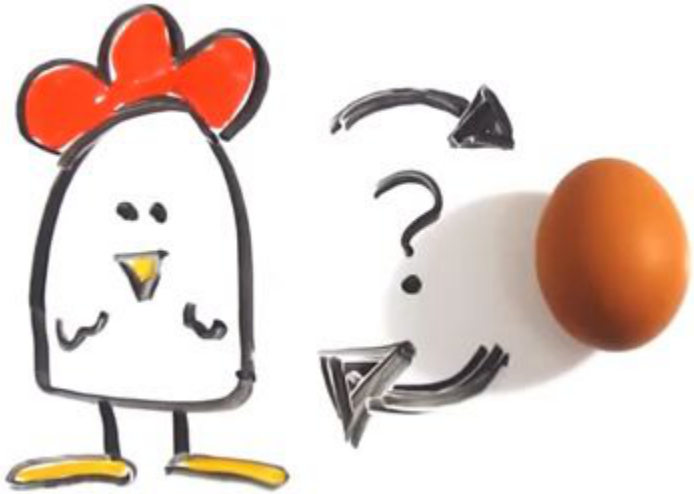
“Piece of cake!”

```
rule(MUL).is(
  UNARY, zeroOrMore('*', UNARY)
);
rule(UNARY).is(firstOf(
  sequence('(', TYPE, ')', UNARY),
  PRIMARY,
  sequence('*', UNARY)
));
rule(PRIMARY).is(firstOf(
  sequence('(', EXPR, ')'),
  ID
));
```

Chicken and egg ~~problem~~

```
    mul → mul * unary
        | unary
  unary → ( type ) unary
        | * unary
        | primary
primary → ( expr )
        | id
```

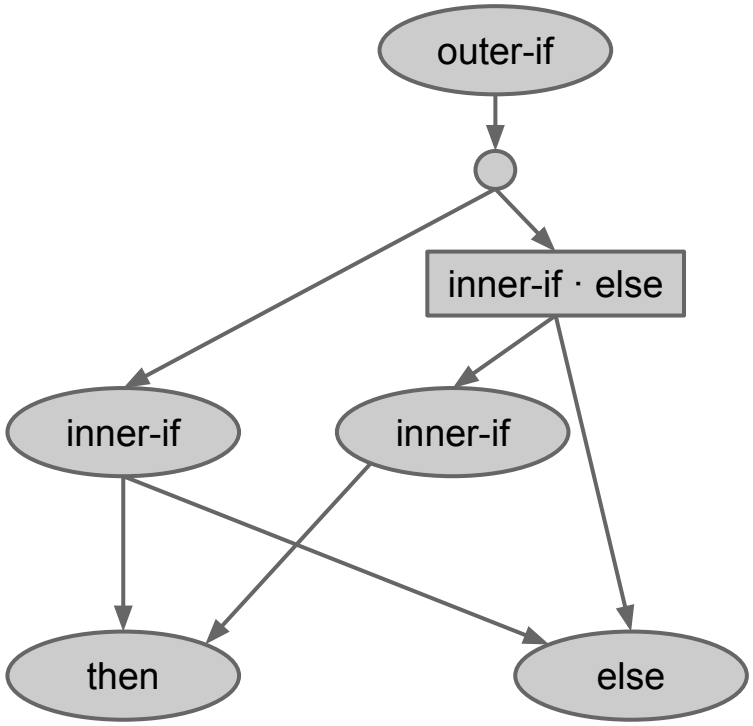




Generalized parsers

- Earley (1968)
  - slow
- GLR (1984)
  - complex
- GLL (2010)
  - ...

Back to the ~~future~~ “dangling else”



Generalized LL

- 2010
- no grammar left behind
  - left-recursive
  - ambiguous
- simpler than GLR, faster than Earley
- reasonable performances
- the only clean choice for C/C++
- only “academic” tools for now ;)

There is no silver bullet

- LL(1)
- Hand-written
- PEG
- GLL

Questions?