The Quest for the One True Parser

Terence Parr
The ANTLR guy
University of San Francisco

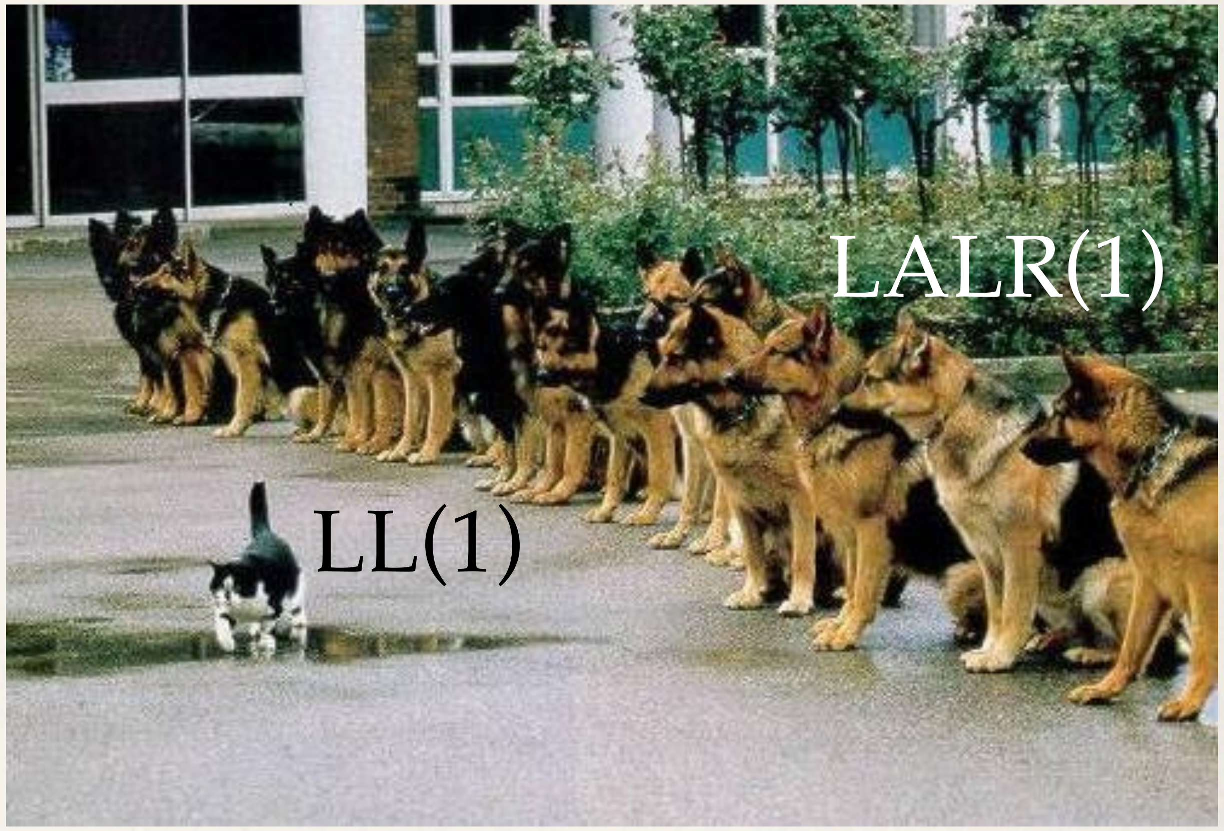
November 4, 2014

Am I behind the times? Buzzword Compliance

- What's a PEG and do I need one? Am I a packrat?
- Should I care about context-sensitive parsing?
- Do we still need the distinction between the tokenizer and the parser?
- Parser Combinators do what, exactly?
- Should I be using Generalized-LR (GLR)?
- Can I parse tree data structures not just token streams?
- How is ANTLR 4's ALL(*) like the Honey Badger?

Parr-t Like It’s 1989

- 25 years ago, LALR/yacc/bison reigned supreme in tools
- In ~1989 you either used yacc or you built parsers by hand
- I didn’t grok **yacc** with its state machines and shift/reduce conflicts



- I set out to create a parser generator that generated what I wrote by hand: recursive-descent parsers
- The quest eventually led to some useful innovations

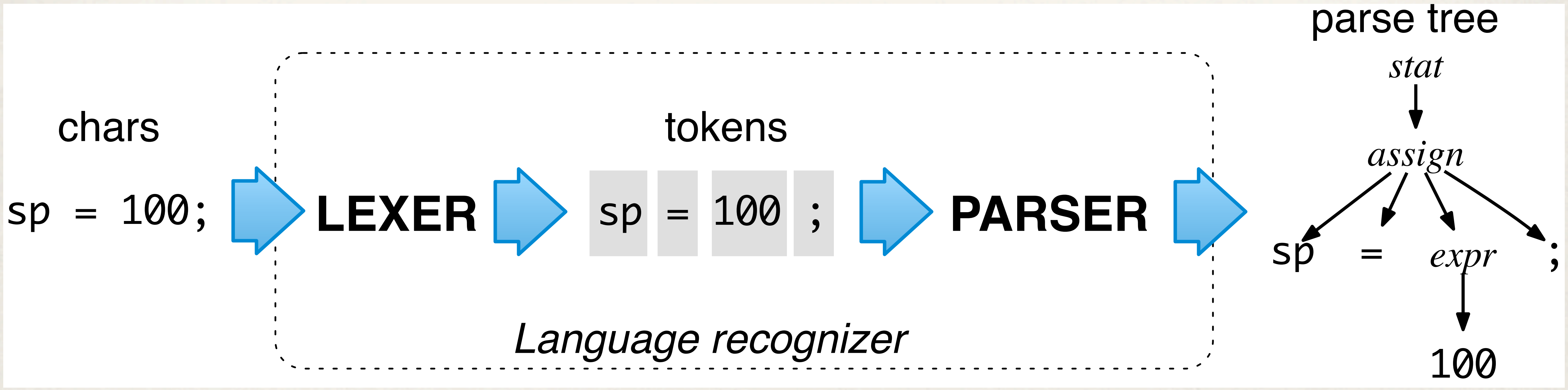
The Players

- Warring religious factions
  - LL-based, “top-down,” recursive-descent, “hand built”, LL(1)
  - LR-based, “bottom-up,” yacc, LALR(1)
- Two other camps; researchers working on:
  - Increasing efficiency of general algorithms Earley, GLL, GLR, Elkhound, …
  - Increasing power of top-down parsers LL(k), predicates, PEG, LL(*), GLL, ALL(*)

Some “Lex Education”
Grammars, parsers, and trees oh my!

Parser Information Flow

- The parser feeds off of tokens from the lexer, which feeds off of a char stream, to check syntax (language membership)
- We often want to build a parse tree that records how input matched



Grammar Meta-languages (DSLs)

- A grammar is a set of rules that describe a language
- A language is just a set of valid sentences
- Each rule has a name and set of one or more alternative productions
- Most tools use a DSL that looks like this:

```
stat: ‘if’ expr ‘then’ stat (‘else’ stat)?
    | ID ‘=’ expr
    | ‘return’ expr
    ;
expr: expr ‘*’ expr
    | expr ‘+’ expr
    | ID
    | INT
    ;
```

Grammar Conditions

- *Left-recursive grammars* have rules that reference rules already in flight; LR ***loves*** this, LL ***hates*** this!

```
expr : expr ‘*’ expr
     | INT
     ;
```

- *Recursive-descent parsers* get infinite recursive loops
- Ambiguous grammars can match an input phrase in more than one way. In C, **i*j** could be an expression or declaration like **FILE *f**.
- GLR was designed for ambiguous grammars; *“Police help dog bite victim*.” LL, LR resolve ambiguities at parse-time, picking one path

Recursive-descent functions Top-down, LL(k) for k ≥ 1

```
assign
    :    ID ‘=’ expr ‘;’
    ;
```

```
expr
    :    INT | STRING
    ;
```

```
void assign() {
    match(ID);
    match(‘=’);
    expr();
    match(‘;’);
}
```

```
void expr() {
    switch ( curtoken.getType() ) {
        case INT :
            match(INT);
            break;
        case STRING :
            match(STRING);
            break;
        default : error;
    }
}
```

Bottom-up, LR(k)

- Yacc is LR-based: LALR(1)
- LR recognizers are bottom-up recognizers; they match from leaves of parse tree towards starting rule at the root whereas LL starts at the root (top-down) and is goal oriented.
- LR consumes tokens until it recognizes an alternative, one that will ultimately lead to a successful parse of the input. At input **int x;** parser reduces the input to a **decl ;** and then reduces to **stat**

```
stat : decl ‘;’ ;
decl : ‘int’ ID
     | ‘int’ ID ‘=’ expr
     ;
```

Should I be using GLR?

Generalized LR (GLR)

- Accepts all grammars since designed to handle ambiguous languages

```
stat : decl ‘;’
     | decl ‘=’ 0 ‘;’
decl : ‘int’ ID
     | ‘int’ ID ‘=’ expr
     ;
```

Matches **int x = 0;** via alternative 1 or 2 of decl

int x = 0; ➞ decl = 0 ; ➞ stat

Or, **int x = 0; ➞ decl ; ➞ stat**

- “Forks” subparsers to pursue all possible paths emanating from LR states with conflicts

- Merges all successful parses into parse “forest”

Issues with GLR

- Must disambiguate parse forests even for if-then-else ambiguity, requiring extra time, space, machinery
- GLR performance is unpredictable in time and space
- Grammar actions don't mix with parser speculation/ambiguities
- Semantic predicates less useful w/o side-effecting user actions
- Without side effects, actions must buffer data for all interpretations in immutable data structures or provide undo actions

What's a PEG and do I need one? Am I a packrat?

Parser Expression Grammars (PEGs)

- PEGs are grammars based upon LL with explicitly-ordered alternatives

```
stat : decl / expr ;
```

```
decl : ‘int’ ID
     / ‘int’ ID ‘=’ expr
     ;
```

- Attempt alternatives in order specified; first alternative to match, wins
- Unambiguous by definition and PEGs accept all non-left recursive grammars; **T(i)** in C++ matches as **decl** not function call in **stat**
- Packrat parsers record partial results to avoid reparsing subphrases

Square PEG, round hole?

- PEGs might not do what you want; 2nd alternative of **decl** never matches!

```
decl : ‘int’ ID
     / ‘int’ ID ‘=’ expr
     ;
```

<== Yes, I’m deadcode

- PEGs are not great at error reporting and recovery; errors detected at the end of file
- Can’t execute arbitrary actions during the parse; always speculating
- Without of side-effecting actions, predicates aren’t as useful
- Hard to debug nested backtracking

Parser Combinators do what, exactly?

Higher-order functions as building blocks

- Use programming language itself rather than separate grammar DSL, avoiding a build step to generate code from grammar
    - Alternation **b | c | d** becomes **Parsers.or(b, c, d)**
    - Sequence **bcd** becomes **Parsers.sequence(b,c,d)**
    - Has higher-order rules; can pass rules to rules

```
list[el] : <el> (‘,’ <el>)* ;
```

- Essentially equivalent to an inline PEG or packrat parser, with same issues
- ANTLR also has an interpreter, btw, to avoid build step

Do we still need the distinction between the tokenizer and the parser?

Scannerless Parsing

GLR and PEGs are typically scannerless

- Tokenizing is natural; we do it. *"Humuhumunukunukuapua'a have a diamond-shaped body with armor-like scales."*
- Tokenizing is efficient and processing tokens is convenient
- But... scannerless parsing supports mixed languages like C+SQL:

```
int next = select ID from users where name='Raj'+1;
int from = 1, select = 2;
int x = select * from;
```

- That's pretty cool and supports modular grammars since we can combine grammar pieces w/o fear that combined input won't tokenize properly
- Easy to fake if parser is strong enough: just make each char a token!

Scannerless Grammars are Quirky

- Must test for white space explicitly and frequently

```
prog:   ws? (var|func)+ EOF ;
plus:   '+' ws? ;
```

- Distinguishing between keywords and identifiers is messy; e.g., **int** or **int[** versus **interest** or **int9**

```
kint:   {notLetterOrDigit(4)}? 'i' 'n' 't' ws? ;
id  :   letter+ {!keywords.contains($text)}? ws? ;
```

Should I care about context-sensitive parsing?

Predicated Parsing

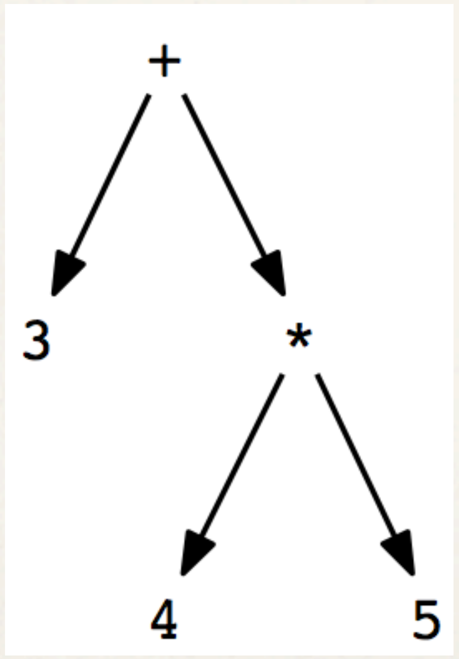
- Context-sensitive rules are viable per a runtime test called a semantic predicate; the expression language depends on the tool

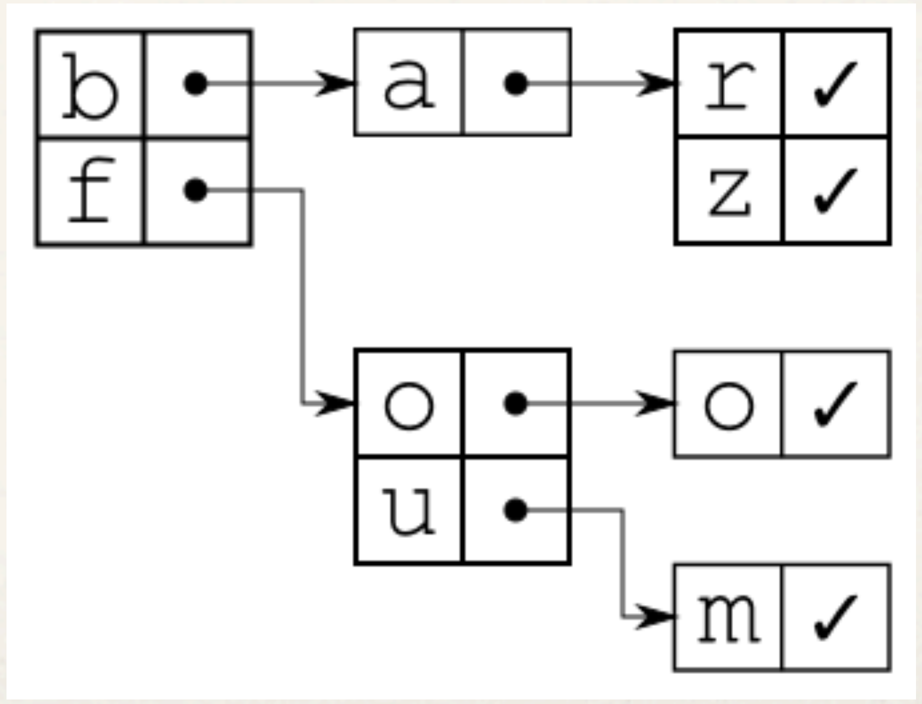
- Disambiguating **a(i)** and **f(x)** in Fortran requires symbol table information about **a**,**f**

```
expr: array
    | call
    ;
array : {isArray(token)}? ID ‘(‘ expr ‘)’ ;
call  : {isFunc(token)}?  ID ‘(‘ expr ‘)’ ;
```

- Or, can build a “parse forest” and disambiguate after the parse, but that can be inefficient in time and space

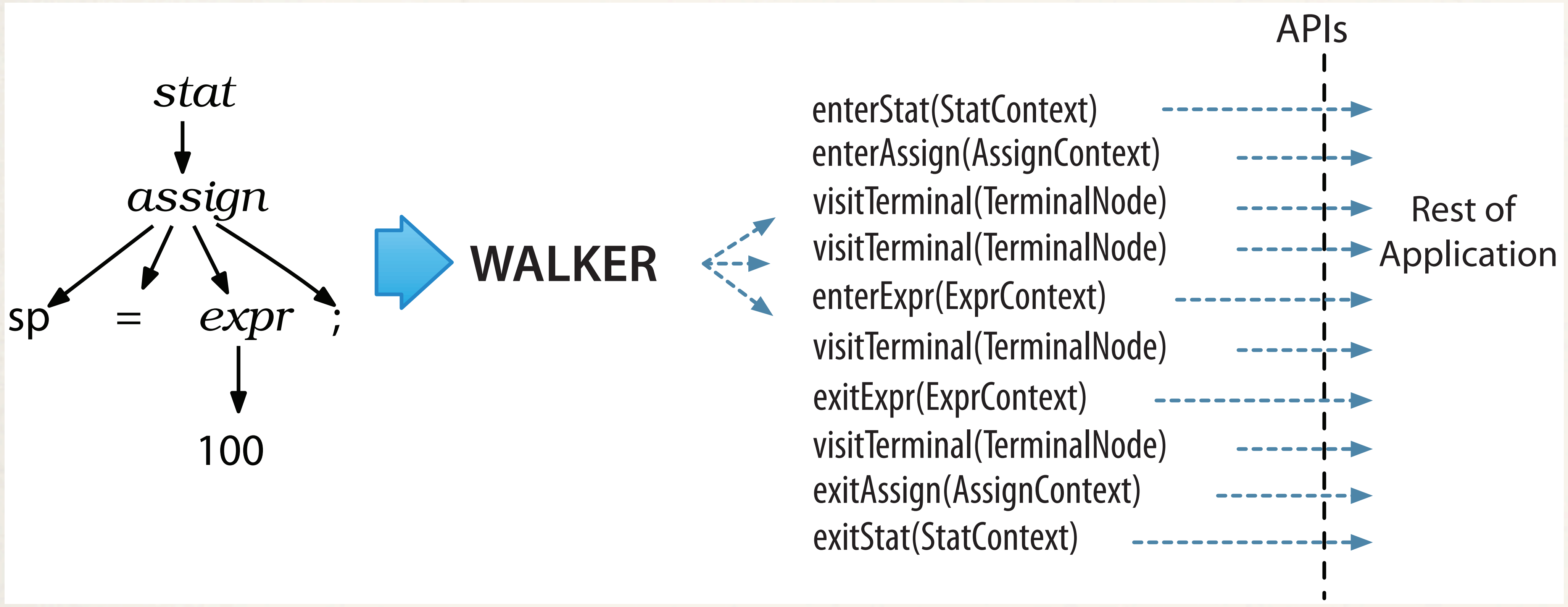
Can I parse data structures like trees? (Are you TRIE-curious?)





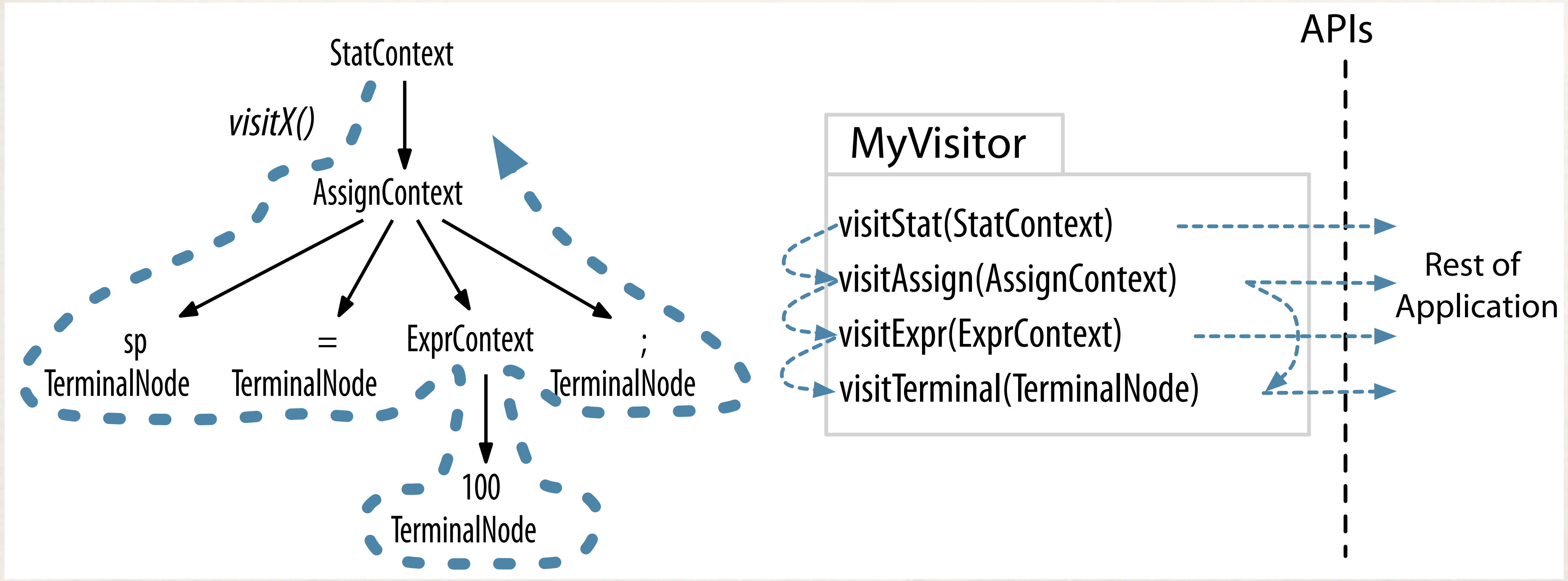
Yes, But First...

Imperative processing of parse trees



stat
assign
sp
=
expr
;
100
WALKER
APIs
enterStat(StatContext)
enterAssign(AssignContext)
visitTerminal(TerminalNode)
visitTerminal(TerminalNode)
enterExpr(ExprContext)
visitTerminal(TerminalNode)
exitExpr(ExprContext)
visitTerminal(TerminalNode)
exitAssign(AssignContext)
exitStat(StatContext)
Rest of Application

Listener



Visitor

ANTLR XPath, pattern matching

Declarative+imperative

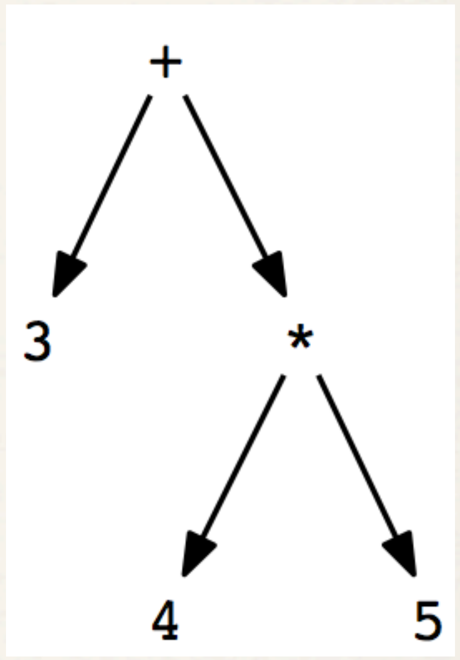
`/prog/func/'def'`	Find all def literal kids of func kid of prog
`/prog/func`	Find all funcs under prog at root

```
// "Find all initialized int local variables (Java)"
ParserRuleContext tree = parser.compilationUnit(); // parse
String xpath = "//blockStatement/*"; // get children of blockStatement
String treePattern = "int <Identifier> = <expression>;";

ParseTreePattern p =
    parser.compileParseTreePattern(treePattern,
        ExprParser.RULE_localVariableDeclarationStatement);
List<ParseTreeMatch> matches = p.findAll(tree, xpath);
matches.get(0).get("expression"); // get 1st init expr subtree

System.out.println(matches);
```

Tree Grammars

- Specify structure of tree declaratively with grammar DSL; translated to parser
- SORCERER in 1994 then into ANTLR directly
- ANTLR 3 could rewrite trees, invoke StringTemplates

```
expr
   | ^('+' expr expr)
   | ^('*' expr expr)
   |  INT
   ;
```



```
expr returns [int value]
   : ^('+' a=expr b=expr) {$value = a+b;}
   | ^('-' a=expr b=expr) {$value = a-b;}
   | ^('*' a=expr b=expr) {$value = a*b;}
   |  INT
      {$value = Integer.parseInt($INT.text);}
   ;
```

OMeta

- "A programming language whose control structure is based on PEGs" with pattern matching like Haskell/LISP
- Can process streams of arbitrary datatypes and nested lists (trees)
- Higher-order rules; can pass rule arg to another rule (coolness alert)
- Actions written in host language not OMeta itself (coder must watch out for action side-effects during speculation/backtracking)
- Other pattern matching, transformation DSLs: TXL, Stratego/XT, TOM (Java superset), Rascal Metaprogramming Language, ...

Well, should I use tree grammars?

- Tree grammars caused lots of code dup; copy grammar for each tree pass, just with different actions
- Change to tree structure requires change to each tree grammar
- Actions cause trouble for grammar inheritance; fragile base-class problem
- Ended up being a maintenance hassle so ANTLR 4 dropped them in favor of listeners/visitors/xpath/tree-patterns imperative+declarative style

ANTLR 4's ALL(*) Parser
The Nasty-Ass Honey Badger

It takes any grammar you give it. "It just doesn't give a shit."

Yes, even left-recursive rules! (except indirect left-recursive rules)

What is ALL(*)?

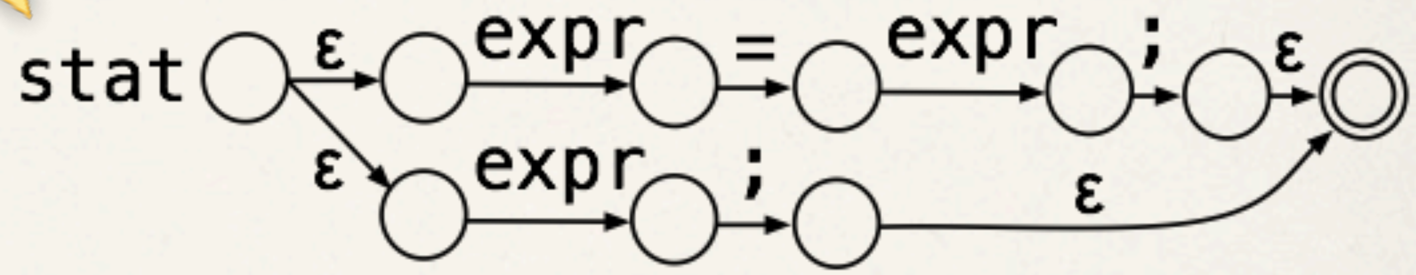
- ALL(*) combines simplicity, efficiency, and predictability of top-down LL(k) parsers with power of GLR-like mechanism to make decisions
- **Key innovation**: shift grammar analysis to parse-time, caching results
- **Launch subparsers** that scan ahead to determine which alternative (path) will ultimately lead to successful parse; warms up like a JIT
- All but one subparser die off, yielding unique prediction (unless ambiguous phrase or syntax error)
- Analogy:

```
ticket_line : PEOPLE+ BORAT
            | PEOPLE+ THE_BODY_GUARD
            ;
```

Adaptive LL(*): ALL(*)

```
// ALL(*) but non-LL(k), non-LL(*) grammar
stat: expr '=' expr ';'
    | expr ';'
    ;
```

ATN

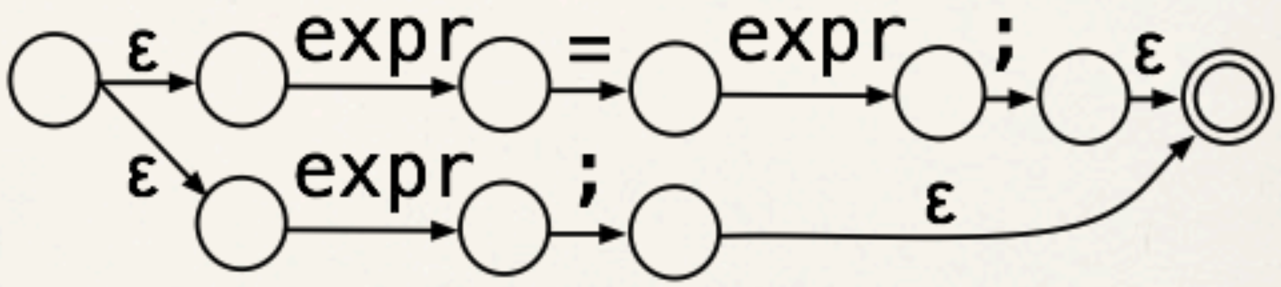


Recursive-descent Parser

```
void stat() { // parse according to rule stat
    switch ( adaptivePredict("stat", call stack) ) {
        case 1 : // predict production 1
            expr(); match('='); expr(); match(';');
            break;
        case 2 : // predict production 2
            expr(); match(';');
            break;
    }
}
```

Incremental DFA construction

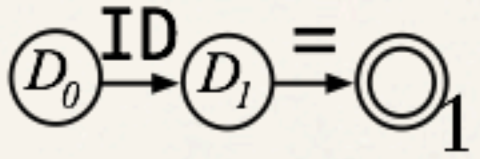
stat's ATN



Multiple threads running >1 parser instance build the DFA faster

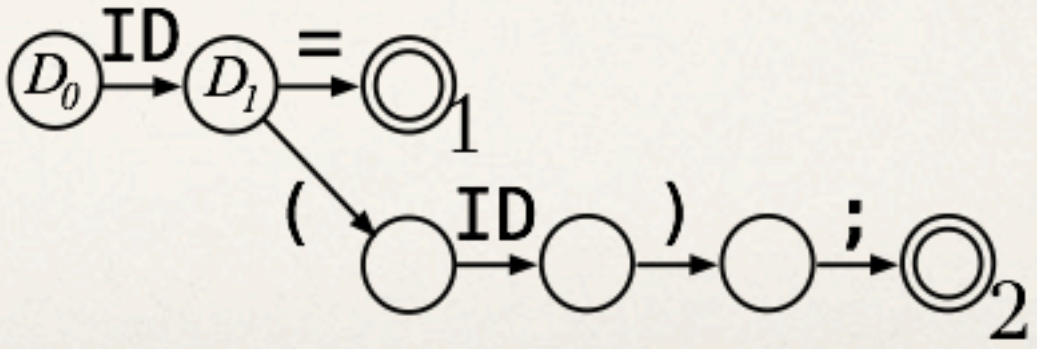
stat's DFA after input

```
x=y;
```

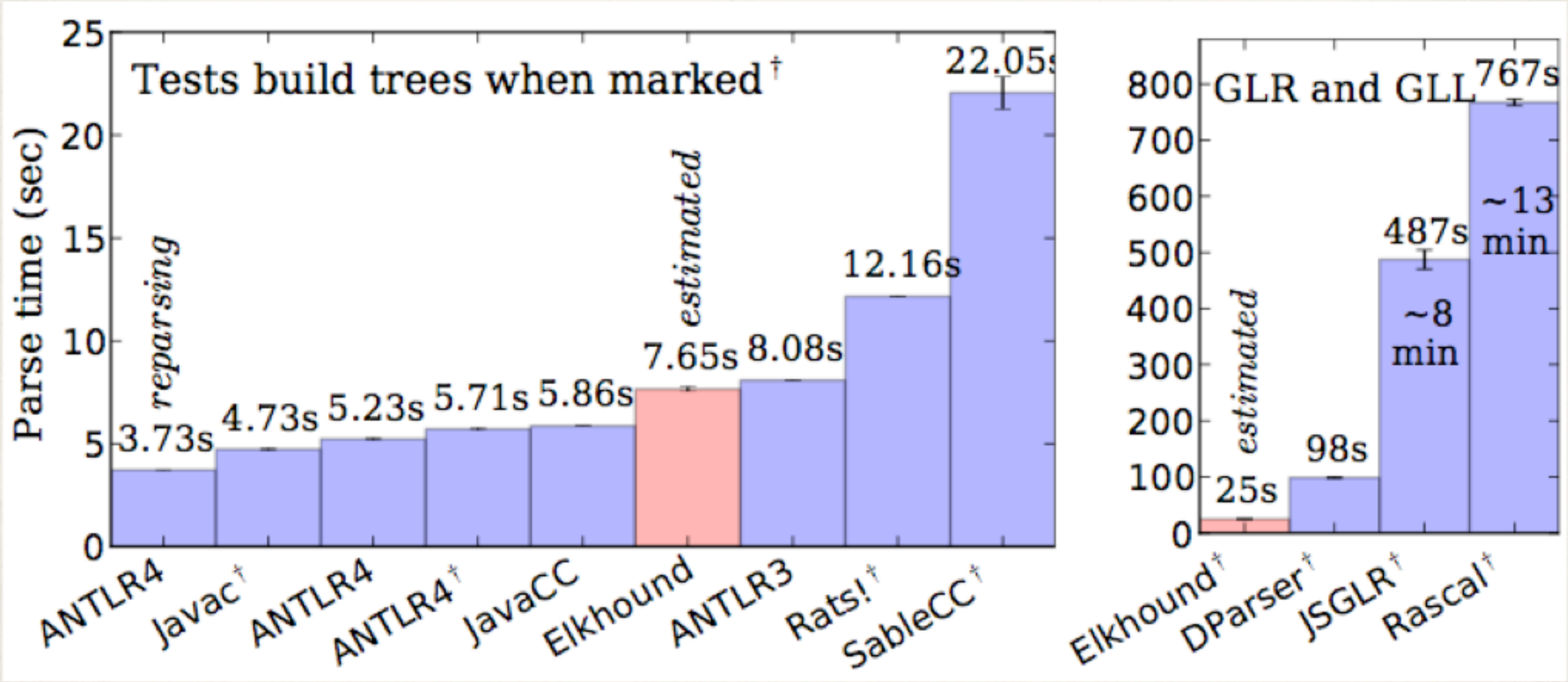


stat's DFA after both

```
x=y;
f(x);
```



Parsing 132M, 12920 Java files



Summary

- You want the most general parser you can get, as long as you don't pay for it with poor performance; ALL(*) is the sweet spot
- GLR is too unpredictable in performance; general parsers like GLL/GLR that support all input interpretations will never compete with more deterministic strategies
- Semantic predicates are rarely needed but a godsend when you need them!
- Declarative tree matching is good, but just for searching; tree grammars with embedded actions, not so good. I like mixed declarative+imperative style
- Scannerless grammars are cool as hell; useful for modular grammars and mixed languages but less convenient to build grammars and process artifacts
- PEGs are great for smaller languages; at a disadvantage for action execution, semantic predicates, error handling. OMeta has same issues

Extras in case...

Parsing Innovation Timeline

(Not exhaustive, obviously)

1985 GLR

1992 GLR fixed to be completely general (looped on cyclic grammars)

1993 Linear approximate lookahead makes LL(k) more attractive

1994 Semantic and syntactic predicates for LL(k)

1997 SGLR scannerless parsing

2002 Packrat parsing

2004 Elkhound: novel combination of GLR and LR parsing

2004 PEGs (formalized speculative, ordered-alternative grammars)

2010 GLL (No parser generator available; Rascal uses GLL variant)

2011 LL(*) (2007 tool released)

2014 ALL(*) (2012 tool released)

Frustration Led To…

```
decl: ‘int’ ID ‘;’
    | ‘int’ ID ‘=’ expr ‘;’
```

First this pissed me off:
Led to LL(k) for k>1

```
expr: ID ‘(‘ expr ‘)’ // array index
    | ID ‘(‘ expr ‘)’ // func call
```

Then this:
semantic predicates

```
stat: decl | expr
```

Syntactic predicates

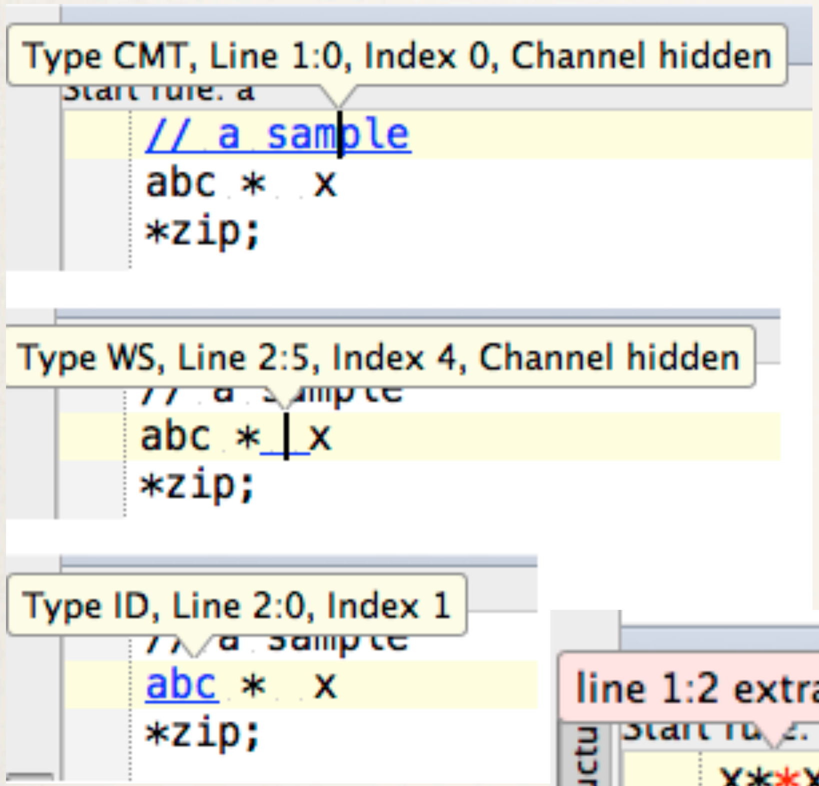
```
decl: modifier* type ID ‘;’
    | modifier* type ID ‘=’ expr ‘;’
```

LL(*)

```
stat: expr ’=’ expr ’;’
    | expr ’;’
```

ALL(*)

Intellij Plug-in



Type CMT, Line 1:0, Index 0, Channel hidden

```
// a sample
abc *  x
*zip;
```

Type WS, Line 2:5, Index 4, Channel hidden

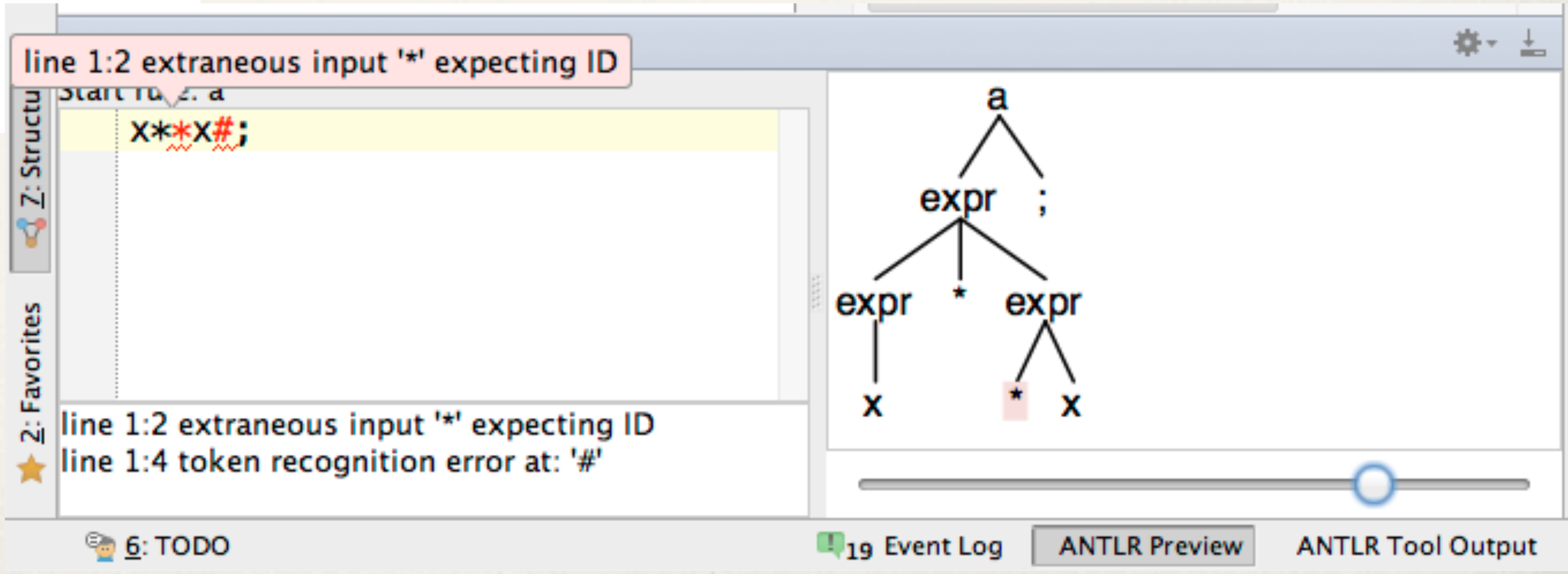
```
abc * x
*zip;
```

Type ID, Line 2:0, Index 1

```
abc *  x
*zip;
```

Answers how was this input tokenized?

Visualizes parse tree live via parser interpreter



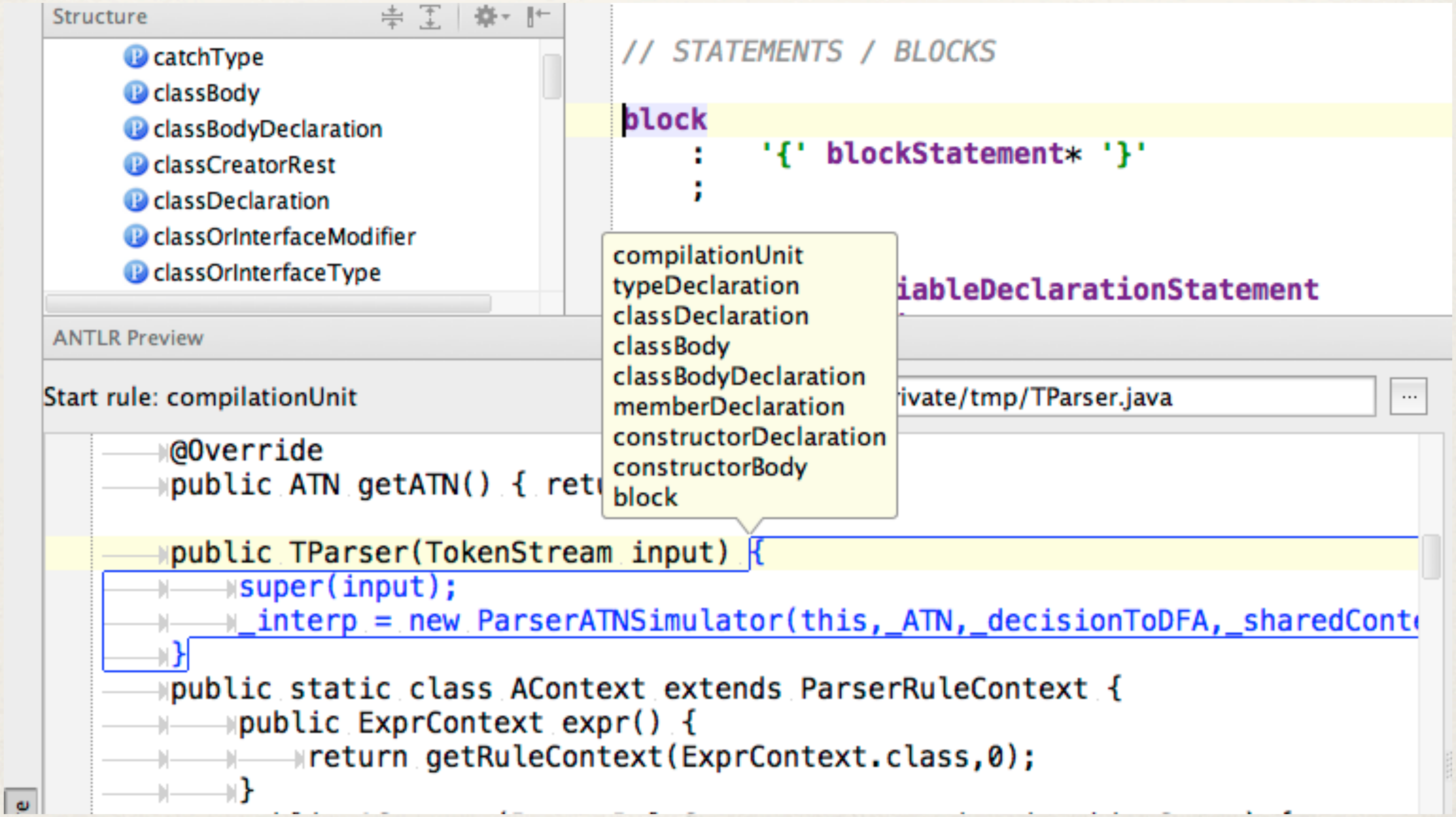
line 1:2 extraneous input '*' expecting ID

```
x**x#;
```

```
line 1:2 extraneous input '*' expecting ID
line 1:4 token recognition error at: '#'
```

How was that phrase recognized?

Critical feature for large input/grammars



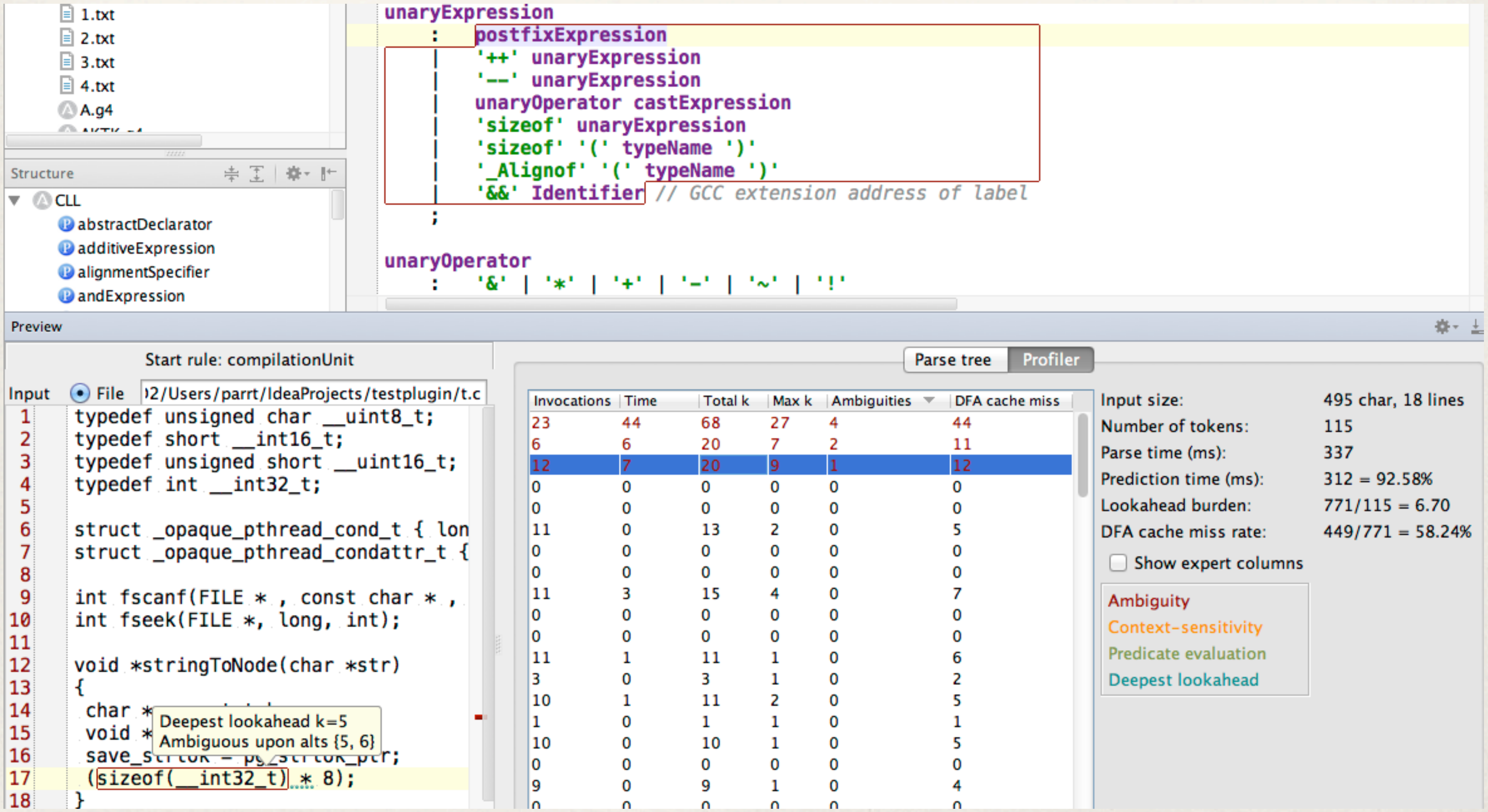
catchType
classBody
classBodyDeclaration
classCreatorRest
classDeclaration
classOrInterfaceModifier
classOrInterfaceType

```
// STATEMENTS / BLOCKS

block
    :   '{' blockStatement* '}'
    ;

                       iableDeclarationStatement
```

compilationUnit
typeDeclaration
classDeclaration
classBody
classBodyDeclaration
memberDeclaration
constructorDeclaration
constructorBody
block

```
    @Override
    public ATN getATN() { retu

    public TParser(TokenStream input) {
        super(input);
        _interp = new ParserATNSimulator(this,_ATN,_decisionToDFA,_sharedConte
    }
    public static class AContext extends ParserRuleContext {
        public ExprContext expr() {
            return getRuleContext(ExprContext.class,0);
        }
```

Identify ambiguities, lookahead depth to optimize



1.txt
2.txt
3.txt
4.txt
A.g4

Structure

CLL
abstractDeclarator
additiveExpression
alignmentSpecifier
andExpression

```
unaryExpression
    :   postfixExpression
    |   '++' unaryExpression
    |   '--' unaryExpression
    |   unaryOperator castExpression
    |   'sizeof' unaryExpression
    |   'sizeof' '(' typeName ')'
    |   '_Alignof' '(' typeName ')'
    |   '&&' Identifier // GCC extension address of label
    ;

unaryOperator
    :   '&' | '*' | '+' | '-' | '~' | '!'
```

Preview

Start rule: compilationUnit

Input ● File ›2/Users/parrt/IdeaProjects/testplugin/t.c

```
typedef unsigned char __uint8_t;
typedef short __int16_t;
typedef unsigned short __uint16_t;
typedef int __int32_t;

struct _opaque_pthread_cond_t { lon
struct _opaque_pthread_condattr_t {

int fscanf(FILE * , const char * ,
int fseek(FILE *, long, int);

void *stringToNode(char *str)
{
 char *
 void *
 save_s
 (sizeof(__int32_t) * 8);
}
```

Deepest lookahead k=5
Ambiguous upon alts {5, 6}

Parse tree | Profiler

Invocations	Time	Total k	Max k	Ambiguities	DFA cache miss
23	44	68	27	4	44
6	6	20	7	2	11
12	7	20	9	1	12
0	0	0	0	0	0
0	0	0	0	0	0
11	0	13	2	0	5
0	0	0	0	0	0
0	0	0	0	0	0
11	3	15	4	0	7
0	0	0	0	0	0
0	0	0	0	0	0
11	1	11	1	0	6
3	0	3	1	0	2
10	1	11	2	0	5
1	0	1	1	0	1
10	0	10	1	0	5
0	0	0	0	0	0
9	0	9	1	0	4
0	0	0	0	0	0

Input size:	495 char, 18 lines
Number of tokens:	115
Parse time (ms):	337
Prediction time (ms):	312 = 92.58%
Lookahead burden:	771/115 = 6.70
DFA cache miss rate:	449/771 = 58.24%

☐ Show expert columns

Ambiguity
Context-sensitivity
Predicate evaluation
Deepest lookahead

AST vs Parse-Tree