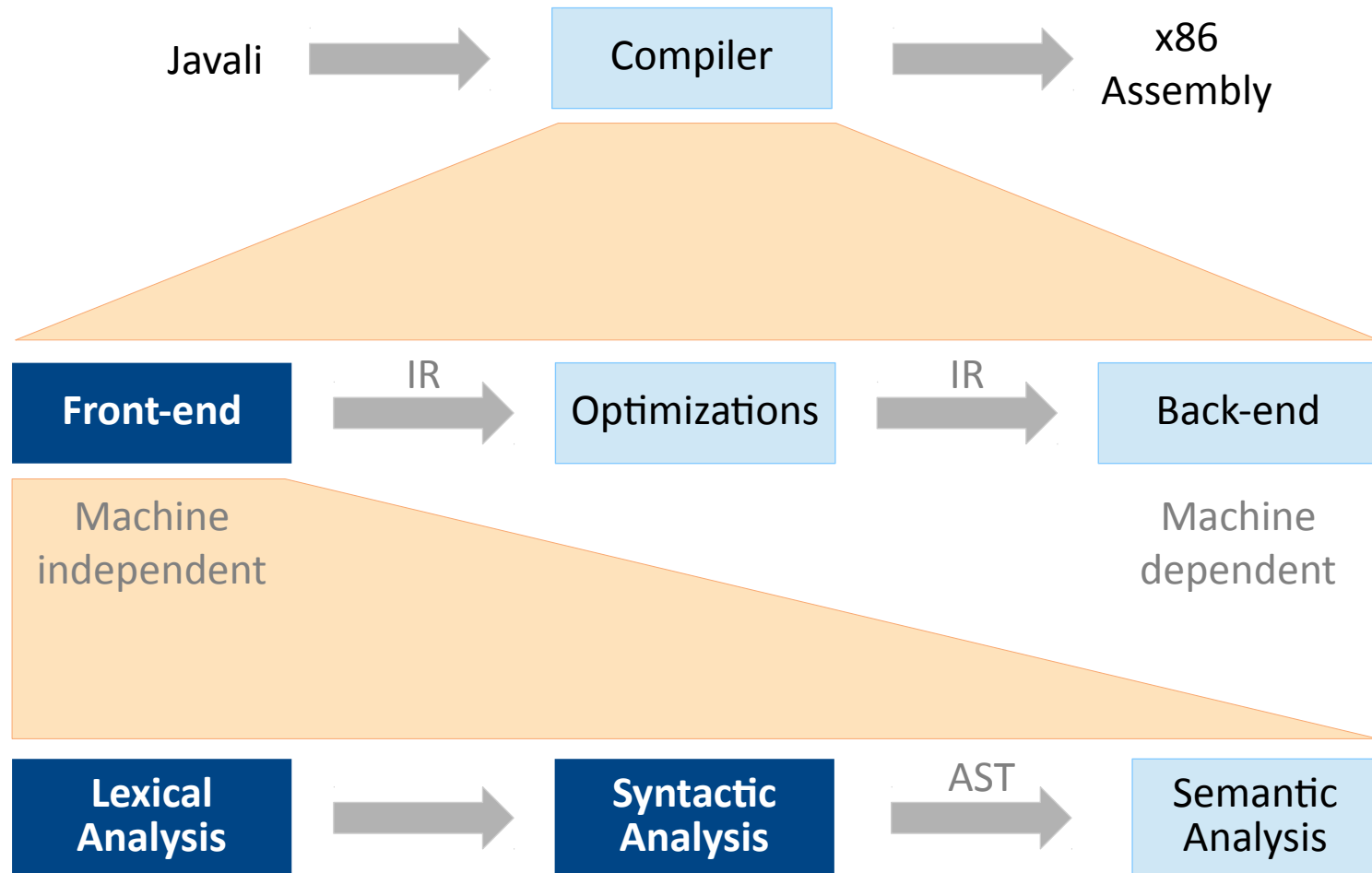


Homework 2: Parser and Lexer

Remi Meier

Compiler Design – 15.03.2018

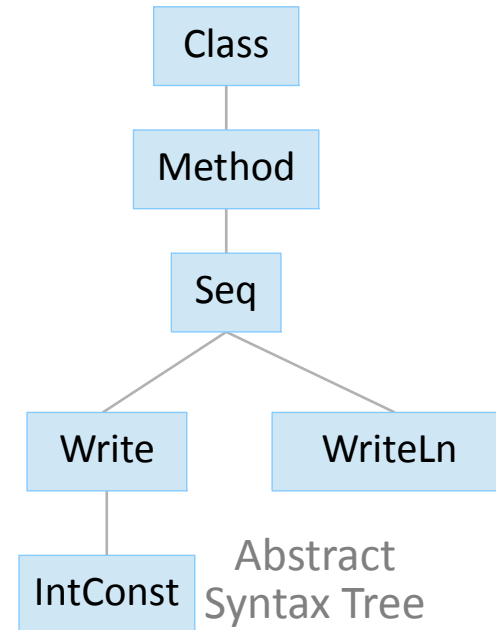
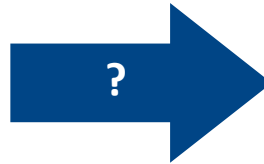
Compiler Phases



Homework 2

```
class Main {  
    void main() {  
        write(222);  
        writeln();  
    }  
}
```

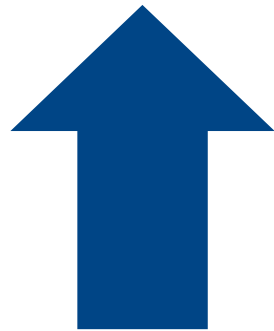
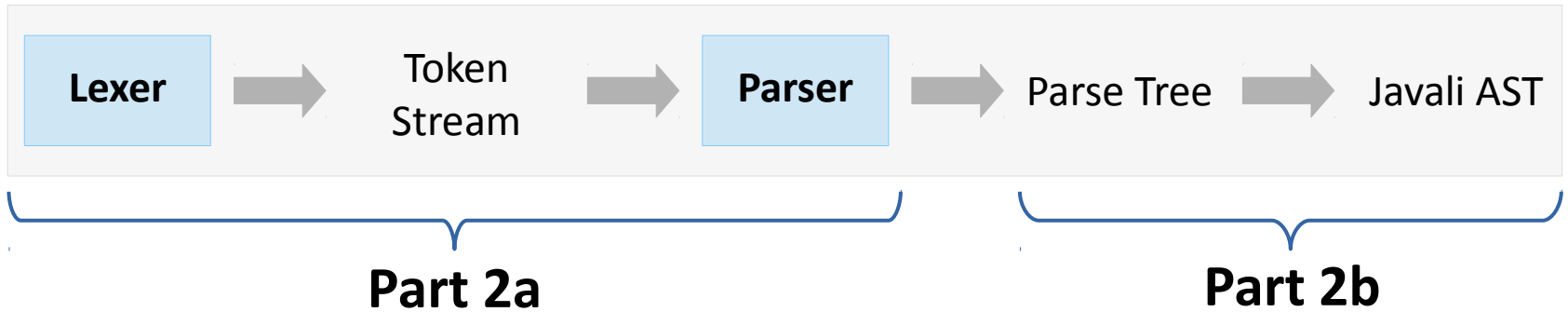
Text



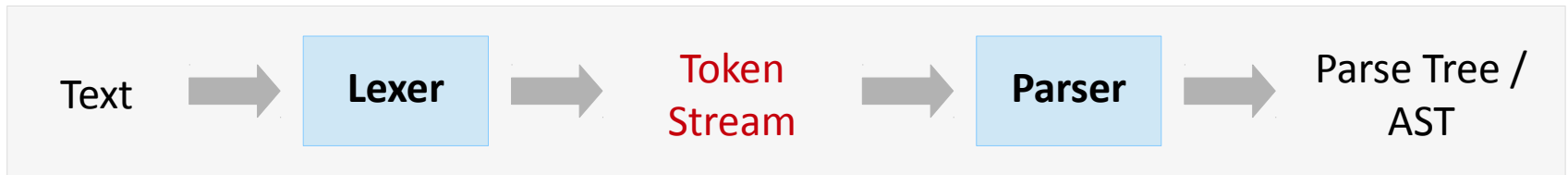
How do we...

- check if a program follows the syntax of Javali?
- extract meaning / structure?

Homework 2



Lexical Analysis



Lexer

- Read input character by character
- Recognize character groups → tokens

Token

- Sequence of characters with a **collective meaning** → grammar terminals
- E.g. constants, identifiers, keywords, ...

Lexical Analysis

```
class Main {  
    void main() {  
        write(222);  
        writeln();  
    }  
}
```

```
ID    : [a-zA-Z]+ ;  
NUM   : [0-9]+ ;  
MISC  : [{()}]; ;  
WS    : ('\\n'|' ') → skip ;
```

Token stream:

ID: *class*

ID: *Main*

MISC: {

ID: *void*

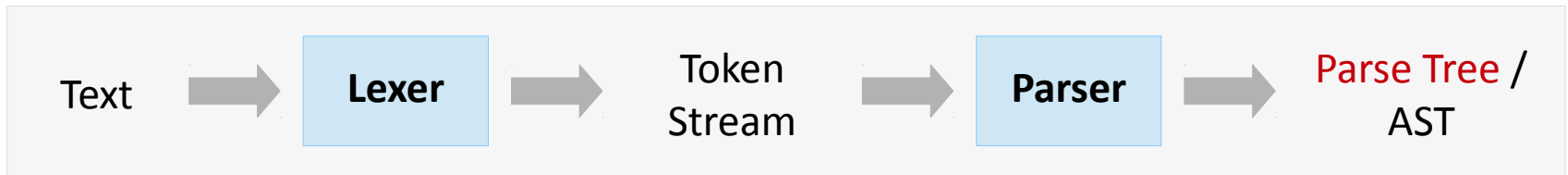
ID: *main*

MISC: (

MISC:)

...

Syntactic Analysis



Parser

- **Check** if token stream follows the grammar
- Group tokens hierarchically (**extract structure**)
→ Parse Tree / Abstract Syntax Tree



TOP-DOWN PARSER

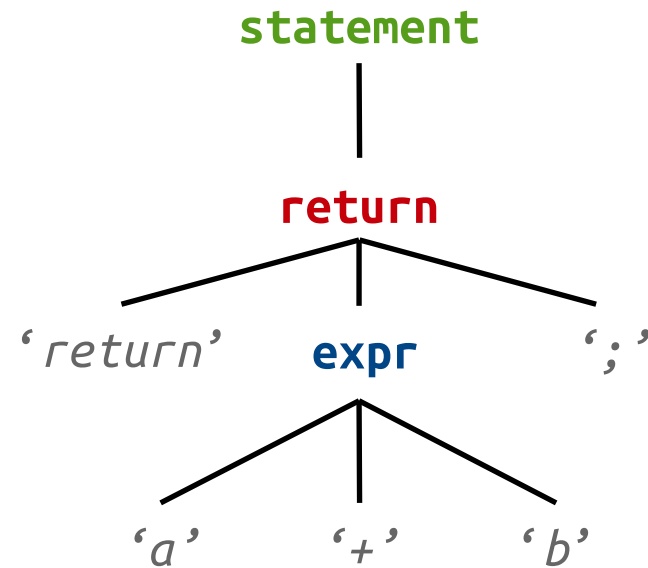
Top-down Parsers

Grammar in Extended Backus-Naur Form (EBNF):

```
statement:  
    return  
    | assign  
return:  
    'return' expr ';' ;  
assign:  
    ID '=' expr ';' ;  
expr: ID '+' ID
```

return a + b ;

return *a* *+* *b* *;*



Implementation

`return` `a` `+` `b` `;`

Grammar in Extended Backus-Naur Form (EBNF):

statement:

`return`
`|` `assign`

return:

`'return'` `expr` `';`

assign:

`ID` `'='` `expr` `';`

expr: `ID` `'+'` `ID`

```
void statement() {  
    return();  
    or assign();  
}  
void return() {  
    match('return');  
    expr();  
    match(';');  
}  
void expr() {
```

How to deal with alternatives?

Lookahead

`return` `a` `+` `b` `;`

Grammar in Extended Backus-Naur Form (EBNF):

```
statement:  
    return  
    | assign  
return:  
    'return' expr ';  
assign:  
    ID '=' expr ';  
expr: ID '+' ID
```

```
void statement() {  
    if (next() is 'return') {  
        return();  
    } else if (next() is ID) {  
        assign();  
    }  
}
```

LL(1)



ANTLR

<http://www.antlr4.org/>

(or HW2 fragment)

ANTLR

Token
specifications
+
Grammar



MyLexer.java
MyParser.java

Top-down parser generator

- ALL(*) adaptive, arbitrary lookahead
- handles any non-left-recursive context-free grammar

ANTLR – Grammar Description

Start rule matching end-of-file

Lower-case initial: Parser

Upper-case initial: Lexer
→ Tokens

```
/* This is an example */  
grammar Example;  
  
/* Parser rules = Non-terminals */  
program :  
    statement* EOF ;  
  
statement :  
    assignment ';' |  
    expression ';' ;  
  
/* Lexer rules = Terminals */  
Identifier : Letter (Letter | Digit)* ;  
Letter : '\u0024' | '\u0041'..' '\u005a' ;
```

ANTLR – Operators

Extended Backus-Naur Form (**EBNF**)

```
program :  
    statement* EOF;  
  
statement :  
    assignment ';' |  
    expression ';' ;  
  
method :  
    type name  
        '(' params? ')'  
    ;
```

EBNF operators	
x y z	(ordered) alternative
x?	at most once (optional)
x*	0 .. n times
x+	1 .. n times
[charset]	one of the chars, e.g.: [a-zA-Z]
'x'..'y'	characters in range

} lexer-only

Demo 1

ANTLR – Troubleshooting

ANTLR does not warn about **ambiguous** rules

- resolves ambiguity at runtime
→ requires lots of testing

ANTLR does not handle indirect **left-recursion**

- direct left-recursion supported

ANTLR – Lexer Ambiguity

What if some input is matched by multiple lexer rules?

```
parserRule : 'foo' parserRule ;  
  
fragment  
Letter : [a-z] ;  
Identifier : Letter+ ;
```

document order
↓

creates implicit lexer rule
T123 : 'foo'

fragment enforces that the rule never produces a token, but can be used in other lexer rules

can never match *foo*, but e.g., *foot*

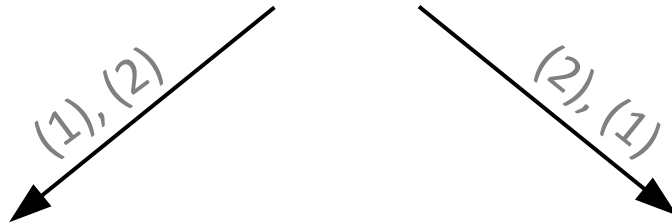
Lexer decides based on:

1. rule with the longest match first
2. literal tokens before all regular Lexer rules
3. document order
4. **fragment** rules never match on their own

ANTLR – Parser Ambiguity

```
stmt: 'if' expr 'then' stmt 'else' stmt (1)
     | 'if' expr 'then' stmt (2)
     | ID '=' expr ;
```

if *a* then if *c* then *d* else *e*



if *a* then if *c* then *d* else *e*

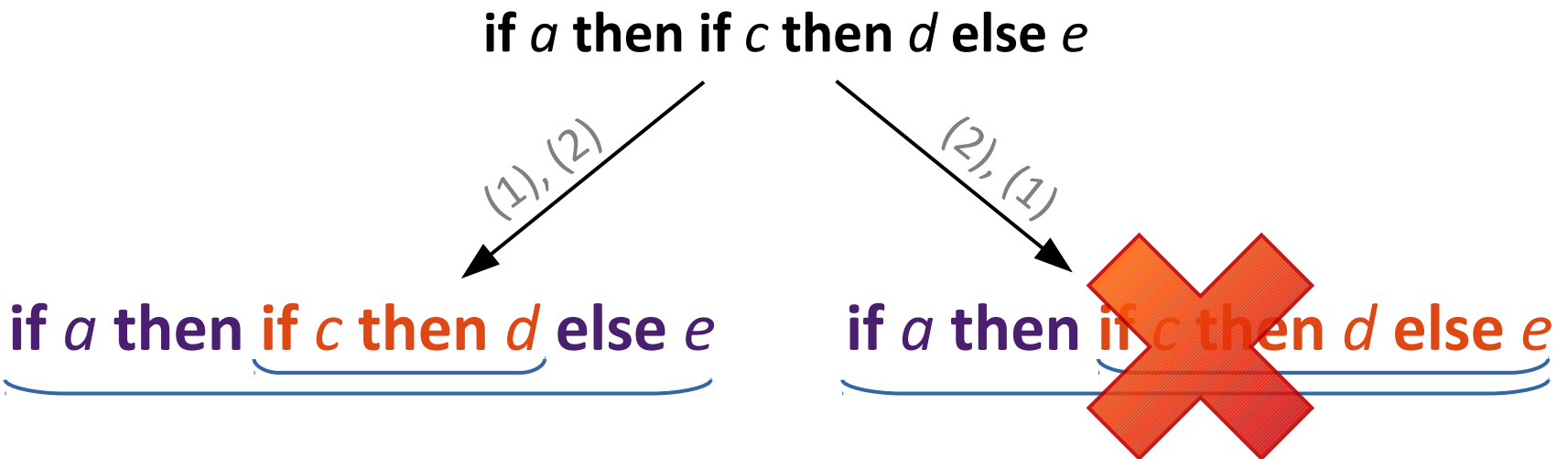
if *a* then if *c* then *d* else *e*

Ambiguous since there exist more than one parse trees for the same input.

ANTLR – Parser Ambiguity

```
stmt: 'if' expr 'then' stmt 'else' stmt (1)
     | 'if' expr 'then' stmt (2)
     | ID '=' expr ;
```

At decision points, if more than one alternative matches a given input, follow **document order**.



ANTLR – Parser Ambiguity

```
stmt: 'if' expr 'then' stmt 'else' stmt    (1)
     | 'if' expr 'then' stmt              (2)
     | ID '=' expr ;
```

At decision points, if more than one alternative matches a given input, follow **document order**.



Solution

```
stmt: 'if' expr 'then' stmt
     | 'if' expr 'then' stmt 'else' stmt
     | ID '=' expr ;
```

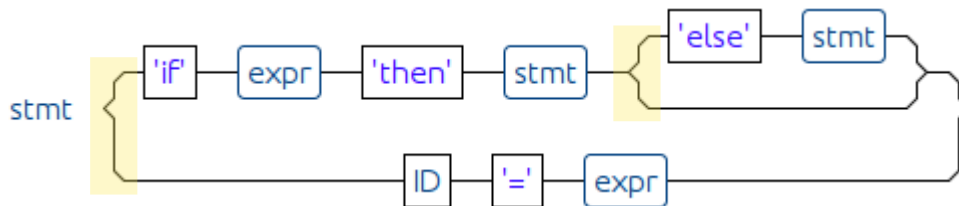
ANTLR – Parser Ambiguity

At **decision points**, if more than one alternative matches a given input, follow **document order**.

Alternative solution:

```
stmt: 'if' expr 'then' stmt ('else' stmt)?  
    | ID '=' expr ;
```

(...)? → (...|)



Sub-rules introduce additional decision points.

ANTLR – Left-recursion

Without: “a, b, c”

```
list : LETTER (',' LETTER)*;
```



Direct:

```
list : list ',' LETTER  
      | LETTER ;
```



Indirect:

```
list : LETTER  
      | longlist ;  
longlist : list ',' LETTER;
```



```
void list() {  
    if (???) {  
        list();  
        match(',');  
        match(LETTER);  
    } else {  
        match(LETTER);  
    }  
}
```

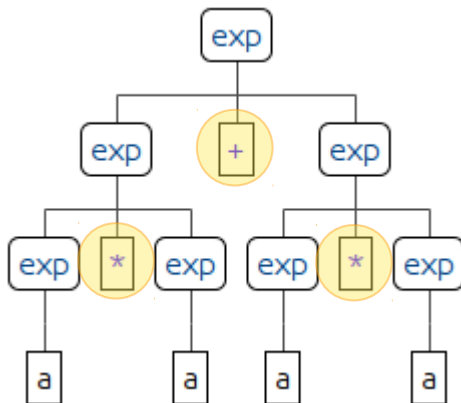
ANTLR – Direct Left-recursion

```
exp : exp '*' exp
    | exp '+' exp
    | ID ;
```

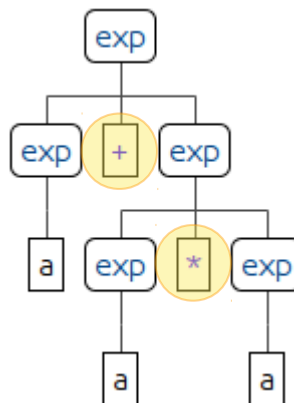


A grammar that implicitly assigns **priorities** to alternatives in document order

$a * a + a * a$



$a + a * a$



Demo 2

Homework

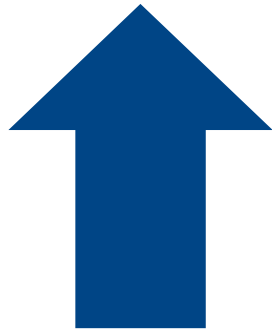


Part 2a

Parser grammar: Javali.g4

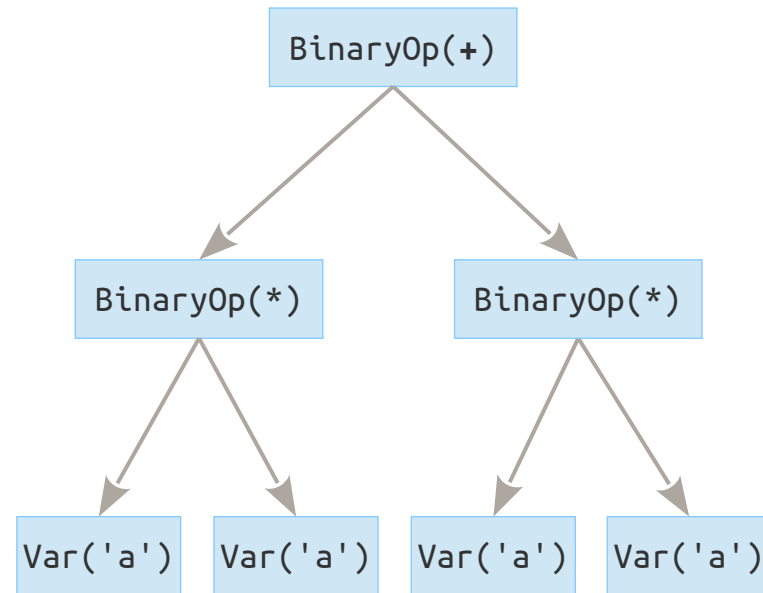
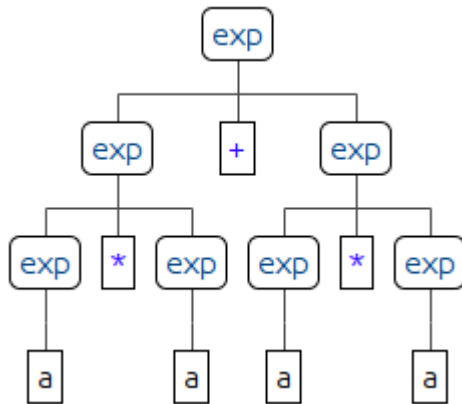
Part 2b

JavaliAstVisitor.java



Javali AST Generation

"a * a + a * a"



→ **JavaliAstVisitor.java**

Generated Files



Javali**Lexer/Parser**.java

- the real thing

Javali(**Base**)**Visitor**.java

- base class for parse-tree visitor

Javali(**Lexer**).tokens

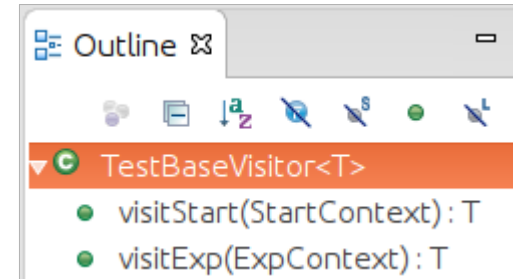
- token → number mapping for debugging

Generated Visitor

```
start : exp EOF
      ;

exp   : exp '*' exp
      | exp '+' exp
      | ID
      ;
```

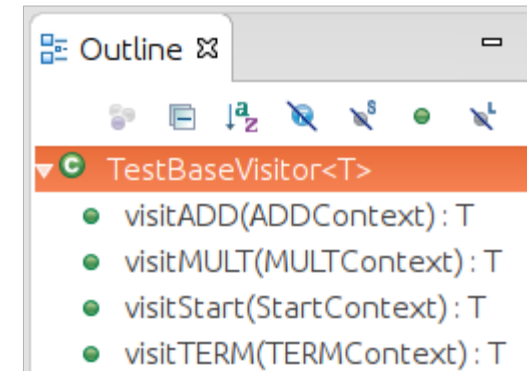
one method
per rule



```
start : exp EOF
      ;

exp   : exp '*' exp # MULT
      | exp '+' exp # ADD
      | ID          # TERM
      ;
```

one method
per label / rule

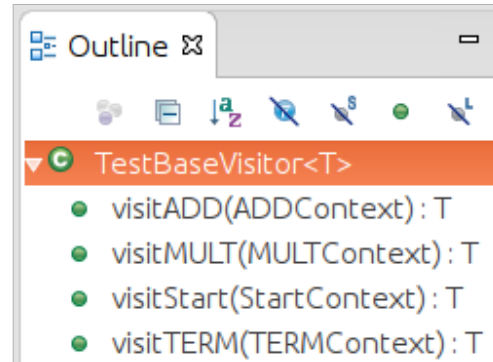


Constructing the Javali AST

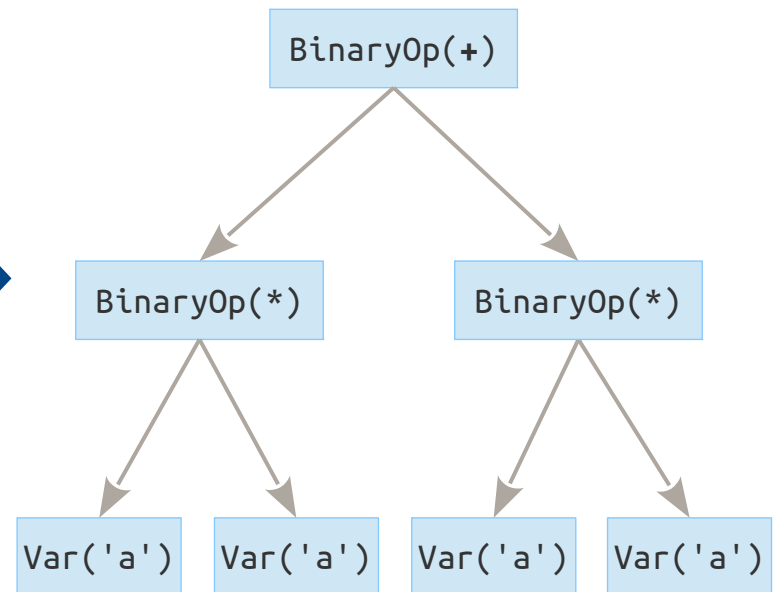
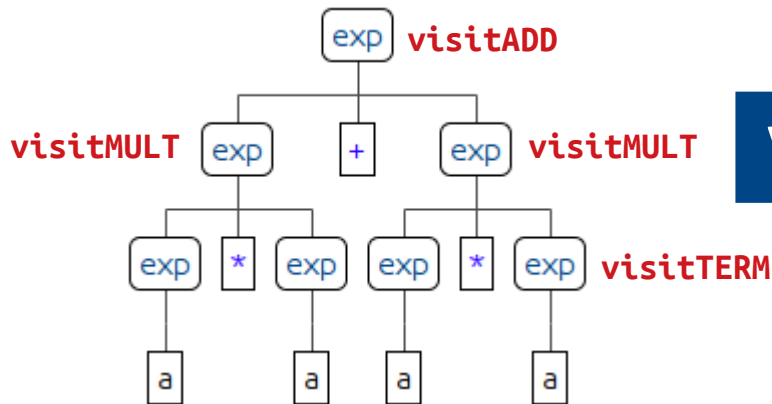
```

start : exp EOF
      ;

exp   : exp '*' exp # MULT
      | exp '+' exp # ADD
      | ID           # TERM
      ;
    
```



“a * a + a * a”



Demo 3

Final Notes

- Look on our website for more material.
- Due date is **March, 29th** at **10 a.m.**



Please don't do that...