Building Interpreters in ANTLR

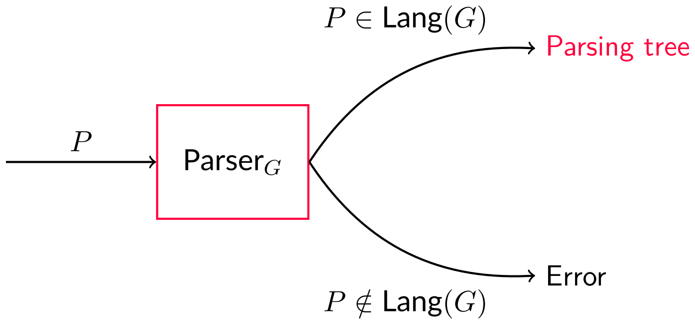
Programming Languages Lab

Samuele Buro
Programming Languages (AY 2020-21)
University of Verona
November 25, 2020

Parser Generation

$$\begin{tikzcd} {} \arrow[r, "G"] & \boxed{\text{ANTLR}} \arrow[r, "\text{Parser}_G"] & {} \end{tikzcd}$$

Syntactic Analysis



Details on ANTLR Syntactic Analysis

Details on ANTLR Syntactic Analysis

1. Tokenizing (Lexical Analysis): Group characters into words (tokens)

Details on ANTLR Syntactic Analysis

1. Tokenizing (Lexical Analysis): Group characters into words (tokens)
2. Parsing: Recognize the program structure and produce the parsing tree

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

```
public     PUBLIC
```

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS

Example

```java
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

`public`	`PUBLIC`
`_`	`WS`
`static`	`STATIC`

Example

```java
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS
int	INT

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
␣	WS
static	STATIC
␣	WS
int	INT
␣	WS

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
—	WS
static	STATIC
—	WS
int	INT
—	WS
factorial	ID

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS
int	INT
_	WS
factorial	ID
(	LPAR

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS
int	INT
_	WS
factorial	ID
(	LPAR
int	INT

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS
int	INT
_	WS
factorial	ID
(	LPAR
int	INT
_	WS

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS
int	INT
_	WS
factorial	ID
(	LPAR
int	INT
_	WS
n	ID

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS
int	INT
_	WS
factorial	ID
(	LPAR
int	INT
_	WS
n	ID
)	LPAR

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS
int	INT
_	WS
factorial	ID
(	LPAR
int	INT
_	WS
n	ID
)	LPAR
_	WS

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
␣	WS
static	STATIC
␣	WS
int	INT
␣	WS
factorial	ID
(	LPAR
int	INT
␣	WS
n	ID
)	LPAR
␣	WS
{	LBRACE

Example

```
public static int factorial(int n) {
  int r = 1;

  for (int i = n; i > 1; --i)
    r *= i;

  return r;
}
```

public	PUBLIC
_	WS
static	STATIC
_	WS
int	INT
_	WS
factorial	ID
(	LPAR
int	INT
_	WS
n	ID
)	LPAR
_	WS
{	LBRACE
…	…

Ambiguities in Lexical Analysis (?)

Ambiguities in Lexical Analysis (?)

1. Come first, match first!

```
FOR : 'for';
ID  : [a-z]+;
```

Ambiguities in Lexical Analysis (?)

1. Come first, match first!

   ```
   FOR : 'for';
   ID  : [a-z]+;
   ```

2. Longest-match rule

   ```
   A  : 'a';
   B  : 'b';
   C  : 'ab';
   ```

Ambiguities in Syntactical Analysis

A grammar is ambiguous if there exists a string which can have more than one parsing tree

```
exp : NAT | exp PLUS exp | exp TIMES exp ;

PLUS  : '+' ;
TIMES : '*' ;
NAT   : '0' | [1-9][0-9]* ;
```

Ambiguities in Syntactical Analysis

A grammar is ambiguous if there exists a string which can have more than one parsing tree

```
exp : NAT | exp PLUS exp | exp TIMES exp ;

PLUS  : '+' ;
TIMES : '*' ;
NAT   : '0' | [1-9][0-9]* ;
```

ANTLR resolves the ambiguity by choosing the first alternative involved in the decision!

The Expression Interpreter INTEXP

$$\texttt{(5 + (2 * 3))} \xrightarrow{[\![\mathsf{INTEXP}]\!]} 11$$

$$\texttt{3} \xrightarrow{[\![\mathsf{INTEXP}]\!]} 3$$

$$\texttt{(3 + 3)} \xrightarrow{[\![\mathsf{INTEXP}]\!]} 6$$

INTEXP Formal Grammar

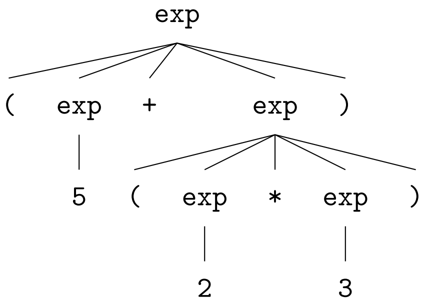
$$\mathit{exp} \rightarrow \mathit{n} \mid (\mathit{exp} + \mathit{exp}) \mid (\mathit{exp} * \mathit{exp}) \qquad \text{where } n \in \mathbb{N}$$

ANTLR Grammar

```
grammar IntExp;

exp   : NAT                           # nat
      | LPAR exp PLUS exp RPAR        # plus
      | LPAR exp MUL exp RPAR         # mul
      ;

LPAR : '(';
RPAR : ')';
PLUS : '+';
MUL  : '*';
NAT  : '0' | [1-9][0-9]*;

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

A Valid INTEXP Expression

`(5 + (2 * 3))`



ANTLR Representation

- Each terminal node is represented by an object `TerminalNode` that carries its string value

ANTLR Representation

- Each terminal node is represented by an object `TerminalNode` that carries its string value
- Each internal node is a `*Context`, which is recursively visitable

ANTLR Representation

- Each terminal node is represented by an object `TerminalNode` that carries its string value
- Each internal node is a `*Context`, which is recursively visitable
    - ANTLR builds a `*Context` for each grammar rule, i.e., for each non-terminal in the grammar; or

ANTLR Representation

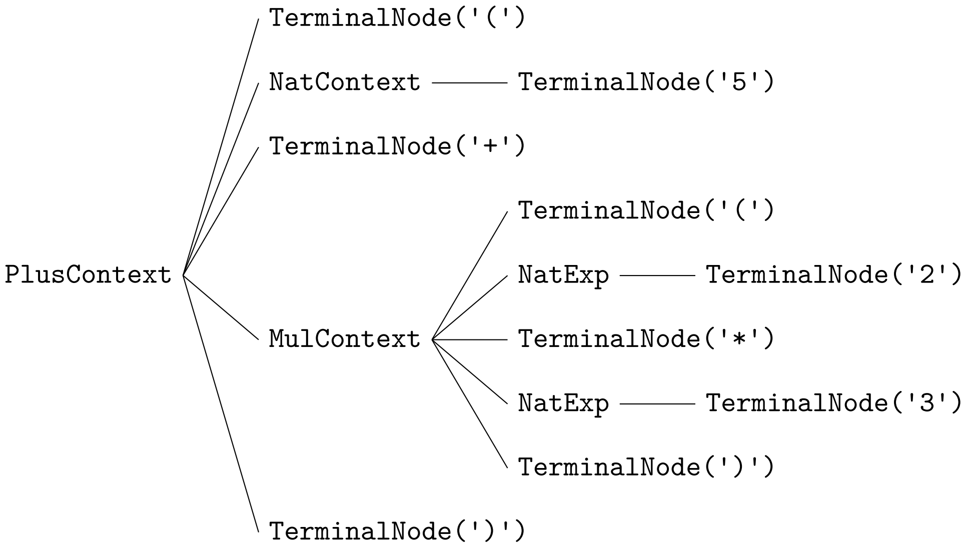
- Each terminal node is represented by an object `TerminalNode` that carries its string value
- Each internal node is a `*Context`, which is recursively visitable
    - ANTLR builds a `*Context` for each grammar rule, i.e., for each non-terminal in the grammar; or
    - ANTLR builds a `*Context` for each label of every grammar rule (without creating the context for that rule)

ANTLR Representation

- Each terminal node is represented by an object `TerminalNode` that carries its string value
- Each internal node is a `*Context`, which is recursively visitable
    - ANTLR builds a `*Context` for each grammar rule, i.e., for each non-terminal in the grammar; or
    - ANTLR builds a `*Context` for each label of every grammar rule (without creating the context for that rule)

Think of a `*Context` as a subtree!

ANTLR Representation (Example)



Creating the parsing tree in Java

1. Obtain the `CharStream` of the source code
2. Create the `Lexer` and the `Parser` (linked by a `CommonTokenStream`)
3. Create the `ParseTree`

1 — Creating the parsing tree in Java

Obtain the CharStream of the source code:

```
ClassLoader classloader =
   Thread.currentThread().getContextClassLoader();
InputStream inputStream =
   classloader.getResourceAsStream(args[0]);
CharStream charStream = CharStreams.fromStream(inputStream);
```

args[0] is the name of the file in src folder that contains the source code.

2 — Creating the parsing tree in Java

Create the `Lexer` and the `Parser` (linked by a `CommonTokenStream`):

```
IntExpLexer lexer = new IntExpLexer(charStream);
CommonTokenStream tokens = new CommonTokenStream(lexer);
IntExpParser parser = new IntExpParser(tokens);
```

3 — Creating the parsing tree in Java

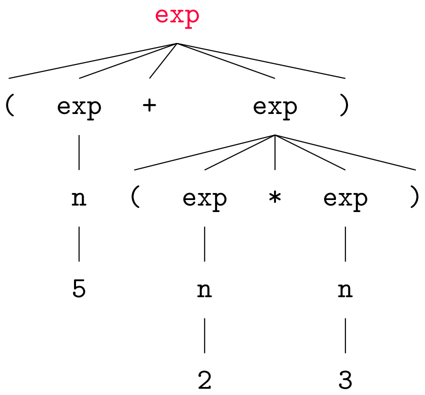
Create the `ParseTree`:

```
ParseTree tree = parser.exp();
```

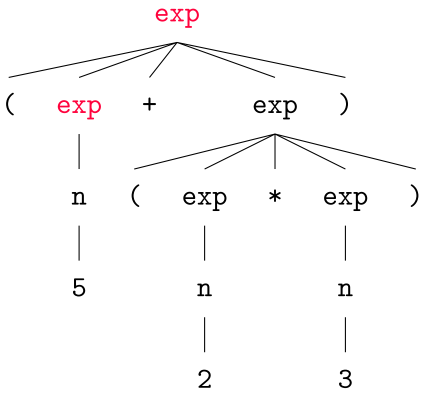
Execution

We can evaluate the program by recursively interpreting the parsing tree

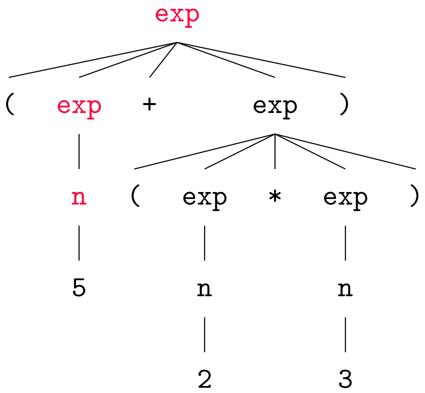
Example



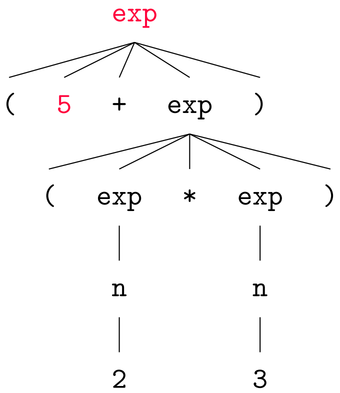
Example



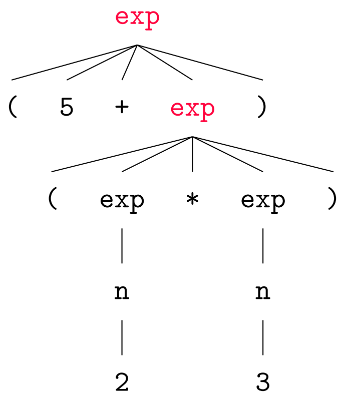
Example



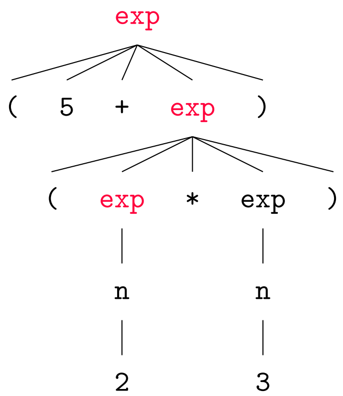
Example



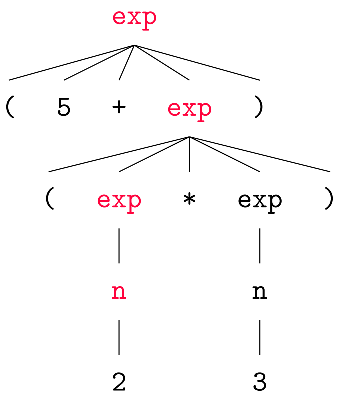
Example



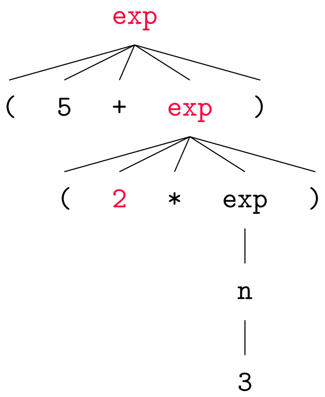
Example



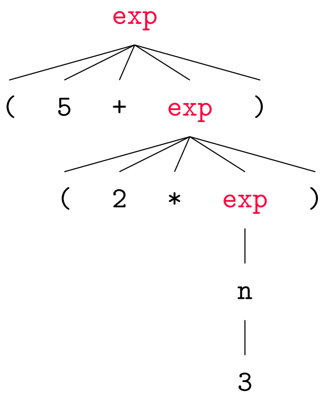
Example



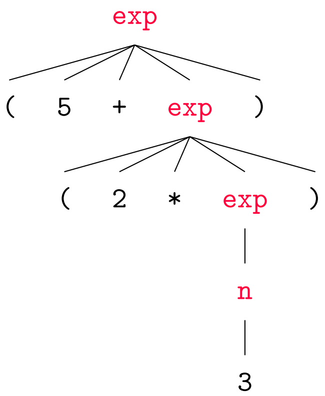
Example



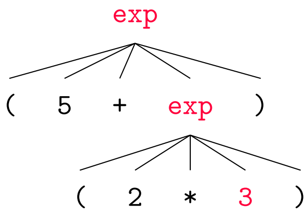
Example



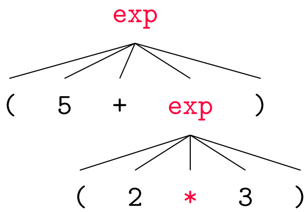
Example



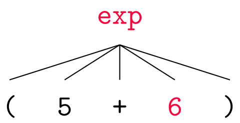
Example



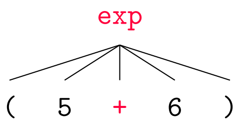
Example



Example



Example



Example

```
11
```

Visitor Pattern

For each type `*Context` ANTLR generates a `visit*`ing method in the `IntExpBaseVisitor` class. We can override these methods to perform a computation on the parsing tree.

Formal Semantics

$$\text{plus-left: } \frac{exp_1 \longrightarrow exp_1'}{(exp_1 + exp_2) \longrightarrow (exp_1' + exp_2)}$$

Formal Semantics

$$\text{plus-left: } \frac{exp_1 \longrightarrow exp_1'}{(exp_1 + exp_2) \longrightarrow (exp_1' + exp_2)}$$

$$\text{plus-right: } \frac{exp_2 \longrightarrow exp_2'}{(n_1 + exp_2) \longrightarrow (n_1 + exp_2')}$$

Formal Semantics

$$\text{plus-left: } \frac{\mathit{exp}_1 \longrightarrow \mathit{exp}'_1}{(\mathit{exp}_1 + \mathit{exp}_2) \longrightarrow (\mathit{exp}'_1 + \mathit{exp}_2)}$$

$$\text{plus-right: } \frac{\mathit{exp}_2 \longrightarrow \mathit{exp}'_2}{(n_1 + \mathit{exp}_2) \longrightarrow (n_1 + \mathit{exp}'_2)}$$

$$\text{plus: } \frac{-}{(n_1 + n_2) \longrightarrow n_3} n_3 = n_1 + n_2$$

Formal Semantics

$$\text{plus-left: } \frac{exp_1 \longrightarrow exp_1'}{(exp_1 + exp_2) \longrightarrow (exp_1' + exp_2)}$$

$$\text{plus-right: } \frac{exp_2 \longrightarrow exp_2'}{(n_1 + exp_2) \longrightarrow (n_1 + exp_2')}$$

$$\text{plus: } \frac{-}{(n_1 + n_2) \longrightarrow n_3} n_3 = n_1 + n_2$$

Rules for multiplication are similar!

Visitor Pattern (Example)

```
visit(tree)
```

Visitor Pattern (Example)

```
visit(tree)
  ⟶ visitPlus(PlusContext ctxplus)
```

Visitor Pattern (Example)

```
visit(tree)
  ⟶ visitPlus(PlusContext ctxplus)
  ⟶ visit(ctxplus.exp(0)) + visit(ctxplus.exp(1))
```

Visitor Pattern (Example)

```
visit(tree)
   ⟶ visitPlus(PlusContext ctxplus)
   ⟶ visit(ctxplus.exp(0)) + visit(ctxplus.exp(1))
   ⟶ visitNat(NatContext ctxnat5) + visitMul(MulContext ctxmul)
```

Visitor Pattern (Example)

```
visit(tree)
  ⟶ visitPlus(PlusContext ctxplus)
  ⟶ visit(ctxplus.exp(0)) + visit(ctxplus.exp(1))
  ⟶ visitNat(NatContext ctxnat5) + visitMul(MulContext ctxmul)
  ⟶ 5 + (visit(ctx.exp(2)) * visit(ctx.exp(3)))
```

Visitor Pattern (Example)

```
visit(tree)
  ⟶ visitPlus(PlusContext ctxplus)
  ⟶ visit(ctxplus.exp(0)) + visit(ctxplus.exp(1))
  ⟶ visitNat(NatContext ctxnat5) + visitMul(MulContext ctxmul)
  ⟶ 5 + (visit(ctx.exp(2)) * visit(ctx.exp(3)))
  ⟶ 5 + (visitNat(NatContext ctxnat2) * visitNat(NatContext ctxnat3))
```

Visitor Pattern (Example)

```
visit(tree)
    ⟶ visitPlus(PlusContext ctxplus)
    ⟶ visit(ctxplus.exp(0)) + visit(ctxplus.exp(1))
    ⟶ visitNat(NatContext ctxnat5) + visitMul(MulContext ctxmul)
    ⟶ 5 + (visit(ctx.exp(2)) * visit(ctx.exp(3)))
    ⟶ 5 + (visitNat(NatContext ctxnat2) * visitNat(NatContext ctxnat3))
```

$\longrightarrow 5 + (2 * 3)$

Visitor Pattern (Example)

```
visit(tree)
    ⟶ visitPlus(PlusContext ctxplus)
    ⟶ visit(ctxplus.exp(0)) + visit(ctxplus.exp(1))
    ⟶ visitNat(NatContext ctxnat5) + visitMul(MulContext ctxmul)
    ⟶ 5 + (visit(ctx.exp(2)) * visit(ctx.exp(3)))
    ⟶ 5 + (visitNat(NatContext ctxnat2) * visitNat(NatContext ctxnat3))
```

$\longrightarrow 5 + (2 * 3)$

$\longrightarrow 11$