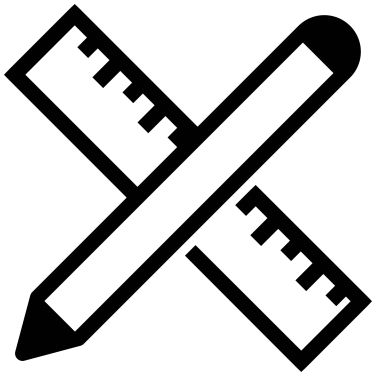
Advanced Grammar Design

Programming Languages Lab

Samuele Buro
Programming Languages (AY 2020-21)
University of Verona
December 9, 2020



Recurring Patterns in PL

Recurring Patterns in PL

Sequence of tokens or subphrases

Example: Assignment in Java

```
x = 3 + 5;
```

Recurring Patterns in PL

Sequence of tokens or subphrases

Example: Assignment in Java

```
x = 3 + 5;
```

Reasonable ANTLR rule:

```
rule : ID '=' exp ';' ;
```

Recurring Patterns in PL

Sequence of tokens or subphrases with terminator

Example: Sequence of command in Java

```
List<Integer> list = new ArrayList<Integer>;
list.add(0);
list.add(1);
list.add(2);
```

Recurring Patterns in PL

Sequence of tokens or subphrases with terminator

Example: Sequence of command in Java

```
List<Integer> list = new ArrayList<Integer>;
list.add(0);
list.add(1);
list.add(2);
```

Reasonable ANTLR rule:

```
rule : (statement ';')* ;
```

`x*` → Match `x` zero or more times

Recurring Patterns in PL

Sequence of tokens or subphrases with separator

Example: Function call arguments in Java

```
3, 5, new Object()
```

Recurring Patterns in PL

Sequence of tokens or subphrases with separator

Example: Function call arguments in Java

```
3, 5, new Object()
```

Reasonable ANTLR rule:

```
rule :  exp (',' exp)*   // non-empty
rule : (exp (',' exp)*)? // potentially empty
```

`x?` → Match `x` or skip it

Recurring Patterns in PL

Token dependency

Example: Array access in Java

`v[4]`

Recurring Patterns in PL

Token dependency

Example: Array access in Java

`v[4]`

Reasonable ANTLR rule:

```
rule :  ID '[' exp ']' ;
```

Recurring Patterns in PL

Nested phrase

Example: Nested class in Java

```
class A {
  class B {
    class C { }
  }
}
```

Recurring Patterns in PL

Nested phrase

Example: Nested class in Java

```
class A {
  class B {
    class C { }
  }
}
```

Reasonable ANTLR rule:

```
classDef : 'class' ID '{' (classDef | method | field) '}' ;
```

`(... | ... | ...)` → Subrule with multiple alternatives

Precedence

Real world programming languages handle precedence, e.g.,

```
3 + 4 * 5 + ++x <= 10 + 5 % 2
```

```
((3 + (4 * 5)) + (++x)) <= (10 + (5 % 2))
```

Precedence in ANTLR

The problem

Grammar: exp ::= exp * exp | exp + exp | n

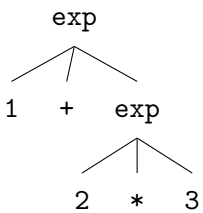
Expression: `1 + 2 * 3`

Precedence in ANTLR

The problem

Grammar: exp ::= exp * exp | exp + exp | n

Expression: 1 + 2 * 3

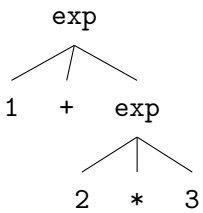


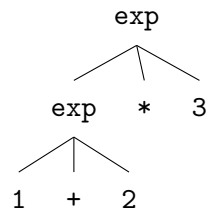
Precedence in ANTLR

The problem

Grammar: exp ::= exp * exp | exp + exp | n

Expression: 1 + 2 * 3



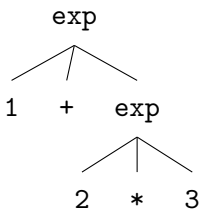


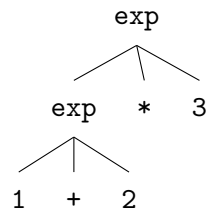
Precedence in ANTLR

The problem

Grammar: exp ::= exp * exp | exp + exp | n

Expression: 1 + 2 * 3

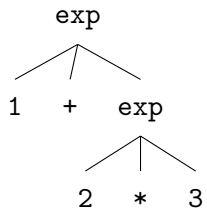




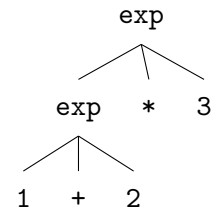
Precedence in ANTLR

ANTLR resolves ambiguities in favor of the alternative given first

```
exp : exp * exp
    | exp + exp
    | NAT
    ;
```




```
exp : exp + exp
    | exp * exp
    | NAT
    ;
```



Associativity

Real world programming languages handle associativity too, e.g.,

```
2^3^4 = 2^(3^4)
10 % 4 % 2 = (10 % 4) % 2
```

Associativity in ANTLR

The problem

Grammar: exp ::= exp ^ exp | n

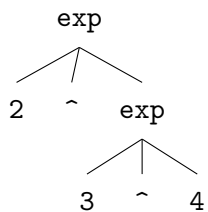
Expression: 2^3^4

Associativity in ANTLR

The problem

Grammar: exp ::= exp ^ exp | n

Expression: 2^3^4

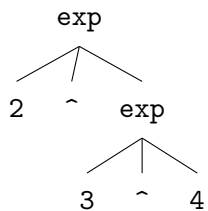


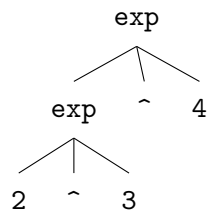
Associativity in ANTLR

The problem

Grammar: exp ::= exp ^ exp | n

Expression: 2^3^4



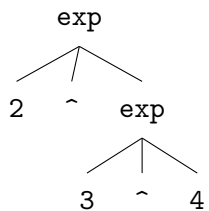


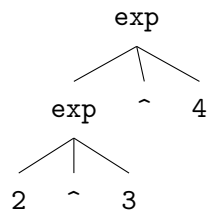
Associativity in ANTLR

The problem

Grammar: exp ::= exp ^ exp | n

Expression: 2^3^4



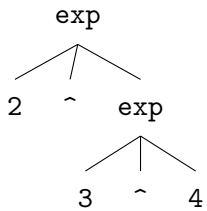


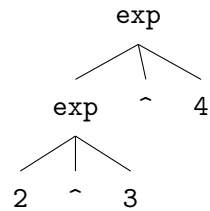
Associativity in ANTLR

By default, ANTLR associates operators left to right, but we can manually specify the associativity of an operator using option `assoc`:

```
exp : <assoc=right> exp ^ exp
    | NAT
    ;
```

```
exp : exp ^ exp
    | NAT
    ;
```





Common Lexical Structures

Common Lexical Structures

Identifier

```
ID : [a-zA-Z]+ ;
```

Match one or more uppercase or lowercase letters

Common Lexical Structures

Natural numbers

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

Match zero or any sequence of digits starting with 1, 2, ..., or 9

Common Lexical Structures

Integers

```
INT : NAT | '-' POS ;
fragment NAT : '0' | POS ;
fragment POS : [1-9][0-9]* ;
```

Match natural numbers or positive numbers preceded by the token –

Common Lexical Structures

Floats

```
FLOAT : INT | (INT | '-' '0') '.' DIGIT+ ;
fragment INT : NAT | '-' POS ;
fragment NAT : '0' | POS ;
fragment POS : POSDIGIT DIGIT* ;
fragment DIGIT : '0' | POSDIGIT ;
fragment POSDIGIT : [1-9] ;
```

Match floating-point numbers

The `fragment` keyword

`fragment` is a prefix for lexer rules

- rules prefixed with `fragment` can be called only from other lexer rules; they are not tokens in their own right
- makes the grammar more readable and easier to maintain

Common Lexical Structures

String literals

```
STRING : '"' STRCHR* '"' ;

fragment STRCHR : ~["\\] | ESC ;
fragment ESC : '\\' [btnfr"'\\] ;
```

Match all the strings delimited by "..."(~ performs the complement of the language specified by `[...]`)

Common Lexical Structures

Comments

```
COMMENT : '/*' .*? '*/' -> skip ;
LINE_COMMENT : '//' ~[\r\n]* -> skip ;
```

Match all the Java-style comments (`*?` is the non-greedy dot wildcard operator; it matches all the symbols until what follows in the rule, i.e., `*/`)

Common Lexical Structures

Whitespace

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

Labeling Rule Alternative for Precise Event Methods

Labeling Rule Alternative for Precise Event Methods

```
exp : exp op=(PLUS | MINUS) exp # plusMinus
    | ...
    ;

@Override
public Integer visitPlusMinus(Parser.PlusMinusContext ctx) {
    switch (ctx.op.getType()) {
      case Parser.PLUS :
          return visit(ctx.exp(0)) + visit(ctx.exp(1));
      case Parser.MINUS :
          return visit(ctx.exp(0)) - visit(ctx.exp(1));
    }

    return 0; // unreachable code statement
}
```