Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Programming Language Design

Syntactic / Syntax Analysis

Francisco Ortin

CORK INSTITUTE OF TECHNOLOGY

Department of Computer Science

Contents

- Objectives of the Syntax Analyzer
- Derivations and Parse Trees
- **Ambiguous Grammars**
- Parsing Strategies
- The ANTLR Parser Generator
- Abstract Syntax Trees

Mandatory Activity

- Recall the following activity (previous lecture):

Determine (by extension) the language defined by the following grammar:

(1)	*stmt*	$\rightarrow$	*if-stmt*
(2)		|	ID = *exp* ;
(3)	*exp*	$\rightarrow$	ID
(4)		|	INT_CONSTANT
(5)	*if-stmt*	$\rightarrow$	IF (*exp*) *stmt*
(6)		|	IF (*exp*) *stmt* ELSE *stmt*

Solution

L(G) = {
  ID = ID ;
  ID = INT_CONSTANT ;
  IF (ID) ID = INT_CONSTANT ;
  IF (INT_CONSTANT) ID = ID ; ELSE ID = INT_CONSTANT ;
  IF (ID) IF (ID) ID = INT_CONSTANT ; ELSE ID = ID ; // P(6)P(5)
  IF (ID) IF (ID) ID = INT_CONSTANT ; ELSE ID = ID ; // P(5)P(6)
  ...
}

G:

(1) *stmt* → *if-stmt*
(2) | ID = *exp* ;
(3) *exp* → ID
(4) | INT_CONSTANT
(5) *if-stmt* → IF (*exp*) *stmt*
(6) | IF (*exp*) *stmt* ELSE *stmt*

Mandatory Activity

- <u>Activity</u>: Given the following grammar

(1)	*stmt*	$\rightarrow$	*if-stmt*
(2)		|	ID = *exp* ;
(3)	*exp*	$\rightarrow$	ID
(4)		|	INT_CONSTANT
(5)	*if-stmt*	$\rightarrow$	IF (*exp*) *stmt*
(6)		|	IF (*exp*) *stmt* ELSE *stmt*

- Is the following program syntactically valid?
- If so, identify the parse tree

```
if (a)
      if (b) c=1;
else  c=2;
```

Ambiguous Grammars

- A grammar that generates two distinct parse trees for the same program is an **ambiguous grammar**
  - Two distinct derivations exist for the same program
- Ambiguous grammars represent a serious problem because the semantics of the distinct trees are (commonly) distinct too
  - Therefore, the generated program may be distinct
- We must not use ambiguous grammars to define a language, because the generated programs may be incorrect
- The decision problem of whether a grammar is ambiguous is **undecidable** (no algorithm exists)

Contents

- Objectives of the Syntax Analyzer
- Derivations and Parse Trees
- Ambiguous Grammars
- **Parsing Strategies**
- The ANTLR Parser Generator
- Abstract Syntax Trees

Parsing Strategies

- Depending on how the parse tree is built, parsers may be either **top-down** or **bottom-up** parsers
- **Top-down** parsers begin with the start symbol and apply the derivation of the left-most non-terminal symbol
- **Bottom-up** parsers begin with leaf nodes and build non-terminal nodes when all the child nodes of that derivation have been created

Program:

a = b + 7

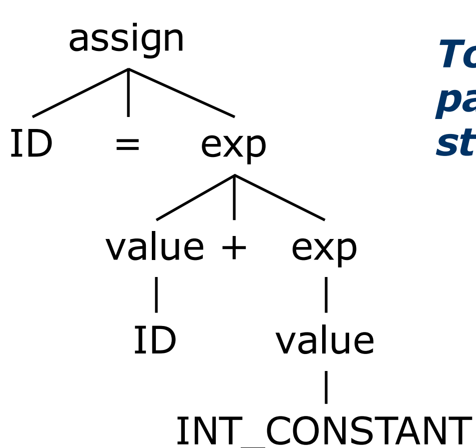
Grammar:

assign → ID = *exp*

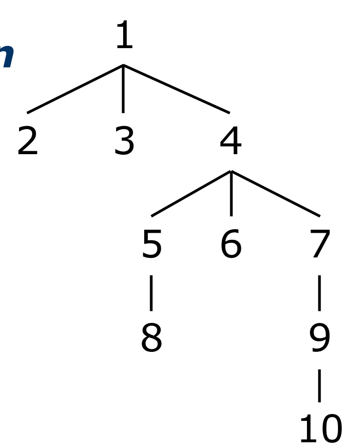
exp → *value* + *exp* | *value*

value → ID | INT_CONSTANT

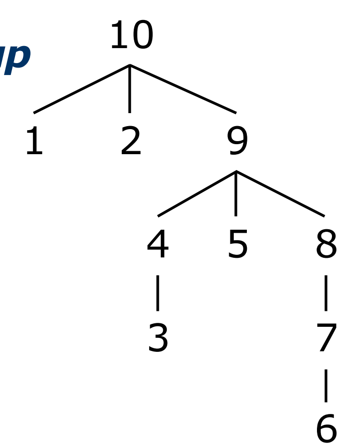
Parse tree:



Top-down parsing steps:



Bottom-up parsing steps:



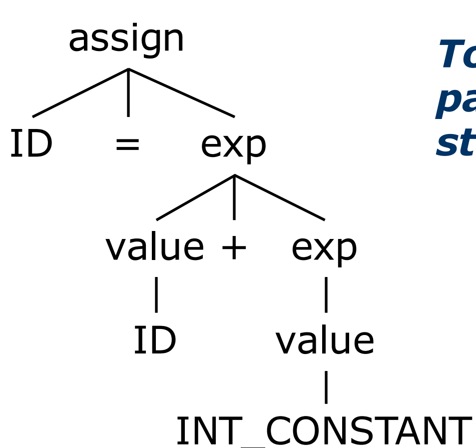
Top-Down Parsers

- Top-down parsers apply **left-most derivations**
  - The <u>left-most non-terminal</u> is replaced at each one-step derivation
- Left-most derivations are denoted by $\Rightarrow_L$

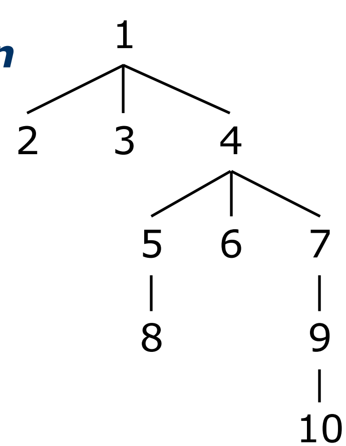
$$\begin{array}{lll}
\underline{\mathit{assign}} & \Rightarrow_L & \text{ID} = \underline{\mathit{exp}} \\
 & \Rightarrow_L & \text{ID} = \underline{\mathit{value}} + \mathit{exp} \\
 & \Rightarrow_L & \text{ID} = \text{ID} + \underline{\mathit{exp}} \\
 & \Rightarrow_L & \text{ID} = \text{ID} + \underline{\mathit{value}} \\
 & \Rightarrow_L & \text{ID} = \text{ID} + \text{INT_CONSTANT}
\end{array}$$

Grammar:
assign → ID = *exp*
exp → *value* + *exp* | *value*
value → ID | INT_CONSTANT

Parse tree:



Top-down parsing steps:



Program:
a = b + 7

Left-Recursion

- Top-down parsers must avoid **left-recursion**, because the parser may enter an infinite recursive call
- Instead of the following grammar

  $$listOfIds \rightarrow listOfIds \text{ , ID} \mid \text{ID}$$

- We write a right-recursive one

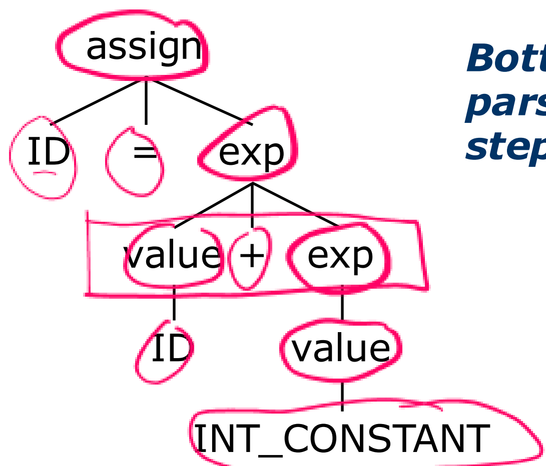
  $$listOfIds \rightarrow \text{ID , } listOfIds \mid \text{ID}$$

- However, bottom-up parsers are more efficient when left-recursion is used
- **Remember**,
  - Right recursion for top-down parsers
  - Left recursion for bottom-up parsers

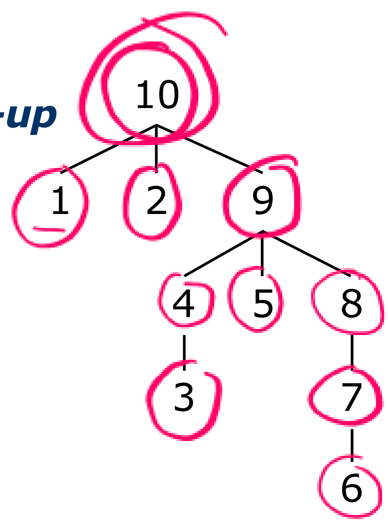
Bottom-Up Parsing

- **Bottom-up** parsers begin with leaf nodes and build non-terminal nodes when all the child nodes of that derivation have been created
- That is, <u>productions</u> are executed in the <u>opposite direction</u> (so, they are called **reductions**)
  - **Reduction**: when the right-hand side (child) nodes have been created, the left-hand side (parent) is then created

Parse tree:



Bottom-up parsing steps:



Grammar:

assign → ID = *exp*
exp → *value* + *exp*
| *value*
value → ID
| INT_CONSTANT

Program:

```
a = b + 7
```

赋值

LL(k) and LR(k) Parsers

- Top-Down parsers are also called **LL(k)**
  - The 1st ***L*** means that the input is read from **left** to right
  - The 2nd ***L*** means that the parser performs **left**-most derivations
  - ***k*** is the number of *loo**k**ahead* tokens analyzed to perform each one-step derivation
- Bottom-up parsers are also called **LR(k)**
  - The ***L*** means that the input is read from ***left*** to right
  - The ***R*** means that the parser performs ***right***-most derivations
  - ***k*** is the number of *loo**k**ahead* tokens
- LR(k) grammars are <u>more expressive</u> than LL(k)
- **Yacc** is <u>LR(1)</u> (actually, it is LALR(1), a kind of LR)
- **ANTLR** is <u>LL(*)</u> and allows <u>**direct** left recursion</u>

Contents

- Objectives of the Syntax Analyzer
- Derivations and Parse Trees
- Ambiguous Grammars
- Parsing Strategies
- **The ANTLR Parser Generator**
- Abstract Syntax Trees

Parser Generators

- A **parser generator** is a program that takes the specification of a language syntax as its input and produces a **parser** for that language
- There are LL and LR parser generators:
  - **LL**: ANTLR, JavaCC, Coco/R, LLgen, LISA
  - **LR**: Yacc / Bison, SableCC, CUP, LISA

Syntax Specification

Parser Generator

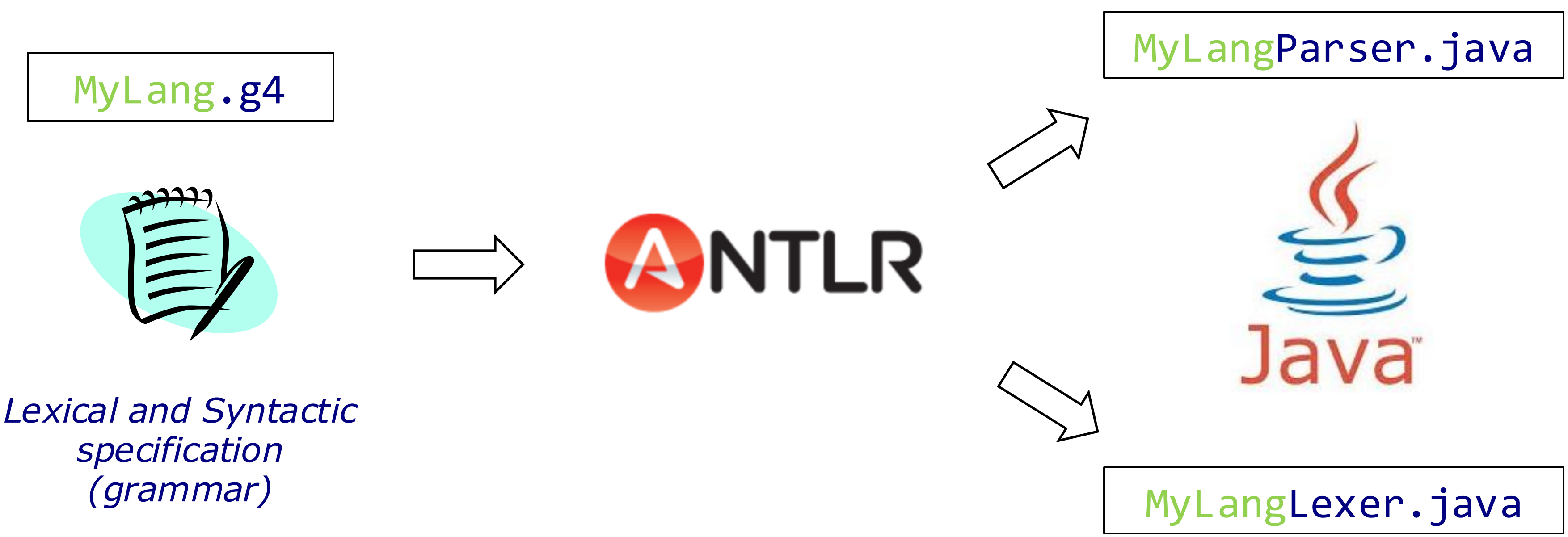
`Parser.java`

ANTLR

- **ANTLR** is a LL(*) parser (and lexer) generator
  - Top-down (LL) parsing
  - With finite but not fixed lookahead (*)
  - Allows dynamic parsing with semantic predicates (parsing depending on dynamic conditions)
  - ANTLR 4 supports **direct** left recursion
- It also provides **tree walkers** (visitors) and **event-based** language **processing** (listeners)
- Supports Java, C#, C++, JavaScript, Python2, Python3, Swift and Go
- Provides grammars for many real languages
- Used in many real systems: X (Twitter), Hadoop, Android, Lex Machina, Oracle, PayPal, NetBeans IDE, HQL Hibernate…

ANTLR

- **ANTLR** receives the **lexical** and **syntactic specification** of a language and generates the **lexer** and **parser** implementations



ANTLR Specification File

- Recall the specification file structure

General Structure

Grammar Name

Options

Syntax rules

Lexical rules

Example

```
grammar Cmm;

@header {
  import ast.*;
  import types.*;
}

// Syntax specification
program: …
       ;
…

// Lexical specification
INT_CONSTANT: …
            ;
…
```

Example

- Question: What is the language recognized by the following ANTLR specification?

```
grammar Example;

program: (type variables ';')*
       ;
type: 'int'
    | 'double'
    | 'char'
    ;
variables: ID (',' ID)*
         ;
// lexical specification...
```

Example

- Example use of the generated parser

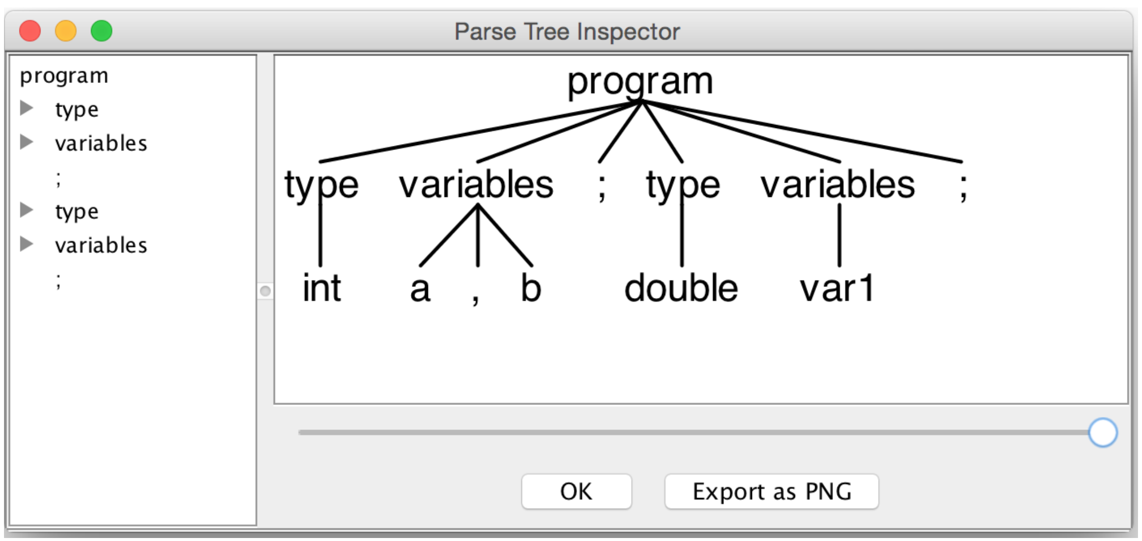
```
// Input char stream for text files
CharStream input = CharStreams.fromFileName("input.txt");
ExampleLexer lexer = new ExampleLexer(input);

// A pipe to connect the lexer and the parser
CommonTokenStream tokens = new CommonTokenStream(lexer);
ExampleParser parser = new ExampleParser(tokens);

// Parsing, returning the parse tree
ProgramContext tree = parser.program();
```

- We do not need to call the lexer (the parser does it for us)

Test Rig

- ANTLR provides a powerful tool called **Test Rig**
- It provides
  - A visual (`-gui`) and textual (`-tree`) representation of the **parse tree** (**not the AST**) generated
  - A description of all the **tokens** recognized (`-token`)
  - A **trace** of the parsing process (`-trace`)
- It can be called programmatically or as a Java application

```
C:> java org.antlr.v4.runtime.misc.TestRig parser.MyLang
                                          program –gui input.txt
```



- Very useful to test grammars (we do not need to implement a `main` method to test the recognizer)!
- The IntelliJ plugin includes Test Rig

Mandatory Activity 1

- Given the following ANTLR syntax specification

```
expression: expression '+' expression
          | expression '-' expression
          | expression '*' expression
          | expression '/' expression
          | ID
          | INT_CONSTANT
          ;
```

- Represent the parse tree for the input program
  `1 – 2 * 3`
- Does it follow the Java / C semantics?

Mandatory Activity 2

- Given the following ANTLR syntax specification

```
expression: expression '+' expression
          | expression '-' expression
          | expression '*' expression
          | expression '/' expression
          | ID
          | INT_CONSTANT
          ;
```

- Represent the parse tree for the input program

  `1 - 2 - 3`

- Does it follow the Java / C semantics?

Mandatory Activity 3

- As mentioned, ANTLR supports **direct left recursion**

1. The ambiguity of different operators

```
expression: expression '*' expression
          | expression '+' expression
```

   is solved with a **highest to lowest precedence** in the order of the productions

2. Since ANTLR is LL(*), the ambiguous left- and right-recursions (*exp* $\rightarrow$ *exp* - *exp*) are solved with **left**-to-right **associativity** (applicable when precedence is the same)

- Question: So, how should the previous grammar be specified in ANTLR?

Mandatory Activity 4

- With the grammar

```
expression: expression ('*'|'/') expression
          | expression ('-'|'+') expression
          | ID
          | INT_CONSTANT
          ;
```

- How do we add the = operator, so that the following program is recognized?

```
a = b = 0
```

- Does it follow the Java / C semantics?

Autonomous Activity 5

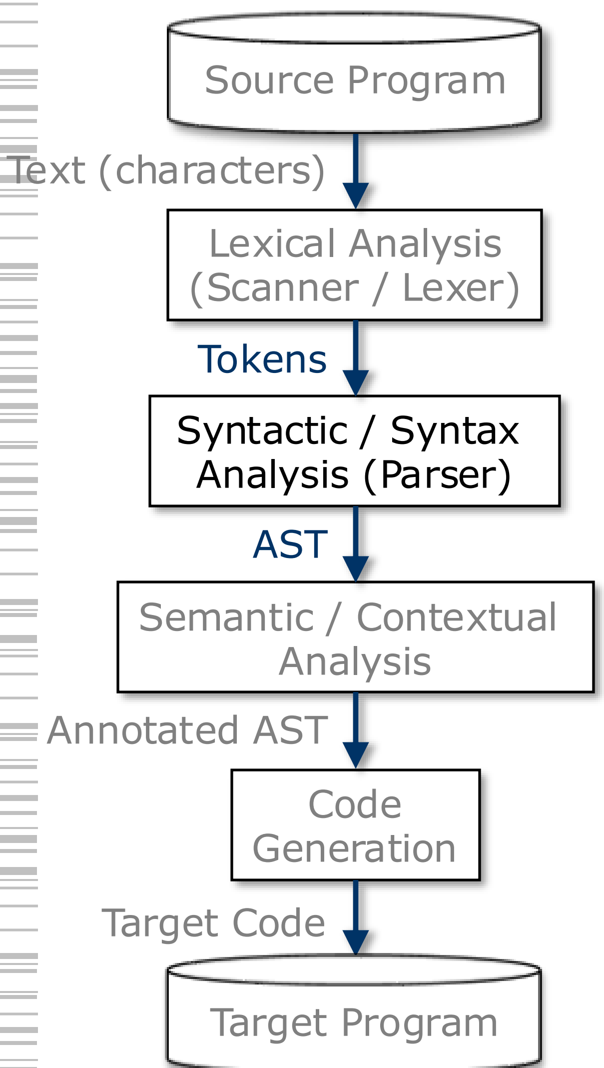
- Assuming that the productions for the `expression` symbol have already been defined…
- Define the `statement` productions to recognize the syntax of `while`, `if` and `assignment` statements, following the Java / C syntax
- Example program that must be recognized:

```
a = v[i*3];
while (a) a = 1;
while (a+b) {
  a = b;
}
if (c) c = 0;
if (d)
  if (e) { e = 1; }
  else e = 0;
```

Contents

- Objectives of the Syntax Analyzer
- Derivations and Parse Trees
- Ambiguous Grammars
- Parsing Strategies
- The ANTLR Parser Generator
- **Abstract Syntax Trees**

Recall, Parser Objectives



- The **syntactic analyzer** (parser) has two objectives:
  1. <u>Checking</u> whether the <u>tokens</u> represent a <u>valid sequence</u> (i.e., the program is syntactically correct), A.K.A. *parsing*

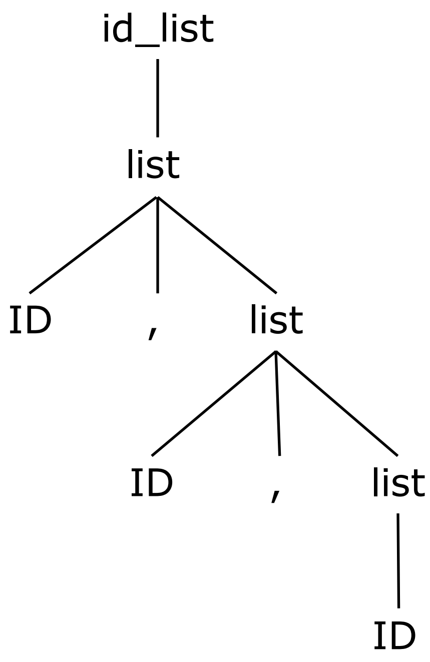
     (we achieve it with the ANTLR parser generator tool)
  2. **Build an AST representing the structure of the source program**

Parse Trees

- An LL parser creates the <u>parse tree</u> nodes **upon production** execution (left-most **derivations**)

id_list → ε
      | list
list → ID , list
      | ID

Parse tree for
`a, b, c`



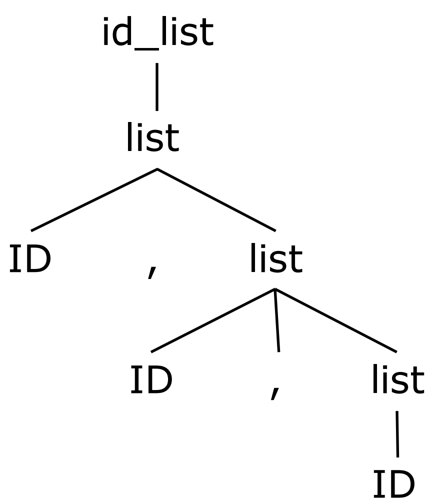
Parse Trees vs. ASTs

- Parse trees have **too many nodes** (even for the EBNF notation used in ANTLR)
- Many nodes are created to **avoid ambiguity** (delimiters, separators, parenthesis…)
- Equivalent trees with **fewer nodes** can be found

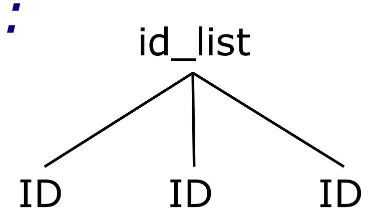
empty_list $\rightarrow$ ε | list
list $\rightarrow$ list , ID | ID

- These simplified parser trees are called **AST**s (Abstract Syntax Trees)

Parse Tree:



AST:



Abstract Syntax

- Since a **CFG grammar** enables the recognition of an input program whose structure is represented by a parse tree…
- Is there a **grammar** for the AST representing an input program?

- Yes, the **abstract** (**syntax**) **grammar**

- Thus, an **AST** can be seen as the data structure that represents an input program for a given **abstract grammar**

Abstract Syntax

- **Abstract grammars** are widely used in the design of programming languages
  - Once we have defined the (concrete) syntax, it is not necessary to work with the verbose concrete grammar
- Abstract grammars are commonly **ambiguous** grammars
  - Causing no trouble, because we already dealt with ambiguity with the original (concrete) grammar
- As concrete grammars, the productions of an abstract grammar specify a relationship between a **parent symbol** (left-hand side) and a sequence of **child symbols** (right-hand side) **of an AST**

Abstract Syntax (Scott Notation)

- There exist different notations to represent abstract grammars
- We will use the one defined by Michael L. Scott
- Grammar productions (P) are formalized as:

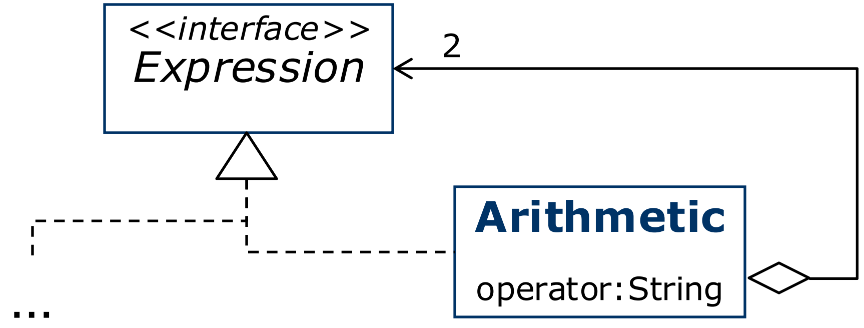
  A: $b \rightarrow \alpha$

  where

  - ***A*** is the concrete (i.e., non-abstract) parent node in the AST for that abstract production
    - I.e., the dynamic type of the parent node in the tree

Example:

Arithmetic: expression$_1$ $\rightarrow$ expression$_2$ (+|-|*|/) expression$_3$



Abstract Syntax (Scott Notation)

A: $b \rightarrow \alpha$ where

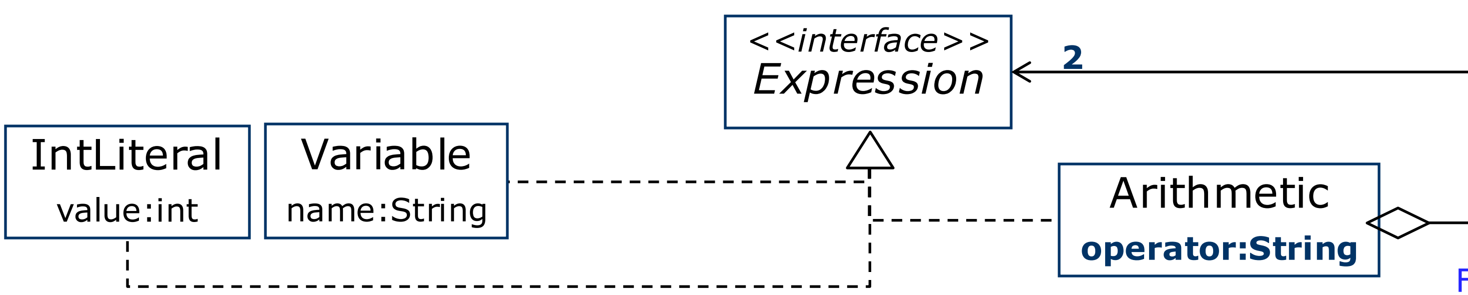
- A is the concrete (i.e., non-abstract) parent node in the AST (the dynamic type of the parent node in the tree)
- α is the sequence of child nodes

Example:

Arithmetic: expression$_1$ $\rightarrow$ **expression$_2$ (+|-|*|/) expression$_3$**

Variable: expression $\rightarrow$ ID

IntLiteral: expression $\rightarrow$ INT_CONSTANT



Abstract Syntax (Scott Notation)

A: $b \rightarrow \alpha$ where

- A is the concrete (i.e., non-abstract) parent node in the AST (the dynamic type of the parent node in the tree)
- α is the sequence of child nodes
- b is the parent of α
  - b **must be reachable** (<u>used as a non-terminal in another production</u>, unless it is the root node (i.e., the start symbol))
  - It may be a generalization of A (i.e., a supertype of A)

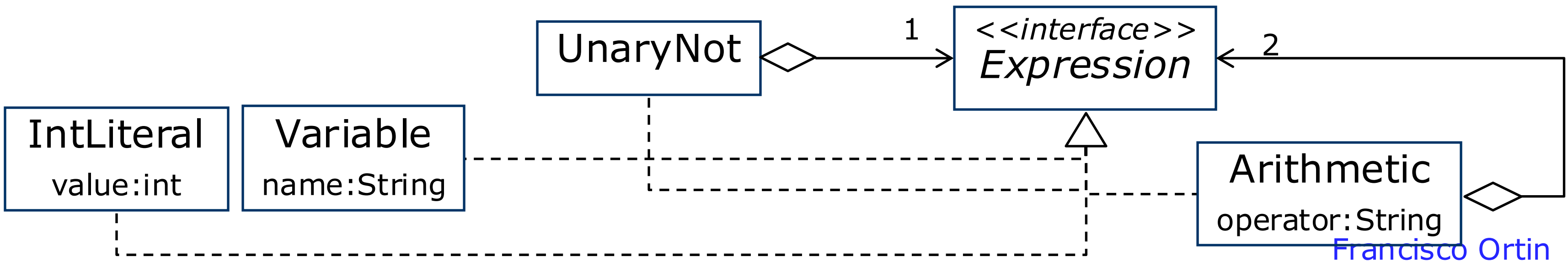
<u>Example</u>: **expression_1**

Arithmetic: arithmetic $\rightarrow$ expression_2 (+|-|*|/) expression_3

UnaryNot: expression_1 $\rightarrow$ expression_2

Variable: expression $\rightarrow$ ID

IntLiteral: expression $\rightarrow$ INT_CONSTANT



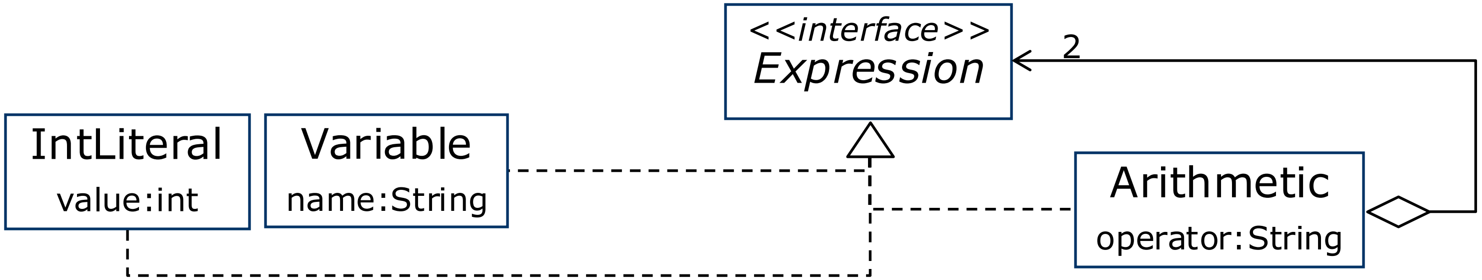
Abstract Syntax (Example)

- Given the following concrete syntax

```
expression: expression ('*'|'/') expression
          | expression ('-'|'+') expression
          | ID
          | INT_CONSTANT
          ;
```

<u>Question</u>: Why is it OK for this abstract grammar to be ambiguous?

- And its corresponding AST class diagram:



- The following abstract grammar represents the same language (**it is another representation of the AST**)

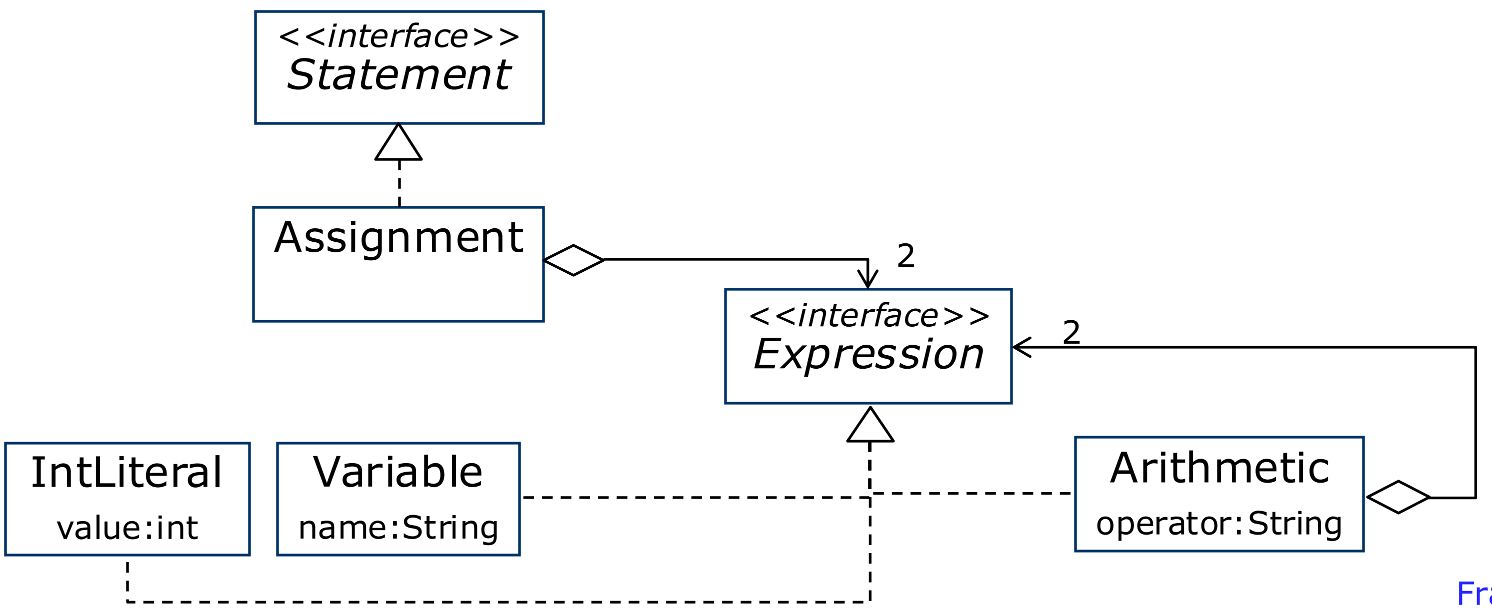
Arithmetic: $expression_1 \rightarrow expression_2$ (+|-|*|/) $expression_3$
Variable: expression $\rightarrow$ ID
IntLiteral: expression $\rightarrow$ INT_CONSTANT

Question

- Is the following abstract grammar correct?

Statement:	assignment $\rightarrow$ expression$_1$ expression$_2$
Arithmetic:	arithmetic $\rightarrow$ expression$_1$ (+|-|*|/) expression$_2$
Variable:	variable $\rightarrow$ ID
IntLiteral:	expression $\rightarrow$ INT_CONSTANT

…



Building the AST

- The **syntactic analyzer** (parser) has two objectives:
  1. Checking whether the tokens represent a valid sequence ⇒ We do that with ANTLR
  2. Building an AST that represents the structure of the source program
- Once we have the design of the AST for our language (e.g., lab 02)

  How do we build the AST at parsing?
- By using **embedded actions** in ANTLR
  - ANTLR *Listeners* (Observer) can also be used but they are less powerful

Embedded Actions

- **Embedded actions** are Java (C#, Python…) code between { and }
- The code is executed after recognizing the previous symbols in the production (recall, ANTLR is LL)
- Example

```
list: { System.out.print("1"); }
      ID { System.out.print("2"); }
      (',' ID { System.out.print("3"); } )*
      { System.out.println("4"); }
      ;
```

- Question: What sequence of actions? are executed for the program `a, b, c`?

Bibliography

- Alfred V. Aho, Monica S. Lam. Compilers: Principles, Techniques, and Tools. Addison Wesley, 2nd Edition, 2006.
- Terence Parr. The Definitive ANTLR 4 Reference, 2nd edition. Pragmatic Bookshelf, 2013.
- David A. Watt. Programming Language Processors in Java: Compilers and Interpreters. Prentice Hall, 2000.
- Michael L. Scott. Programming Language Pragmatics. Morgan Kaufmann, 4th edition 2015.