02141 Computer Science Modelling
Context-Free Languages

CFL3: Abstract Syntax and Parser Generators

Last Time on Context-Free Languages...

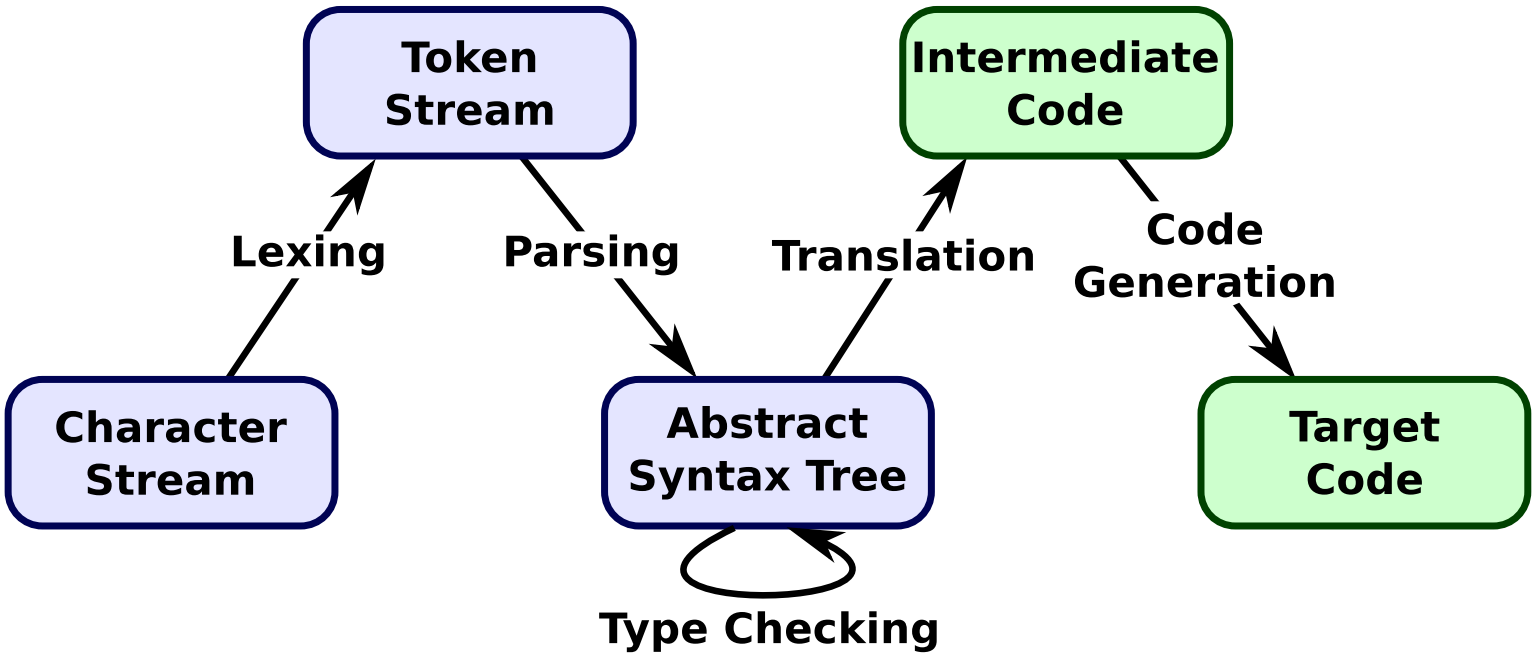
- Semi-structured data formats (e.g. XML, JSON) are often context-free languages!
- A grammar called **ambiguous** if there exists more than one **parse tree** for a word of the language.
- Ambiguities lead to problems when working with the parsed string. Always try to resolve them!
- Often we can resolve ambiguities without changing the language:
  - stratification to enforce precedences
  - limiting recursion to enforce associativity
- There are some context-free languages that are **inherently ambiguous**: there is no unambiguous grammar for the language.

Parsers and Compilers

http://xkcd.com/303/

Parsers and Compilers

The typical workflow of a compiler looks like this



Interpreters, type checkers, program analyzers, model checkers, and all that very often have a similar workflow.

Lexing: From Characters to Tokens

Consider a fragment of a program:

```
if (x == 42) { y = "42 is the answer"; }
```

Lexing: From Characters to Tokens

Consider a fragment of a program:

```
if (x == 42) { y = "42 is the answer"; }
```

To the computer, this is just a **stream of characters**:

```
i f ␣ ( x ␣ = = ␣ 4 2 ) ␣ { ␣ y ␣ = ␣ "
4 2 ␣ i s ␣ t h e ␣ a n s w e r " ; ␣ }
```

Lexing: From Characters to Tokens

Consider a fragment of a program:

```
if (x == 42) { y = "42 is the answer"; }
```

To the computer, this is just a **stream of characters**:

```
i f ␣ ( x ␣ = = ␣ 4 2 ) ␣ { ␣ y ␣ = ␣ "
4 2 ␣ i s ␣ t h e ␣ a n s w e r " ; ␣ }
```

We want to convert this into a **stream of tokens**, which are more meaningful:

```
IF LPAREN ID("x") EQ NUM(42) RPAREN LBRACE ID("y") ASSIGN
      STRING("42 is the answer") SEMICOLON RBRACE
```

Lexing: From Characters to Tokens

Consider a fragment of a program:

```
if (x == 42) { y = "42 is the answer"; }
```

To the computer, this is just a **stream of characters**:

```
i f ␣ ( x ␣ = = ␣ 4 2 ) ␣ { ␣ y ␣ = ␣ "
4 2 ␣ i s ␣ t h e ␣ a n s w e r " ; ␣ }
```

We want to convert this into a **stream of tokens**, which are more meaningful:

```
IF LPAREN ID("x") EQ NUM(42) RPAREN LBRACE ID("y") ASSIGN
      STRING("42 is the answer") SEMICOLON RBRACE
```

This is the job of the lexer!

Lexing: From Characters to Tokens

We can identify tokens...

Lexing: From Characters to Tokens

We can identify tokens... using **regular expressions**!

Lexing: From Characters to Tokens

We can identify tokens... using **regular expressions**!

Examples:

- Positive integers: `[0-9]+`

Lexing: From Characters to Tokens

We can identify tokens... using **regular expressions**!

Examples:

- Positive integers: `[0-9]+`
- Identifiers: `[A-Za-z][A-Za-z0-9]*`

Lexing: From Characters to Tokens

We can identify tokens... using **regular expressions**!

Examples:

- Positive integers: `[0-9]+`
- Identifiers: `[A-Za-z][A-Za-z0-9]*`
- Keywords: `if`, `while`, etc.

Lexing: From Characters to Tokens

We can identify tokens... using **regular expressions**!

Examples:

- Positive integers: `[0-9]+`
- Identifiers: `[A-Za-z][A-Za-z0-9]*`
- Keywords: `if`, `while`, etc.
- Whitespace: `[ \t\r\n]+`

Lexing: From Characters to Tokens

We can identify tokens... using **regular expressions**!

Examples:

- Positive integers: `[0-9]+`
- Identifiers: `[A-Za-z][A-Za-z0-9]*`
- Keywords: `if`, `while`, etc.
- Whitespace: `[ \t\r\n]+`

We **match** a regular expression from the character stream, and then create a **token** from it (or discard it).

Lexing: From Characters to Tokens

We can identify tokens... using **regular expressions**!

Examples:

- Positive integers: `[0-9]+`
- Identifiers: `[A-Za-z][A-Za-z0-9]*`
- Keywords: `if`, `while`, etc.
- Whitespace: `[ \t\r\n]+`

We **match** a regular expression from the character stream, and then create a **token** from it (or discard it). Note that * and + are interpreted **greedily**.

Lexing: From Characters to Tokens

We can identify tokens... using **regular expressions**!

Examples:

- Positive integers: `[0-9]+`
- Identifiers: `[A-Za-z][A-Za-z0-9]*`
- Keywords: `if`, `while`, etc.
- Whitespace: `[ \t\r\n]+`

We **match** a regular expression from the character stream, and then create a **token** from it (or discard it). Note that `*` and `+` are interpreted **greedily**.

The lexer provides a function `lex()` which returns the next token from the input stream. This is used by the **parser**.

Parser Generators

Writing a parser may be challenging:

- Simple parsing algorithms work only on a limited subset of grammars, and hence require you to transform your beautiful grammar into an ugly one, hence hampering understandability, extendability, etc.
- Parsing algorithms that admit (almost) any practical grammar are very sophisticated, and error-prone.

Parser Generators

Writing a parser may be challenging:

- Simple parsing algorithms work only on a limited subset of grammars, and hence require you to transform your beautiful grammar into an ugly one, hence hampering understandability, extendability, etc.
- Parsing algorithms that admit (almost) any practical grammar are very sophisticated, and error-prone.

Parser generators can save your day!

Parser Generators

Writing a parser may be challenging:

- Simple parsing algorithms work only on a limited subset of grammars, and hence require you to transform your beautiful grammar into an ugly one, hence hampering understandability, extendability, etc.
- Parsing algorithms that admit (almost) any practical grammar are very sophisticated, and error-prone.

Parser generators can save your day!

A parser generator is a tool that, given a grammar, automatically generates the (lexer and) parser for you.

There are toons of parser generators (see `https://en.wikipedia.org/wiki/Comparison_of_parser_generators`)

Parser Generators

That sounds nice... but are there downsides?

Parser Generators

That sounds nice... but are there downsides?

- Not all parser generators support the same languages, hence hampering multi-language development.
- Not all parser generators use the same language to describe grammars: you have to learn yet another language.
- A parser generator usually produces a parser based on a specific parsing algorithm, which may impose limitations on the grammars that it will easily digest.
- Some parser generators automatically build syntax structures that you may have to get acquainted with, others give you more freedom but also more work to do.

Two species of parser generators

The `ANTLR` family of generators of top-down parsers.

The `yacc` family of generators of bottom-up parsers: `FsLexYacc`, `bison`, etc.

Generating Abstract Syntax

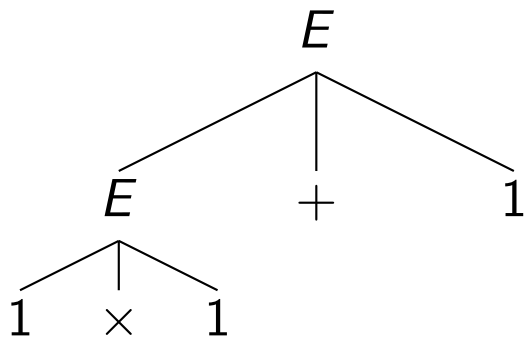
If we want to do something with the parsed string it is very convenient to store it in a suitable data structure, often called **abstract syntax**:

- A data structure that holds the relevant aspects of the parsed information;
- In particular, this data structure must represent the structure of the data and its contents;
- This is usually of a tree shape (hence the name AST), e.g. when parsing an arithmetic expression.
- Abstract syntax may not be a tree, e.g. when parsing a list we could just use a suitable list data structure.
- ASTs are similar to parse trees, but usually throw away irrelevant syntactic details (parenthesis, syntactic sugar, derived operators, etc.).

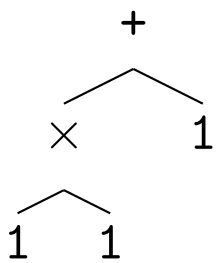
The original string and its parse tree(s) are often called concrete syntax.

Abstract and Concrete Syntax

Example of a parse tree:



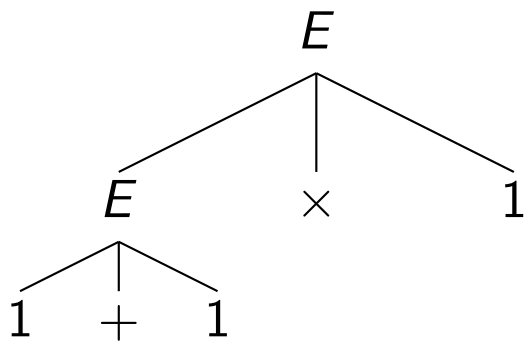
Removing irrelevant details we obtain the abstract syntax tree:



Abstract and Concrete Syntax

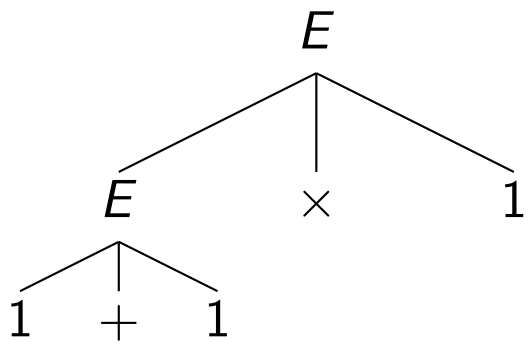
Abstract and Concrete Syntax

If $\times$ has higher precedence than $+$, string $1 + 1 \times 1$ should **not** be parsed as:

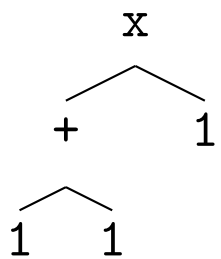


Abstract and Concrete Syntax

If $\times$ has higher precedence than $+$, string $1 + 1 \times 1$ should **not** be parsed as:

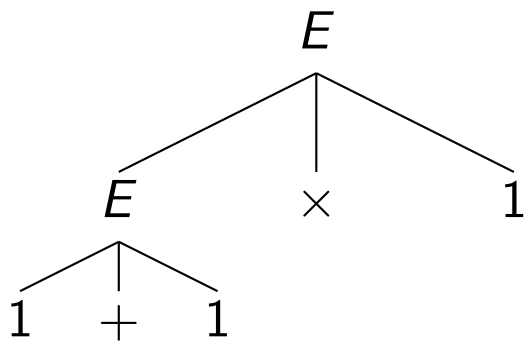


Can we ever have this abstract syntax tree?

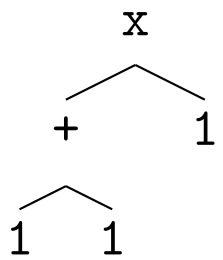


Abstract and Concrete Syntax

If $\times$ has higher precedence than $+$, string $1 + 1 \times 1$ should **not** be parsed as:



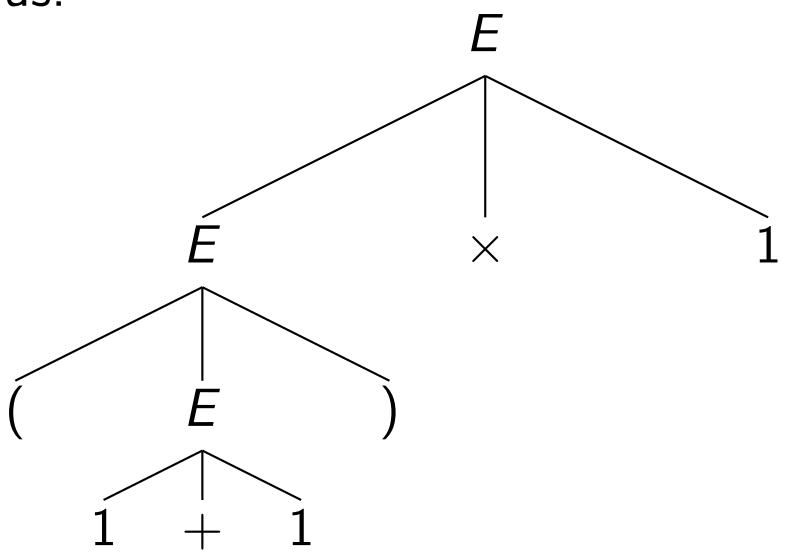
Can we ever have this abstract syntax tree?



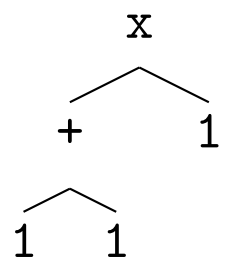
Yes we can!

Abstract and Concrete Syntax

String $(1+1)\times 1$ should be parsed as:



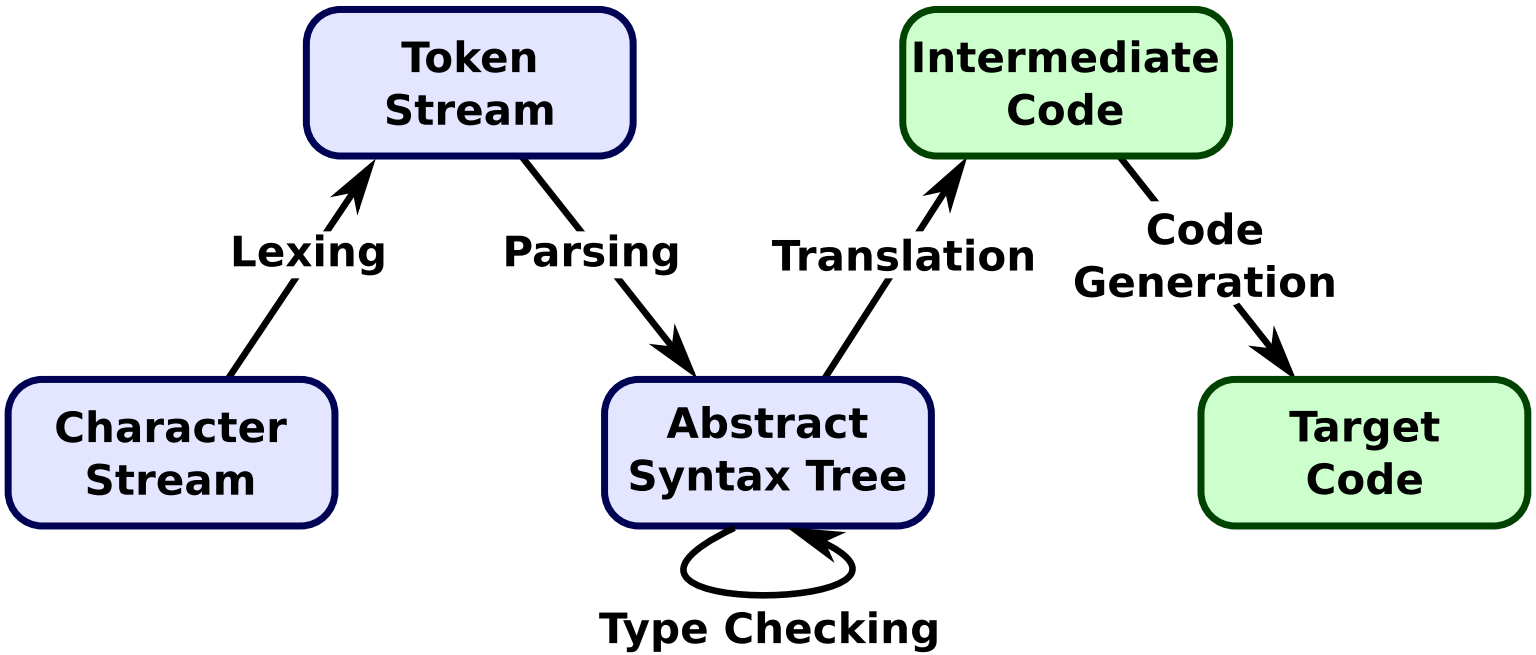
This gives us the abstract syntax:



The abstract syntax is unambiguous by construction.
Ambiguity is only a problem of the concrete syntax/parsing.
Designing the abstract syntax we do not need to care about ambiguity!

Generating Abstract Syntax

The typical workflow of a compiler looks like this



Interpreters, type checkers, program analyzers, model checkers, and all that very often have a similar workflow.

Generating Abstract Syntax for OO languages

Consider our grammar for arithmetic expressions

$$\begin{array}{lcccccccc} E & \rightarrow & N & | & E + E & | & E \times E & | & (E) \\ N & \rightarrow & 0 & | & 1 & | & N0 & | & N1 \end{array}$$

Generating Abstract Syntax for OO languages

Consider our grammar for arithmetic expressions

$$\begin{array}{lcccccccc} E & \rightarrow & N & | & E + E & | & E \times E & | & (E) \\ N & \rightarrow & 0 & | & 1 & | & N0 & | & N1 \end{array}$$

We implement an **abstract Java class** for expressions:

```
public abstract class ArithExpr { }
```

Generating Abstract Syntax for OO languages

Consider our grammar for arithmetic expressions

$$\begin{array}{lcl|l|l|l} E & \rightarrow & N & E+E & E \times E & (E) \\ N & \rightarrow & 0 & 1 & N0 & N1 \end{array}$$

We implement an **abstract Java class** for expressions:

```
public abstract class ArithExpr { }
```

Each production in the grammar corresponds to a **subclass** of `ArithExpr`.

```
public class PlusExpr extends ArithExpr {
    private ArithExpr expression1;
    private ArithExpr expression2;

    public PlusExpr(ArithExpr e1, ArithExpr e2) {
        expression1 = e1;
        expression2 = e2;
    }
}
```

Generating Abstract Syntax for OO languages

Consider our grammar for arithmetic expressions

$$\begin{array}{lcccccccc} E & \rightarrow & N & | & E+E & | & E \times E & | & (E) \\ N & \rightarrow & 0 & | & 1 & | & N0 & | & N1 \end{array}$$

We implement an **abstract Java class** for expressions:

```
public abstract class ArithExpr { }
```

Each production in the grammar corresponds to a **subclass** of `ArithExpr`.

```
public class NumExpr extends ArithExpr {

    private int value;
    public NumExpr(int v) {
        value = v;
    }

}
```

Generating Abstract Syntax for OO languages

Consider our grammar for arithmetic expressions

$$\begin{array}{lcccccccc} E & \rightarrow & N & | & E + E & | & E \times E & | & (E) \\ N & \rightarrow & 0 & | & 1 & | & N0 & | & N1 \end{array}$$

We implement an **abstract Java class** for expressions:

```
public abstract class ArithExpr { }
```

No new subclass for $E \rightarrow (E)$ why?

Generating Abstract Syntax for OO languages

Consider our grammar for arithmetic expressions

$$\begin{array}{lcl|l|l|l|l} E & \to & N & E + E & E \times E & (E) \\ N & \to & 0 & 1 & N0 & N1 \end{array}$$

We implement an **abstract Java class** for expressions:

```
public abstract class ArithExpr { }
```

No new subclass for $E \to (E)$ why?

Parentheses are part of the concrete syntax to structure terms.
In the abstract syntax, the `ArithExpr` data type is already structured!

Generating Abstract Syntax for OO languages

Example: $1 + 1 \times 1$ in the **abstract syntax**:

```
ArithExpr e = new PlusExpr(  new NumExpr(1),
                             new MultExpr(  new NumExpr(1),
                                            new NumExpr(1)));
```

Generating Abstract Syntax for OO languages

Example: $1 + 1 \times 1$ in the **abstract syntax**:

```
ArithExpr e = new PlusExpr(  new NumExpr(1),
                             new MultExpr(  new NumExpr(1),
                                            new NumExpr(1)));
```

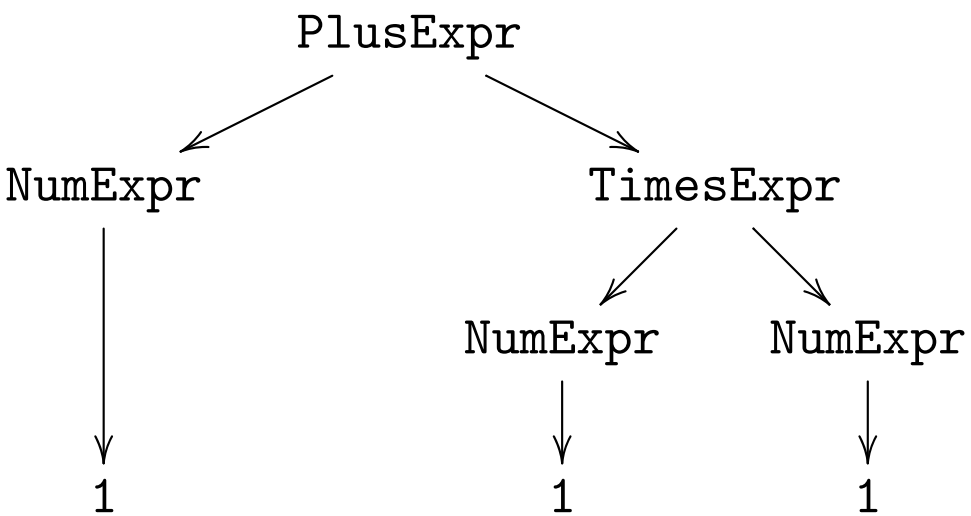
IMPORTANT!: An **abstract syntax tree** is different from a **parse tree**.

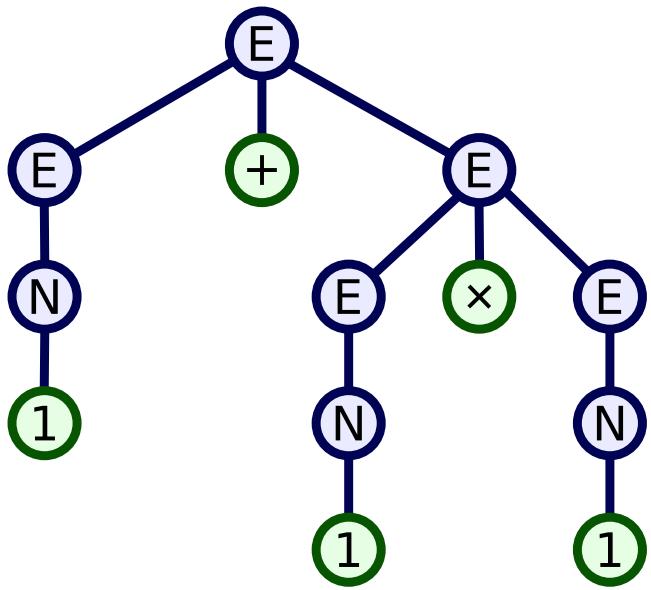
Generating Abstract Syntax for OO languages

Example: $1 + 1 \times 1$ in the **abstract syntax**:

```
ArithExpr e = new PlusExpr(   new NumExpr(1),
                              new MultExpr(   new NumExpr(1),
                                              new NumExpr(1)));
```

IMPORTANT!: An **abstract syntax tree** is different from a **parse tree**.





Generating Abstract Syntax for Functional Languages

In functional programming languages, abstract syntax is often based on algebraic datatypes.

Concrete Syntax

$$
\begin{array}{lll}
E \rightarrow & E + E \\
\mid & E \times E \\
\mid & N \\
\mid & (E)
\end{array}
$$

Abstract Syntax

```
type E  =  Plus of E * E
        |  Mult of E * E
        |  Num of Int
--- no data constructor for (E)
```

Exercise 3.1: abstract syntax for boolean expressions

Consider again the following grammar for boolean expressions:

$$B \quad \rightarrow \quad \texttt{true} \mid B \ \texttt{or} \ B \mid \texttt{not} \ B \mid (B)$$

(a): Design a set of classes/structures that are suitable to represent abstract syntax for boolean expressions generated by the grammar, and show a program statement to build the abstract syntax for expression:

```
not ( true or true )
```

Use Java or any other imperative language of your choice.

(b): Design a functional datatype that is suitable to represent abstract syntax for boolean expressions generated by the grammar, and show a statement that represents the above expression.

Use F# or any other functional language of your choice.

Generating Abstract Syntax

But how can we generate abstract syntax? The approach can be roughly sketched as follows:

- adopt the functional/inference/bottom-up perspective of grammars where productions $A \to \gamma(B_1, .., B_n)$ are seen as functions $\gamma : B_1 \times .. \times B_n \to A$;
- decorate the productions with code or annotations to construct the structure of type A starting from the structures obtained from B_1 .. B_n.

For example, in the grammar for arithmetic expresions the production

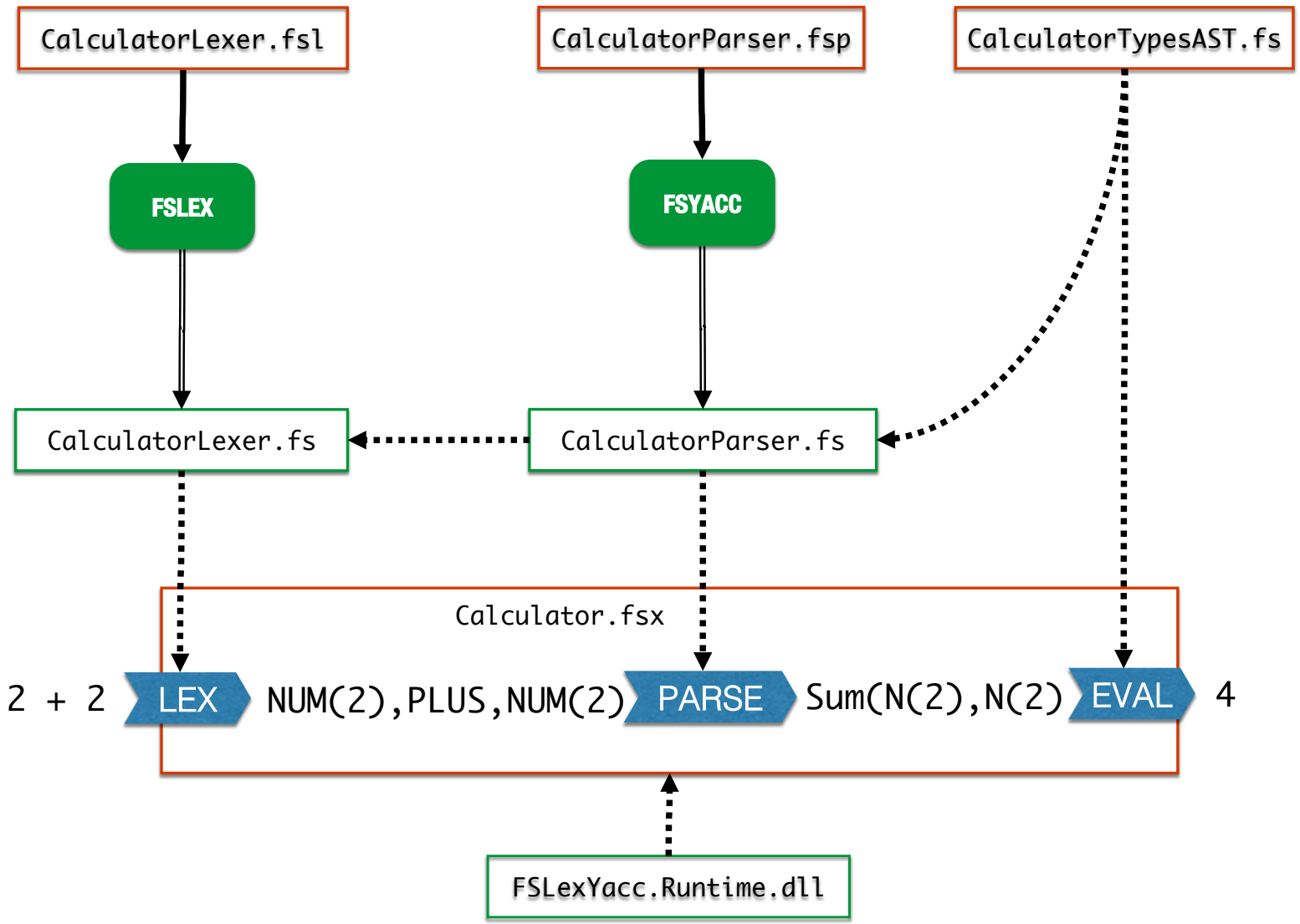
$$E \to E \times E$$

can be decorated (using ANTL3-like notation) as

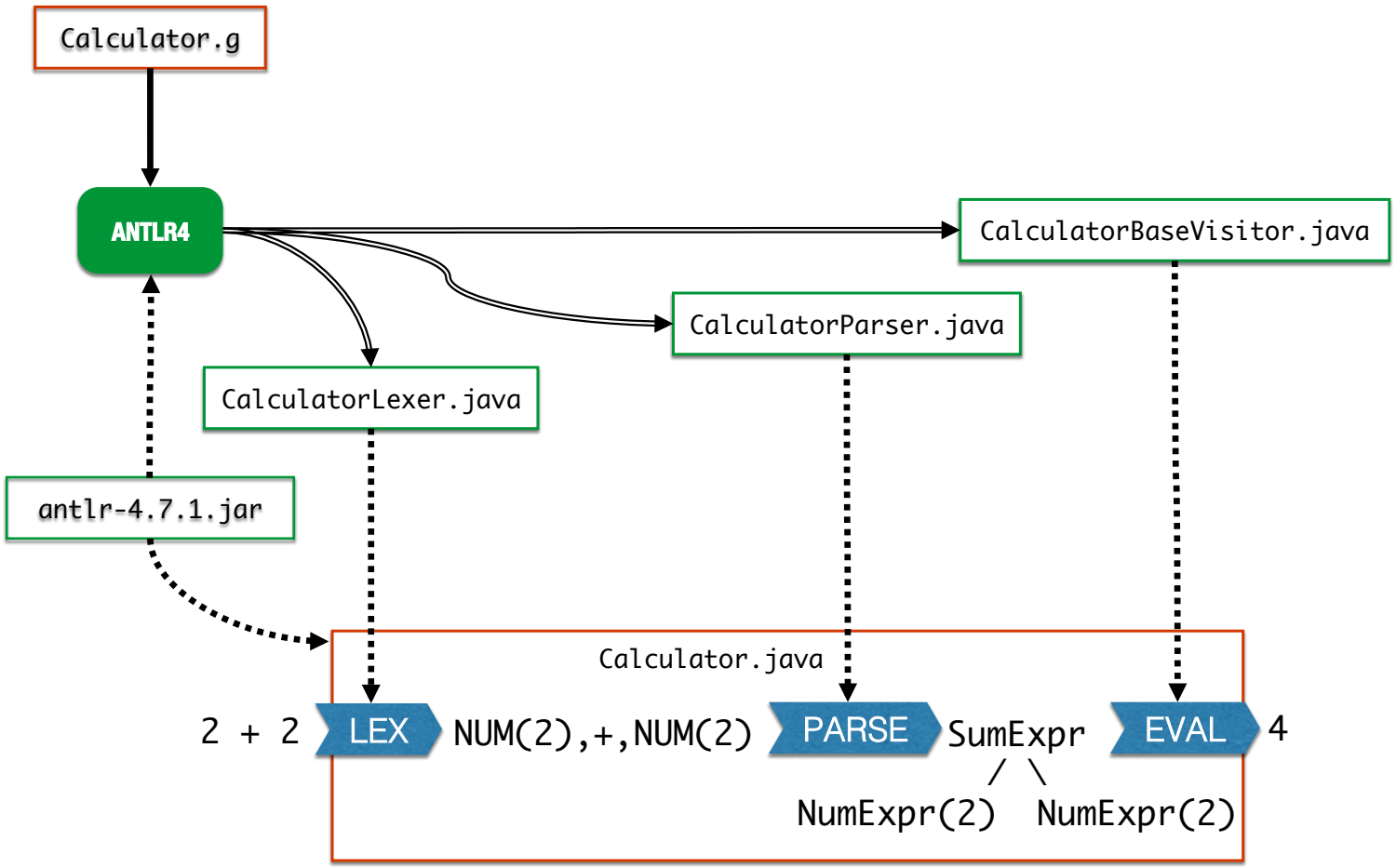
```
E returns [ArithExpr a] →
    e1 = E  ×
    e2 = E
    { $a = new PlusExpr(e1,e2);}
```

DEMO TIME!

F#/FsLexYacc workflow



JAVA/ANTLR4 workflow



NOTE: We don't use JAVA/ANTLR4 this year, but this diagram and example can be useful for you in other courses.

Exercise session

1. Go to `https://gitlab.gbar.dtu.dk/02141` and follow the instructions to install `F#/FSLexYacc`.
2. Get the `Hello` example to work: `https://gitlab.gbar.dtu.dk/02141/mandatory-assignment/tree/master/hello`
3. Get the `Calculator` example to work: `https://gitlab.gbar.dtu.dk/02141/mandatory-assignment/tree/master/calculator`.
4. Extend the `Calculator` example with additional arithmetic operations: roots, logarithms, etc. You can have a look at scientific calculators for inspiration. Most likely, you will have to
   1. Extend the lexer with new tokens and rules in file `CalculatorLexer.fsl`.
   2. Extend the parser with the new tokens and their precedence and associativity rules in file `CalculatorLexer.fsp`.
   3. Extend the parser with new grammar rules in file `CalculatorLexer.fsp`.
   4. Extend the abstract syntax: type `expr` in file `CalculatorTypesAST.fs`.
   5. Extend the evaluation function `eval` in file `CalculatorTypesAST.fs`