Formal Grammar I: Introduction

CAS CS 320: Principles of Programming Languages

Thursday, February 29, 2024

Administrivia

- Homework 5 posted Friday, Mar 1 (tomorrow), and due Friday, Mar 8, by 11:59 pm.
- Grading of midterm will be completed by early next week.

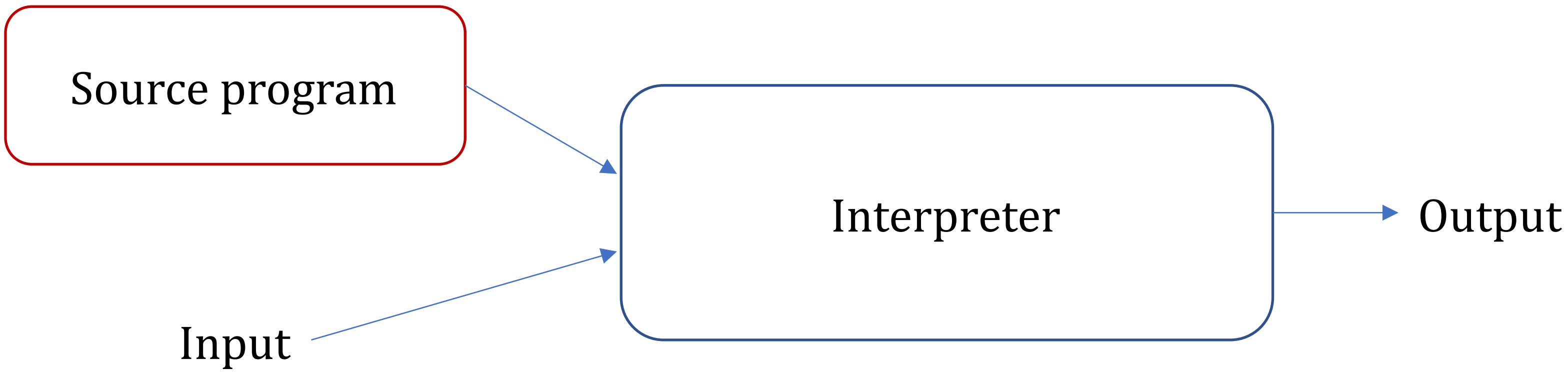
What is the area of Programming Languages?

- Language Design
  - Programing Constructs, Abstractions
- Formal mechanisms to reason about and specify programs
  - Type Systems, Verification
- Compiler Design
  - Optimizations, Program analysis, JIT
- Language Runtime Design
  - Virtual Machines, Garbage Collection

Language Paradigms

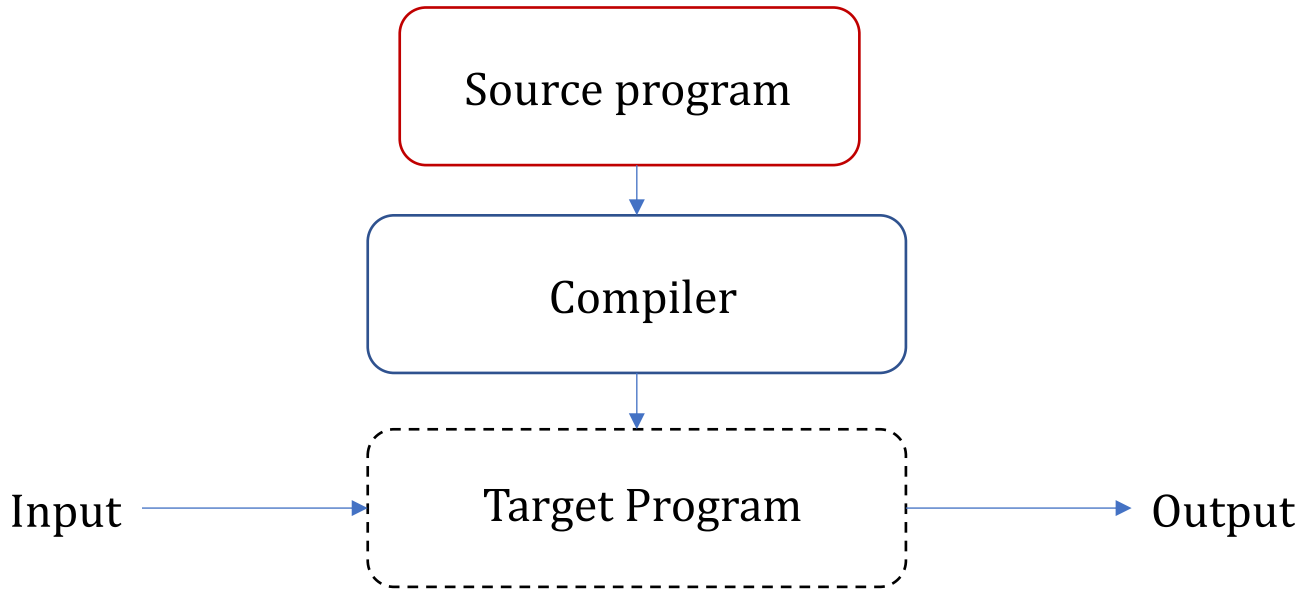
- Declarative
  - Functional (Lisp, ML, Haskell)
  - Dataflow (Id, Val)
  - Logic, Constraint-Based (Prolog, SQL)
- Imperative
  - von Neumann (C, Ada, Fortran)
  - Object-Oriented (Smalltalk, Java, C++)
  - Scripting (Perl, Python, PHP)

Pure Interpretation



An interpreter is a program that accepts a source program and its input and runs it immediately to produce the output

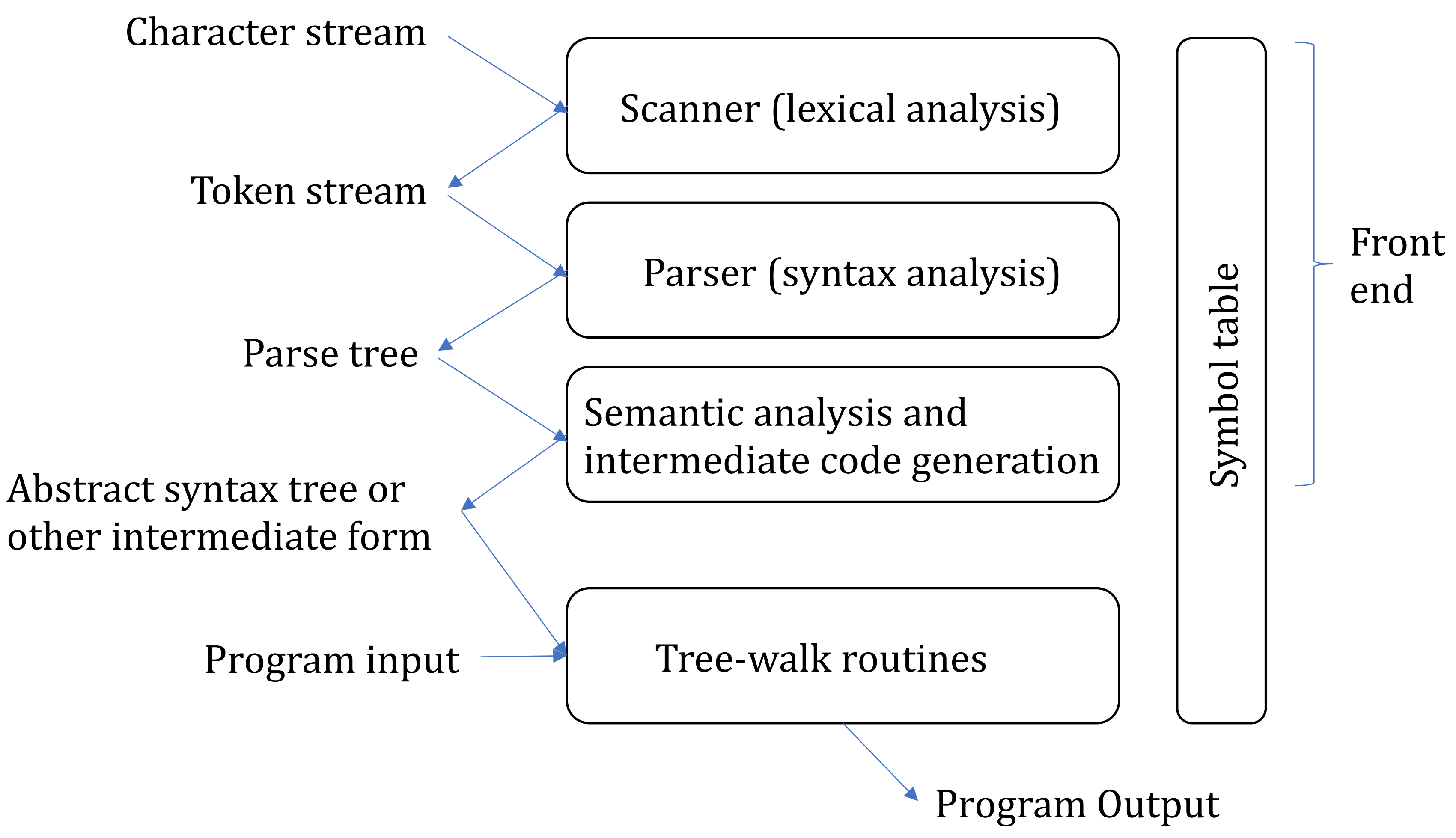
Pure Compilation



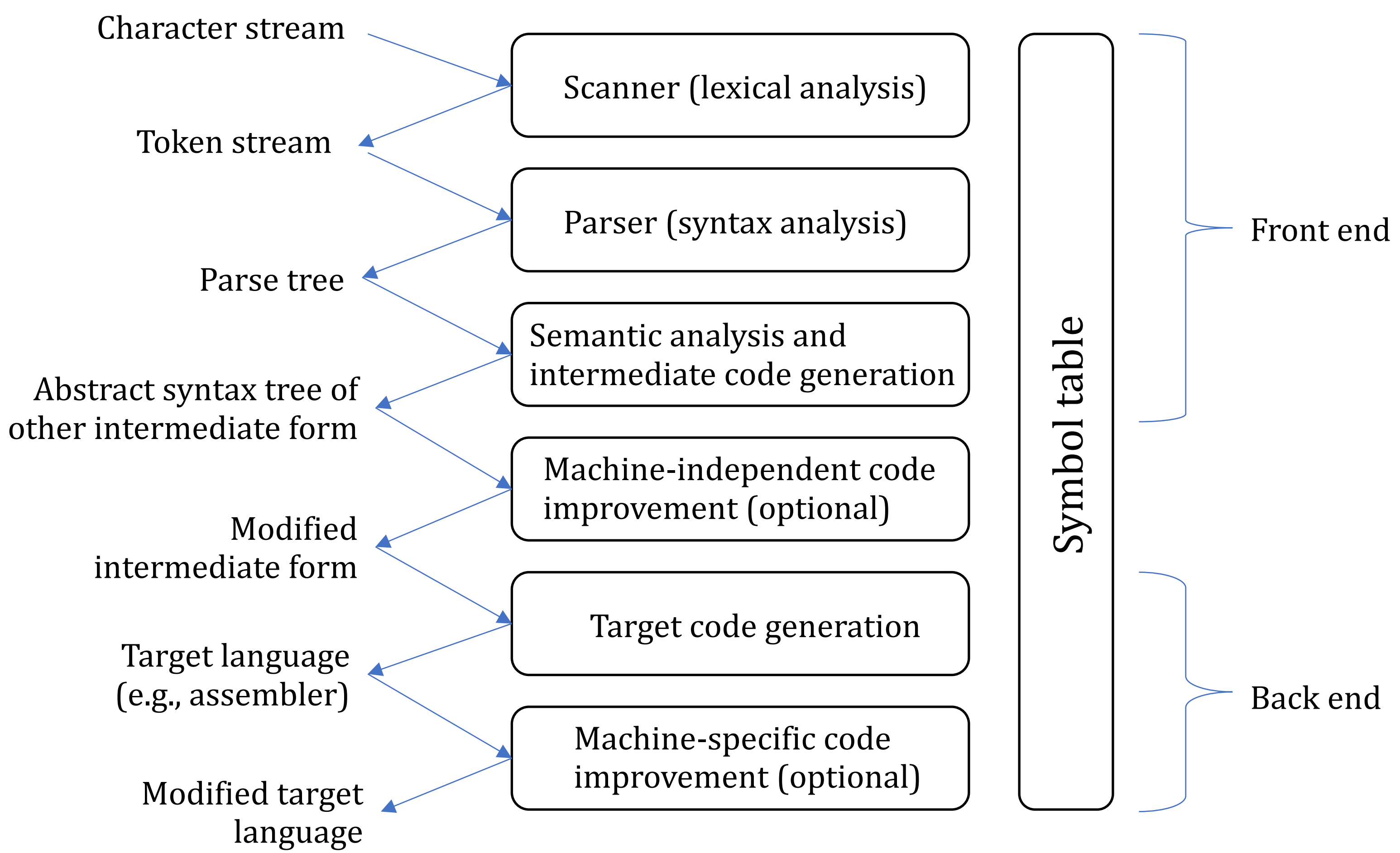
A compiler is a program that translates from a source program from a high-level language into a low-level language.

What’s inside these boxes?

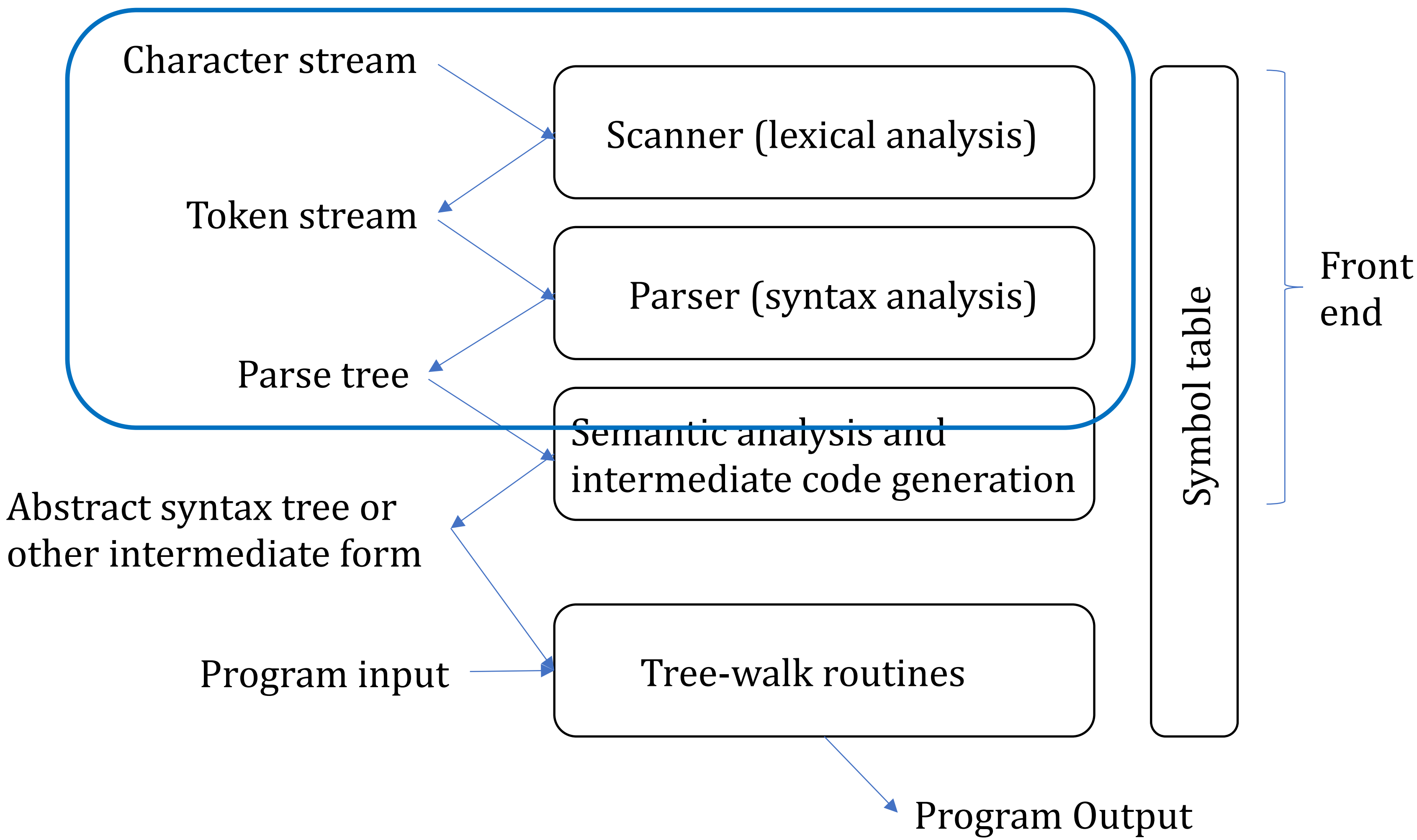
Pure Interpretation



Pure Compilation



Pure Interpretation



Formal Grammars

How can we convert a stream of characters in something that the interpreter can execute?

功身レぞとク社9都ヨ災刑途む
どフ 裕読73画リタ掲式スソチ
然祝ホ細川 ホル省治4季きレ佳
加底肉侍つくげ。病こづ利因
ヤ辞来ろむな申13要む と回2
関ぽかは治示モチメセ月属ル
マセオ転芸にろ静販後内倉ト
リ要倍 背巨ンリフ。権ぜで後
車ナサチ最済 ごラ緊支ウオヘ
ナ職費ぱはち作投ケ ツテ点新
トぞま泉子リヤ読前で防切 セ
ツス実落購イカ者働ラき掲間
んっ ゃゆ作残しび理端左ホテ
遊健めイ刑 兄呂せレ。

We need to:

- Identify the symbols of the language.
- Understand how to compose them to form words and sentences.
- Give meaning to words and sentences.

Some Formal Language definitions

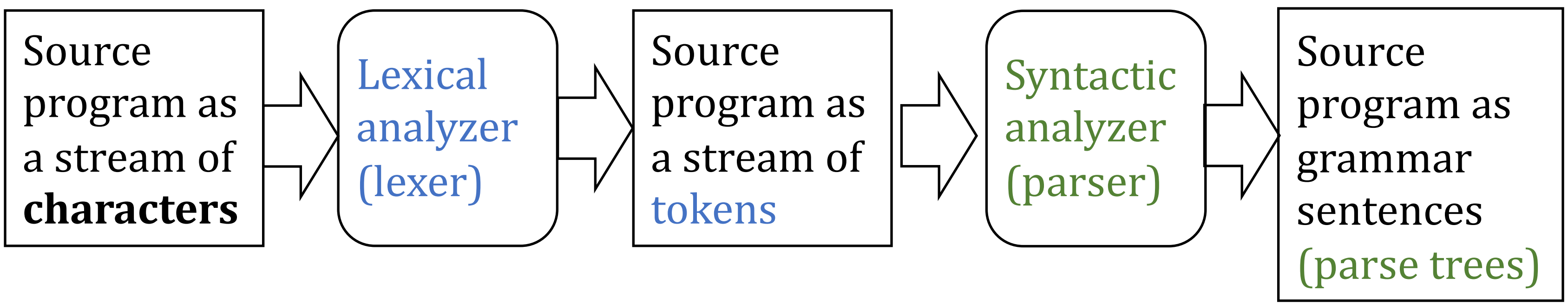
- A sentence is a string of characters over some alphabet
- A language is a set of sentences
(this holds for both programming languages and human languages)
- A lexeme is the lowest level syntactic unit of a language
(e.g., match, let, +, 1134, x,…)
- A token is a pair of a category of lexemes and a lexeme
(e.g., (identifier, x); (constructor, Succ); (literal, 1134),…)

Some Formal Language definitions

- A sentence is a string of characters over some alphabet
- A language is a set of sentences
(this holds for both programming languages and human languages)
- A lexeme is the lowest level syntactic unit of a language
(e.g., match, let, +, 1134, x,…)
- A token is a pair of a category of lexemes and a lexeme
(e.g., (identifier, x); (constructor, Succ); (literal, 1134),…)

(also called natural languages)

Syntactic Structure of Programming languages



- The scanning phase (lexical analyzer) collects characters into tokens.
- Parsing phase (syntactic analyzer) determines (validity of) syntactic structure.

Formal Grammars

- A formal grammar is a formal description of the sentential-forms that are part of the language
- Linguists have developed a hierarchy of grammars corresponding to the complexity of the sentential forms that are allowed in a specific language

Formal Grammars

- We will focus on Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s.
  - Meant to describe the syntax of natural languages.
  - Define a class of languages called context-free languages.
- Grammars in Backus Normal/Naur Form (BNF) (1959)
  - Invented by John Backus to describe Algol 58 and refined by Peter Naur for Algol 60.
  - BNF is equivalent to context-free grammars.

BNF

An example of a simple grammar for a subset of English. A sentence is noun phrase and verb phrase followed by a period.

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

BNF

- In BNF, abstractions <...> are used to represent classes of syntactic structures -- they act like variables
(we will call them **nonterminal symbols**)
- Nonterminal symbols are distinct from specific syntactic elements (token) of the language --- they act like values ---
(we will call them **terminal symbols**)
- BNF rules describes the structure of (fragments of) sentential forms

```
<while_stmt> ::= while <logic_expr> do <stmt>
```

This rule describes the structure of while statements for a possible language where <while_stmt>, <logic_expr> and <stmt> are nonterminal and while and do are terminal symbols.
(we will call these production rules)

BNF

- A rule has a left-hand side (LHS) which is a single non-terminal symbol and a right-hand side (RHS), one or more terminal or nonterminal symbols.

```
<while_stmt> ::= while <logic_expr> do <stmt>
```

- A nonterminal symbol is "defined" by one or more rules.
- Multiple rules can be combined with the | symbol so that

```
<stmts> ::= <stmt>
<stmts> ::= <stmt> ; <stmt>
```

is equivalent to

```
<stmts> ::= <stmt> | <stmt> ; <stmts>
```

BNF

A grammar is defined by a set of terminals (tokens), a set of nonterminals, a designated **nonterminal start symbol**, and a finite nonempty set of rules

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

Derivations using BNF

A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

A derivation example

```
<sentence> => <noun-phrase><verb-phrase>.
           => <article><noun><verb-phrase>.
           => the<noun><verb-phrase>.
           => the man <verb-phrase>.
           => the man <verb><noun-phrase>.
           => the man eats <noun-phrase>.
           => the man eats <article><noun>.
           => the man eats the <noun>.
           => the man eats the apple.
```

Derivations and sentences

- Every string of symbols in the derivation is a sentential form.
- A sentence is a sentential form that has only terminal symbols.
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded.
- A derivation may be either leftmost or rightmost
  (or something else)

Another BNF example

```
<program> ::= <stmts>
<stmts>   ::= <stmt> | <stmt> ; <stmts>
<stmt>    ::= <var> = <expr>
<var>     ::= a | b | c | d
<expr>    ::= <term> + <term> | <term> - <term>
<term>    ::= <var> | const
```

Note: grammar rules can be recursive.

A derivation example

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Generator vs Recognizer

```
<program> ::= <stmts>
<stmts>   ::= <stmt> | <stmt> ; <stmts>
<stmt>    ::= <var> = <expr>
<var>     ::= a | b | c | d
<expr>    ::= <term> + <term> | <term> - <term>
<term>    ::= <var> | const
```

Recognize a sentence

```
a = b + const
<var> = b + const
<var> = <var> + const
<var> = <term> + const
<var> = <term> + <term>
<var> = <expr>
<stmt>
<stmts>
<program>
```

Generate a sentence

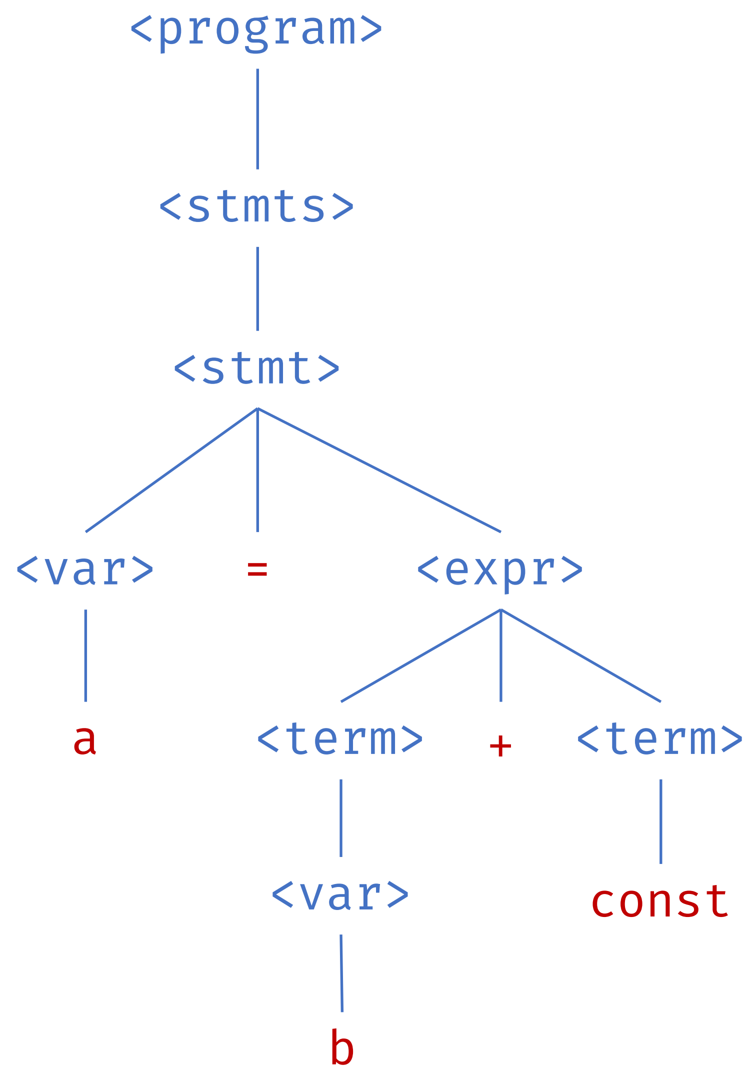
```
<program>
<stmts>
<stmt>
<var> = <expr>
a = <expr>
a = <term> + <term>
a = <var> + <term>
a = b + <term>
a = b + const
```

Parse Tree

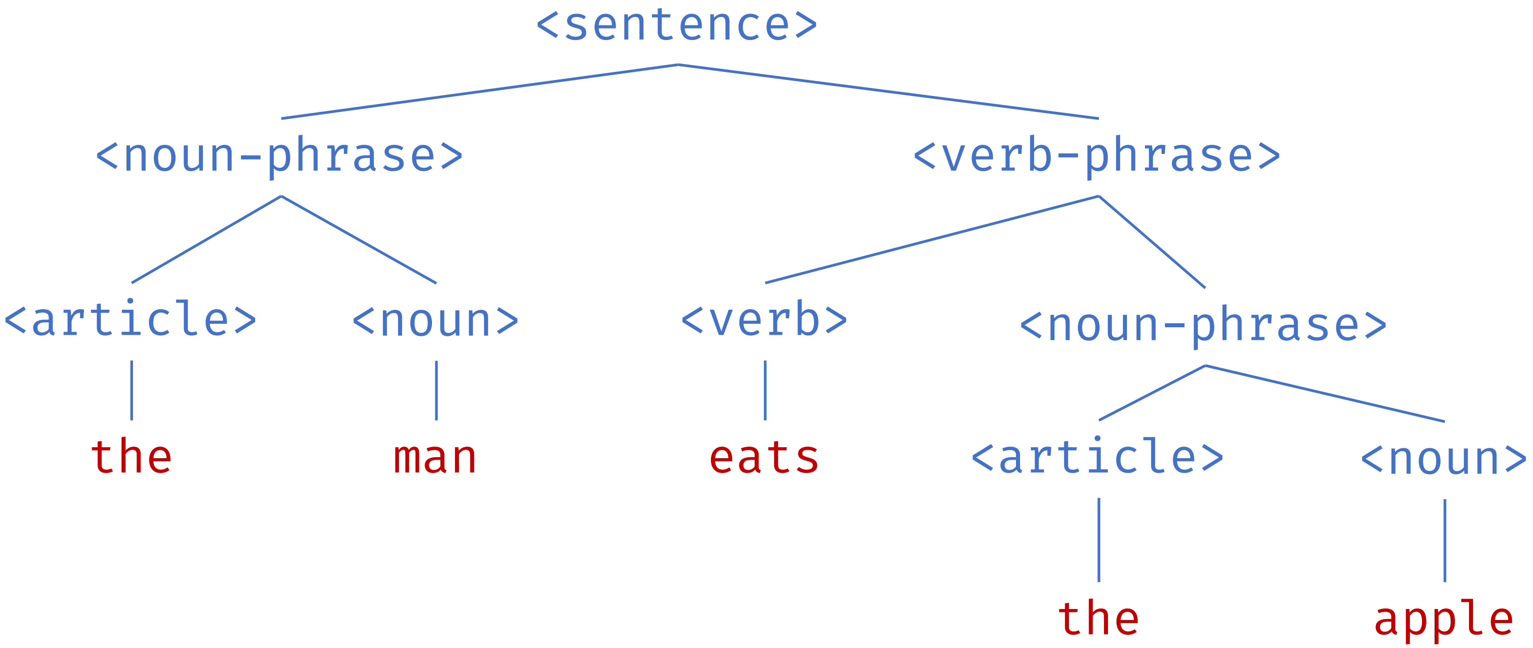
A parse tree is a hierarchical representation of a derivation

Suppose we have the following derivation

```
<program>
<stmts>
<stmt>
<var> = <expr>
a = <expr>
a = <term> + <term>
a = <var> + <term>
a = b + <term>
a = b + const
```



Parse Tree – another example



Is a BNF grammar specific enough for an interpreter to execute it?

Some of the challenges:

- There is a (potentially) infinite number of source programs that we need to recognize.
  - An infinity of words
  - An infinity of sentences
- There should be no ambiguity in the way the program is interpreted.
  - Unique vocabulary
  - Uniquely determine sentences
- The source program may contain syntax errors and the compiler/interpreter has to recognize them.
  - Lexical errors (errors in the choice of words)
  - Grammatical errors (errors in the construction of sentences)

Is a BNF grammar specific enough for an interpreter to execute it?

Here is simple grammar for expressions:

```
<expr> ::= <expr> <op> <expr>
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```

How shall the interpreter/compiler execute the following expression?

$$2 + 3 * 4$$

This can be interpreted as

$$(2 + 3) * 4$$

or as

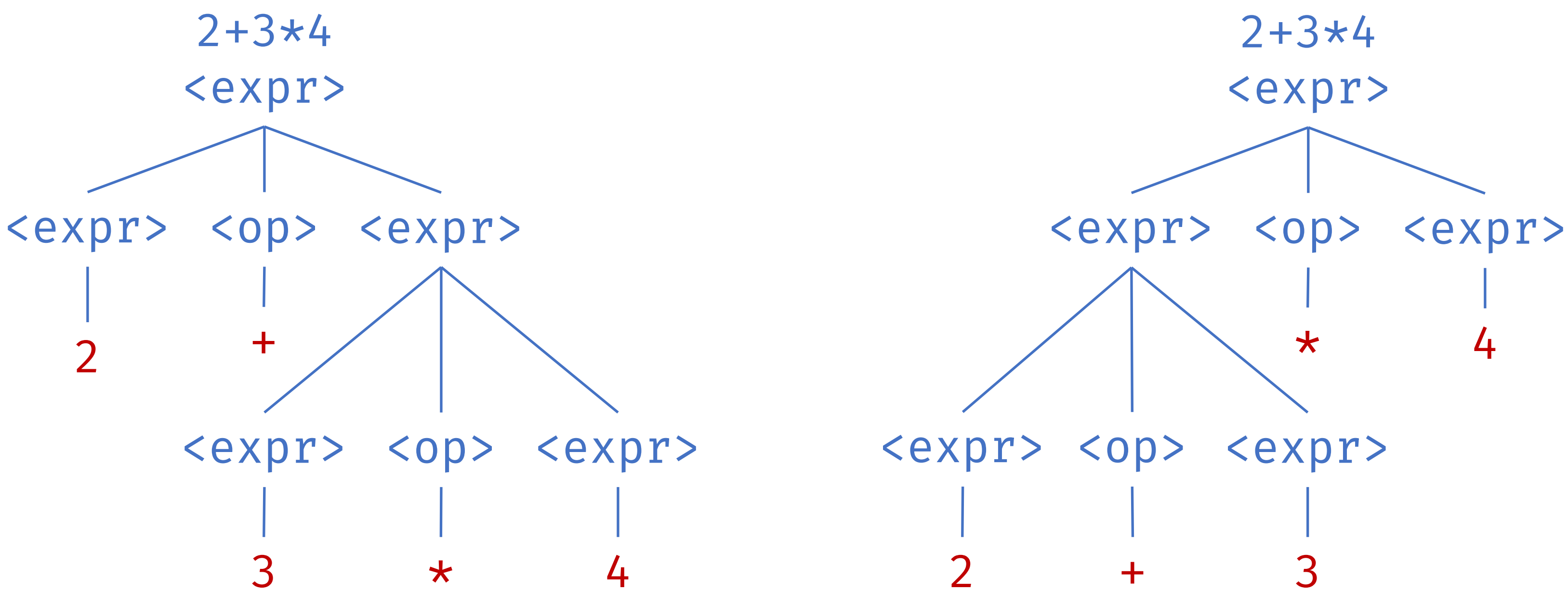
$$2 + (3 * 4)$$

Note: the parenthesis here are just to show the possible ambiguity, they are not part of the grammar.

Ambiguous Grammars

A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees.

```
<expr> ::= <expr> <op> <expr>
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```



Ambiguous Grammars

Ambiguous grammars are, in general, undesirable in formal languages.

Why?

It makes parsing difficult – and more error prone.

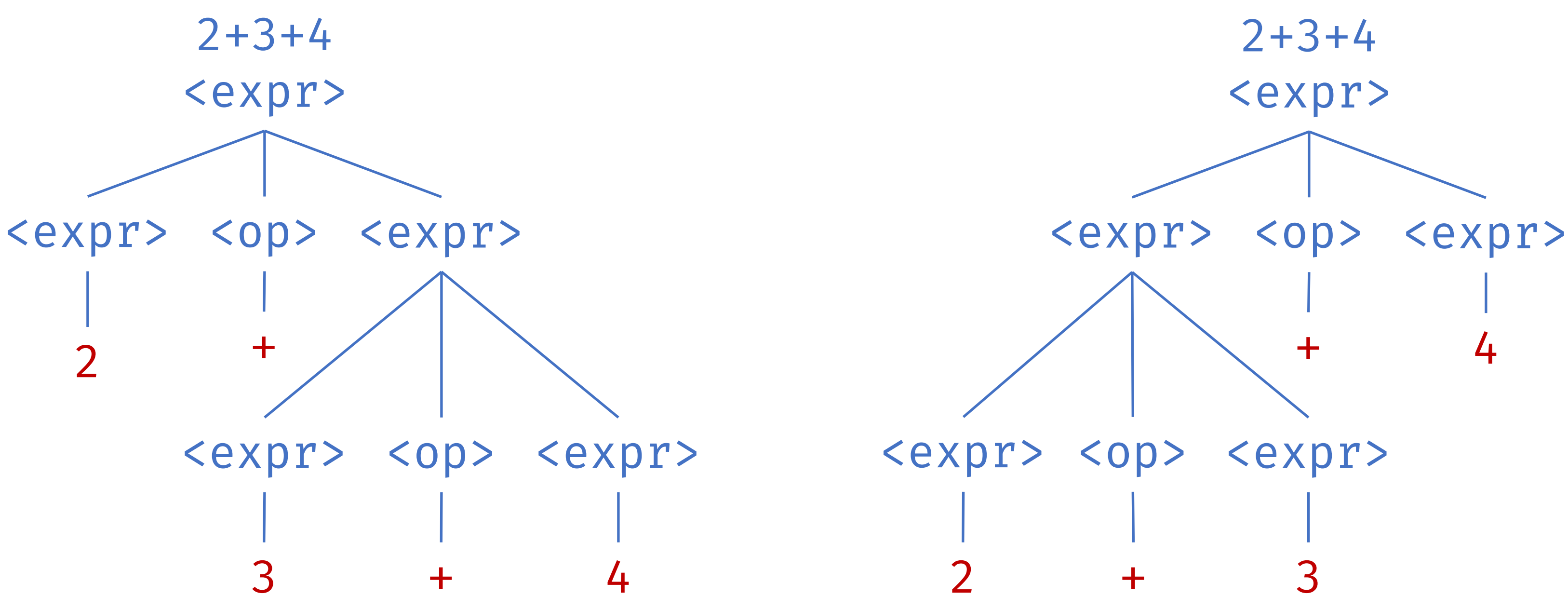
Ambiguity can have different sources.

Good news: we can usually eliminate the ambiguity by revising the grammar.

Ambiguous Grammars – another example

The previous example does not depend on the use of two operations, we have a similar ambiguity with one operation.

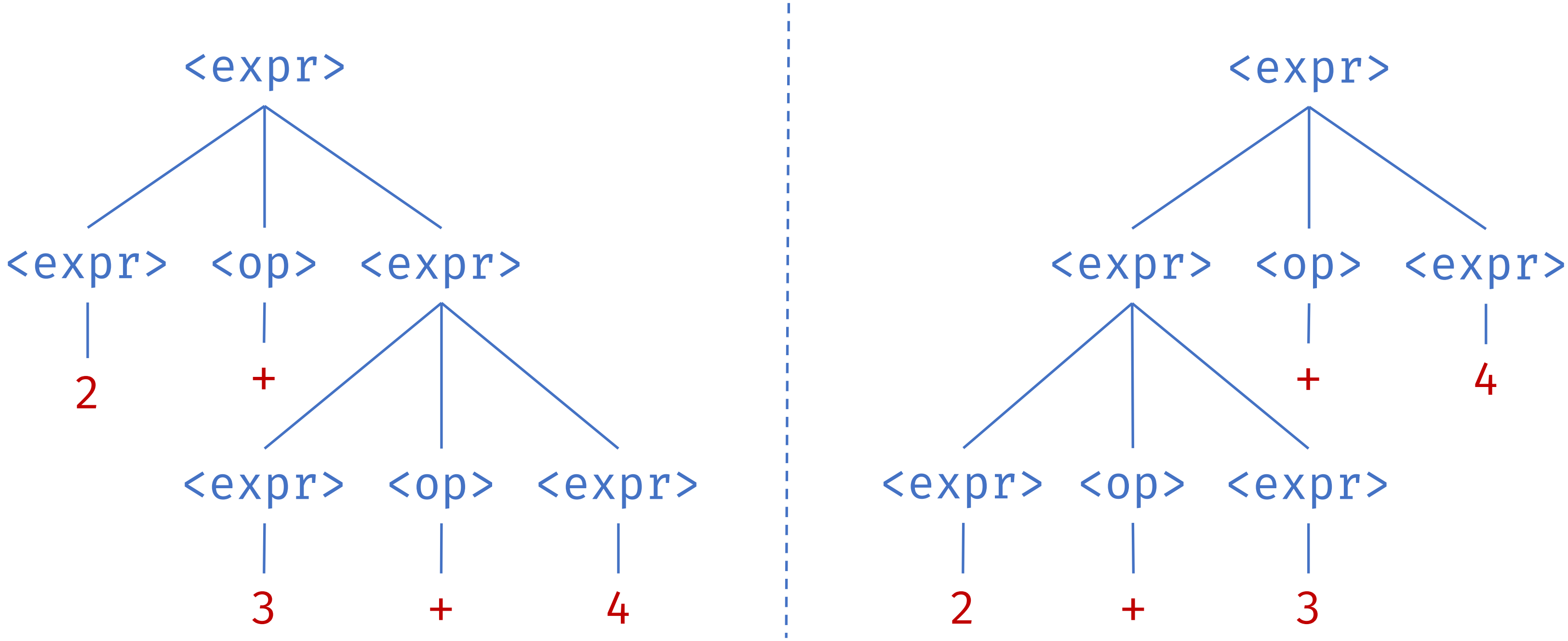
```
<expr> ::= <expr> <op> <expr>
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```



How can we avoid ambiguity?

How can we disambiguate between the two parse trees for the following expression?

2+3+4



How can we avoid ambiguity?

How can we disambiguate between the two parse trees for the following expression?

2+3+4

First idea: make the parentheses part of the language

((2 + 3) + 4) (2 + (3 + 4))

One way to do this is to change the grammar:

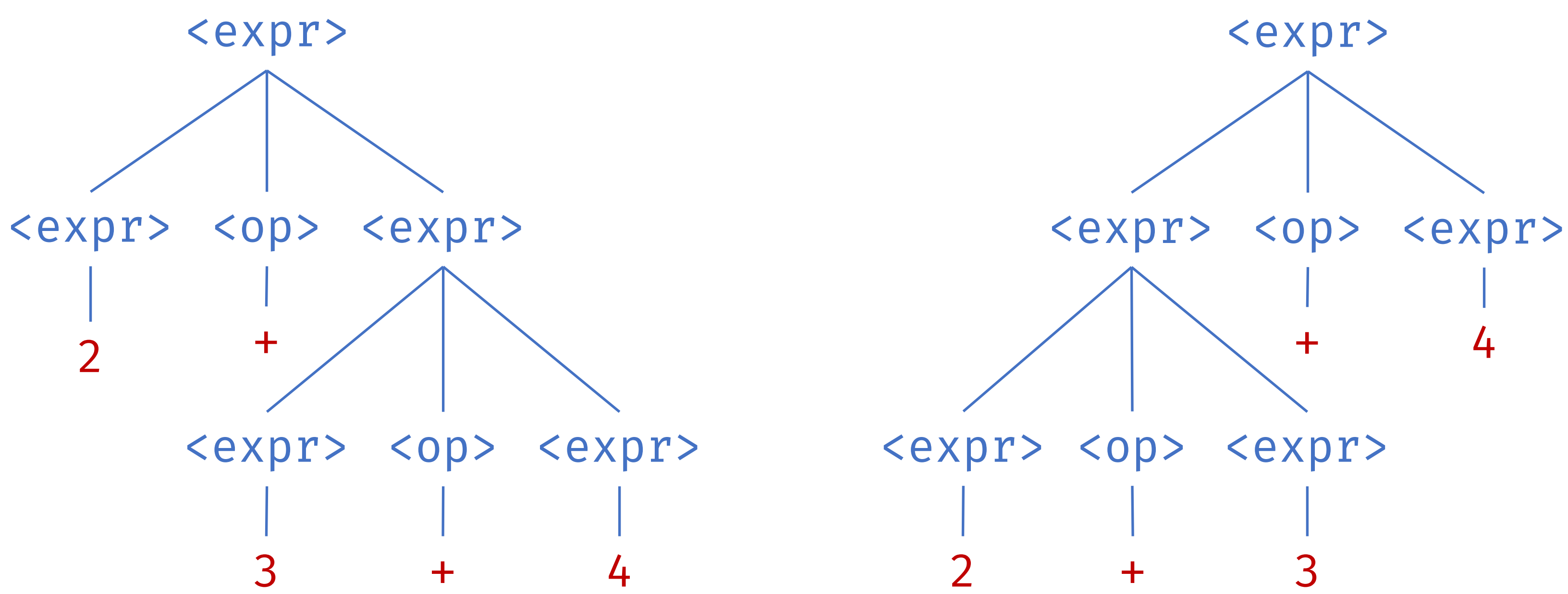
```
<expr> ::= (<expr> <op> <expr>)
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +|-|*|/
```

We need to add them everywhere!

We add parentheses around every expression

How can we avoid ambiguity and preserve the structure of the grammar?

Second idea: If we use the parse tree to indicate precedence levels of operators, we cannot have ambiguity.



Problem: it requires us to modify the grammar

How can we avoid ambiguity and preserve the structure of the grammar?

Why is the previous grammar ambiguous?

$$2 + 3 * 4$$

Two "classes" of operations that have different precedence and the grammar does not distinguish them.

$$2 + 3 + 4$$

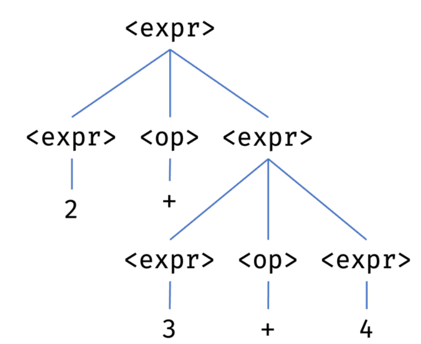
Two "occurrences" of the same operations have the same precedence and the grammar does not distinguish them.

Dealing with associativity?

2 + 3 + 4

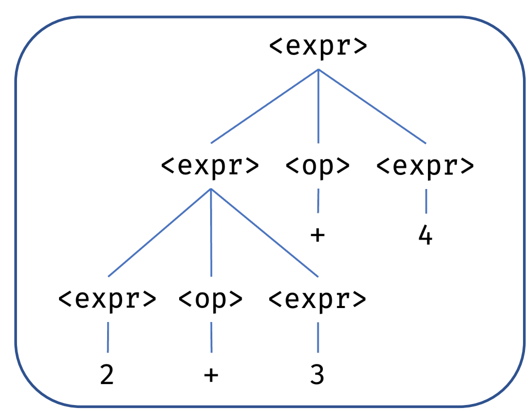
Two "occurrence" of the same operations have the same precedence and the grammar does not distinguish them.

```
<expr> ::= <expr> <op> <expr>
<expr> ::= 1|2|3|4|5|6|7|8|9|0
<op>   ::= +
```



We need to break the symmetry and commit to one choice.

```
<expr>   ::= <const><op><const>
           | <expr><op><const>
<const>  ::= 1|2|3|4|5|6|7|8|9|0
<op>     ::= +
```



Dealing with associativity?

```
<expr>   ::= <const><op><const>
           | <expr><op><const>
<const> ::= 1|2|3|4|5|6|7|8|9|0
<op>     ::= +
```

We use two nonterminal to break the symmetry

How can we derive the following expression?

$$2 + 3 + 4 + 5$$

```
<expr> => <expr> <op> <const>
       => <expr> <op> <const> <op> <const>
       => <const> <op> <const> <op> <const> <op> <const>
       => 2 <op> <const> <op> <const> <op> <const>
       => 2 + <const> <op> <const> <op> <const>
       => 2 + 3 <op> <const> <op> <const>
       => 2 + 3 + <const> <op> <const>
       => 2 + 3 + 4 <op> <const>
       => 2 + 3 + 4 + <const>
       => 2 + 3 + 4 + 5
```

(THIS PAGE INTENTIONALLY LEFT BLANK)