Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Nathan Mull**
for teaching CS320: Concepts of Programming Languages
at Boston University [1].
Content in this document is based on content provided by Mull.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisite Definitions

This text assumes that the reader has a basic understanding of programming languages and grade-school mathematics along with a fundamentals grasp of discrete mathematics. The following definitions are provided to ensure that the reader is familiar with the terminology used in this document.

Definition 0.1: Token

A **token** is a basic, indivisible unit of a programming language or formal grammar, representing a meaningful sequence of characters. Tokens are the smallest building blocks of syntax and are typically generated during the lexical analysis phase of a compiler or interpreter.

Examples of tokens include:

- `keywords`, such as `if`, `else`, and `while`.
- `identifiers`, such as `x`, `y`, and `myFunction`.
- `literals`, such as `42` or `"hello"`.
- `operators`, such as `+`, `-`, and `=`.
- `punctuation`, such as `(`, `)`, `{`, and `}`.

Tokens are distinct from characters, as they group characters into meaningful units based on the language's syntax.

Definition 0.2: Non-terminal and Terminal Symbols

Non-terminal symbols are placeholders used to represent abstract categories or structures in a language. They are expanded or replaced by other symbols (either terminal or non-terminal) as part of generating valid sentences in the language.

- **E.g.**, "Today is ⟨name⟩'s birthday!!!", where ⟨name⟩ is a non-terminal symbol, expected to be replaced by a terminal symbol (e.g., "Alice").

Terminal symbols are the basic, indivisible symbols in a formal grammar. They represent the actual characters or tokens that appear in the language and cannot be expanded further. For example:

- `+`, `1`, and `x` are terminal symbols in an arithmetic grammar.

Definition 0.3: Symbol ":="

The symbol := is used in programming and mathematics to denote "assignment" or "is assigned the value of". It represents the operation of giving a value to a variable or symbol.

For example:

$$x := 5$$

This means the variable x is assigned the value 5.

In some contexts, := is also used to indicate that a symbol is being defined, such as:

$$f(x) := x^2 + 1$$

This means the function $f(x)$ is defined as $x^2 + 1$.

Definition 0.4: Substitution: $[v/x]e$

Formally, $[v/x]e$ denotes the substitution of v for x in the expression e. For example:

$$[3/x](x + x) = 3 + 3$$

This means that every occurrence of x in e is replaced with v. We may string multiple substitutions together, such as:

$$[3/x][4/y](x + y) = 3 + 4$$

Where x is replaced with 3 and y is replaced with 4.

— 1 —

The Interpretation Pipeline

1.1 Formal Grammars

1.1.1 Defining a Language

Now we introduce formal grammars as a way of building up our language. This is similar to English, where we have a grammar system that tells us how to build sentences. For example, we know the basic structure of a sentence is *subject-verb-object*.

We can write **linear** statements such as "John hit the ball", which has an underlying **hierarchical** structure that permits it:

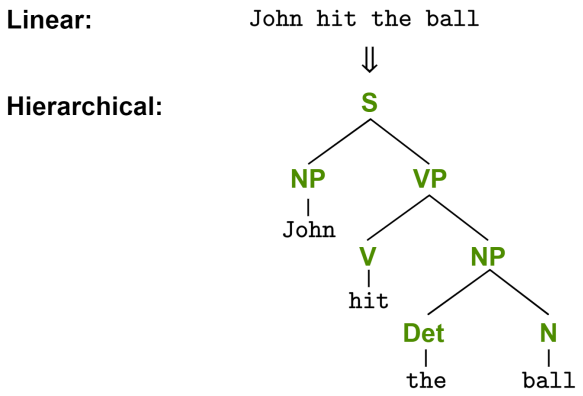

Figure 1.1: The sentence "John hit the ball" has an underlying hierarchical structure of a tree. Here, **S**: Sentence (the root of the tree), **NP**: Noun Phrase (a phrase centered around a noun), **VP**: Verb Phrase (a phrase centered around a verb), **V**: Verb (the action in the sentence), **Det**: Determiner (words like "the", "a", "an", which specify nouns), and **N**: Noun (person, place, thing, or idea).

Grammar vs. Semantics: Notice the english sentence

"Your air tied a toothbrush at school!"

is grammatically correct, but carries little to no meaning. In contrast, the sentence

"Colorless the of allegator run am sleepily"

is perhaps an unsettling read, as it is not grammatically correct.

The same way we can represent english sentences in a tree structure, is the same way we can represent programs. First we define the difference between an **interpreter** and a **compiler**.

Definition 1.1: Interpreter

An **interpreter** is a program that directly executes instructions written in a programming language without requiring a machine code translation. The typical stages are:

1. **Lexical Analysis**: Reads a string of characters (program), converting it into tokens.
2. **Syntax Analysis**: Parses these tokens to build an abstract syntax tree (AST).
3. **Semantic Analysis**: Checks for semantic errors and annotates the AST.
4. **Intermediate Representation (IR) Generation**: Converts the AST into an intermediate representation (IR) to facilitate execution.
5. **Direct Execution**: Executes the IR or AST directly using an interpreter.

Interpreted languages are **evaluated at runtime** (e.g., Python, Ruby, JavaScript). Some interpreters use an **AST-based execution**, while others generate an **IR** (e.g., Python's bytecode for the CPython interpreter).

Definition 1.2: Compiler

A **compiler** is a program that translates code written in a high-level programming language into a lower-level language, typically machine code, to create an executable program. This involves several stages:

1. **Lexical Analysis**: Reads the source code and converts it into tokens.
2. **Syntax Analysis**: Parses these tokens to construct an abstract syntax tree (AST).
3. **Semantic Analysis**: Validates the AST against language rules and performs type checking.
4. **Intermediate Representation (IR) Generation**: Transforms the AST into a lower-level representation that is easier to optimize and translate.
5. **Optimization**: Enhances the IR to improve performance and efficiency.
6. **Code Generation**: Translates the optimized IR into machine code or another target language.

Compiled languages are **translated before runtime** (e.g., C, C++, Rust, OCaml). The **IR** plays a crucial role in optimizing the compilation process, as seen in LLVM or Java's bytecode execution in the JVM.

The following diagram illustrate the translation process of a compiler and an interpreter:

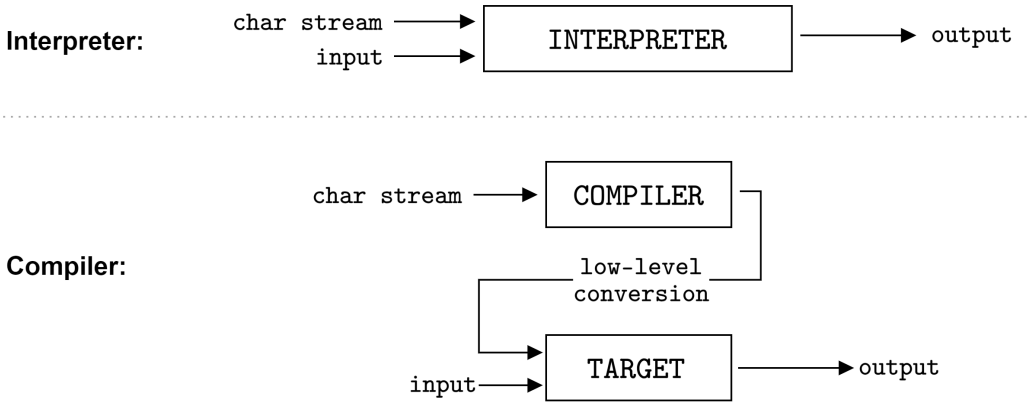

Figure 1.2: The high-level processes of a compiler and an interpreter.

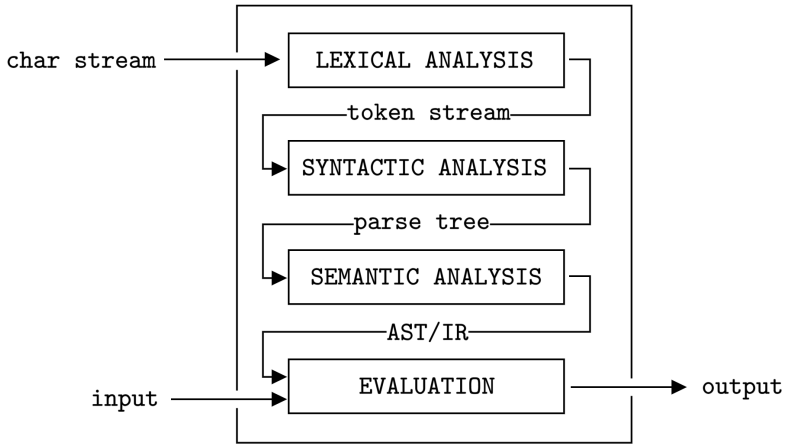

Figure 1.3: Stages of program processing: A character stream undergoes lexical, syntactic, and semantic analysis, transforming into an AST or IR before evaluation. Interpreters execute the AST/IR directly, while compilers translate it into machine code.

To formally layout our language, we use the **Backus-Naur Form (BNF)** notation. But before we do so, we gain some intuition by breaking down an english sentence from its **terminal symbols** to its **non-terminal symbols**. Recall these terms from the pre-requisite section (0.2).

```
the cow jumped over the moon
<article> cow jumped over the moon
<article> <noun> jumped over the moon
<noun-phrase> jumped over the moon
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> <verb-phrase>
<sentence>
```

Figure 1.4: The sentence "The cow jumped over the moon." broken down into terminal and non-terminal symbols. This is a derivation showing how the sentence is built up from the start symbol of a `<sentence>`.

From which the above can be represented in a tree structure:

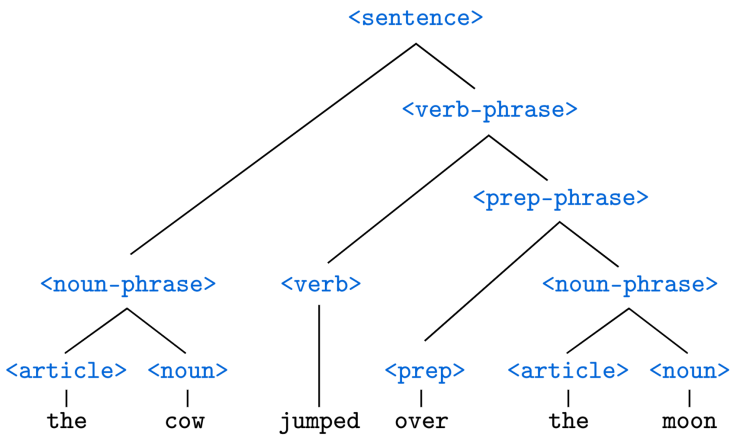

Figure 1.5: The sentence "The cow jumped over the moon." represented as a **parse-tree**.

Now if we wanted to state these rules before-hand that a `<sentence>` is made up of a `<noun-phrase>` and a `<verb-phrase>` we can do so as such:

```
<sentence>     ::= <noun-phrase> <verb-phrase>
<verb-phrase>  ::= <verb> <prep-phrase>  | <verb>
<prep-phrase>  ::= <prep> <noun-phrase>
<noun-phrase> ::= <article> <noun>
<article> ::= the
<noun>  ::= cow  | moon
<verb>   ::= jumped
<prep>   ::= over
```

Figure 1.6: The rules for the sentence "The cow jumped over the moon." via a thread of production rules (**??**). This example illustrates **Backus-Naur Form (BNF)** notation.

Definition 1.3: Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a formal notation used to define the context. It consists of production rules that specify how symbols in a language can be recursively composed. Each rule follows the form:

```
<non-terminal> ::= expression
```

where `<non-terminal>` represents a syntactic category, and `expression` consists of terminals, non-terminals, or alternative sequences.

Consider a more programmer-like example, assuming we read from left-to-right:

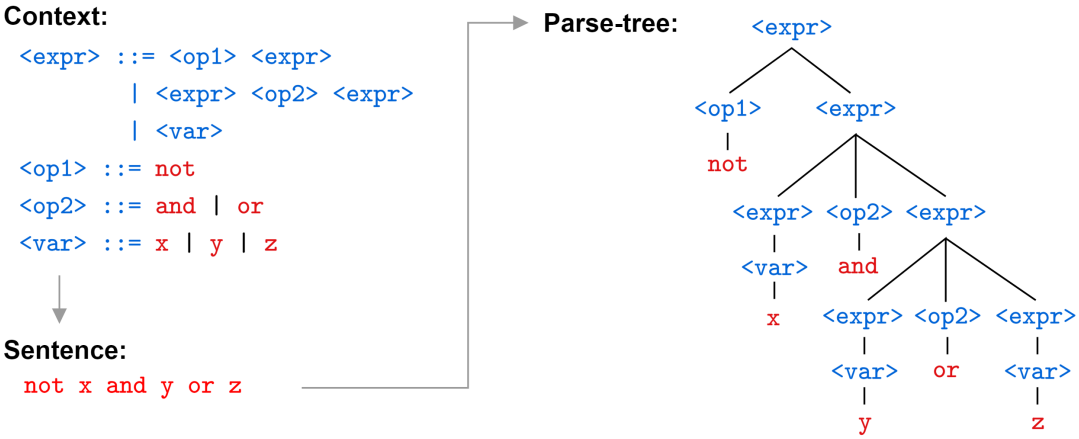

Figure 1.7: A simple language, consisting of `and` and `or` operations.

The process of taking non-terminal symbols and replacing them with terminal symbols is called a **leftmost derivation**.

Definition 1.4: Leftmost Derivation

A **leftmost derivation** is a sequence of sentential forms in which, at each step, the **leftmost** non-terminal symbol is replaced according to a production rule. This process continues until the entire string consists only of terminal symbols. Leftmost derivations are useful for defining deterministic parsing strategies and constructing parse trees in a structured manner.

1.1.2 Ambiguity in Grammars

In natural language we have cases where what we say can be ambiguous. For example,

"The duck is ready for the dinner table."

Natural questions may arise:

- Was the duck ready to be eaten?
- Was the duck preparing the table for dinner?
- Was the duck a wrestler, ready to body-slam a constituent?

For instance, let's clean up the previous sentence, avoiding any and all ambiguity:

"The duck is ready ***to body slam*** *the dinner table."*

Now Consider the previous example of `and` and `or` operations from Figure (1.7):

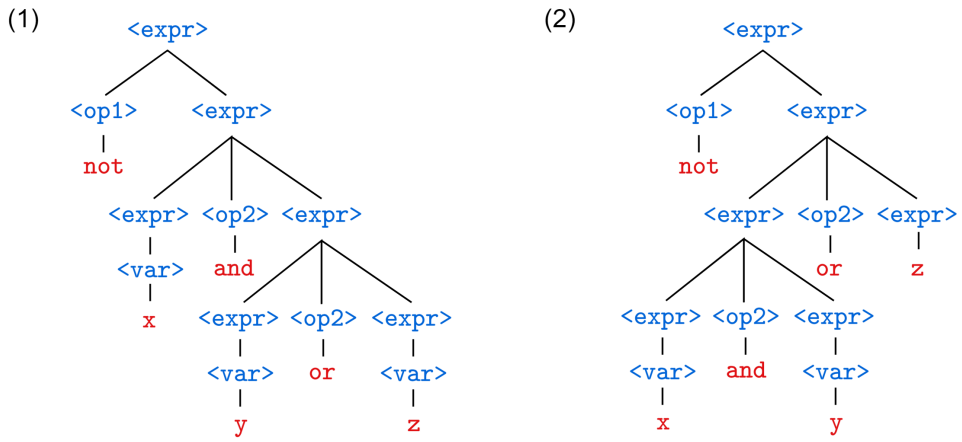

Figure 1.8: Two possible parse-tree derivations based off of Figure (1.7). (1) Shows our previous intention, while (2) shows a different interpretation, if parsing order was not specified.

In Figure (1.8), (1) shows, `not x and (y or z)` while (2) shows, `not (x and y) or z`. This ambiguity can cause issues; for example, $x = y = False$, $z = True$, yields different results for (1) and (2).

Definition 1.5: Ambiguity in BNF Grammar

A **BNF grammar** is **ambiguous** if there exists a sentence with multiple valid parse-trees.

These, small pieces of ambiguity can crop up anywhere. Consider the following example:

```
<expr> ::= x
         | if <expr> then <expr>
         | if <expr> then <expr> else <expr>
...
```

Figure 1.9: An ambiguous grammar `if` expressions.

In Figure (1.9), contains sub-cases which can lead to ambiguity. As, "is it the sub-case or the super-case?" For example, consider:

`if x then (if y then z else w)` or is it `if x then (if y then z) else w`

These problems often arise when interpreting expressions at the linear level. On an important note:

Theorem 1.1: Ambiguity is Undecidable

It is impossible to write a generalized program to determine if a given grammar set is ambiguous. This requires to finding all possible sentences; **However**—our grammar is recursive—this would mean generating an infinite number of sentences.

Though we can avoid ambiguity by specifying our operation order:

Definition 1.6: Fixity

The **fixity** of an operator refers to its placement relative to its operands:

- **Prefix**: The operator appears *before* its operand (e.g., $f\ x$, $-x$).
- **Postfix**: The operator appears *after* its operand (e.g., $a!$ for factorial or dereferencing).
- **Infix**: The operator appears *between* two operands (e.g., $a * b$, $a + b$, $a \mod b$).
- **Mixfix**: The operator is interleaved with its operands, appearing in multiple positions (e.g., `if` b `then` x `else` y).

So by specifying only Prefix or only Postfix operations ambiguity can be avoided:

Definition 1.7: Polish Notation

Polish Notation is a notation for arithmetic expressions in which the operator precedes its operands. For example, the polish notation $-/+2*1-23$ is written as $-(2+(1*(-2)/3)$.

In contrast, **Reverse Polish Notation** (RPN) places the operator after its operands. For example, Infix: $(3+4)*5$ is $3\ 4+5\ *$ in RPN.

However, this is not always practical. Consider `if` expressions dealt this way:

```
<expr> ::= <bool>
         | <var>
         | ifthen <expr> <expr>
         | ifthenelse <expr> <expr> <expr>

<bool> ::= tru | fls
<var> ::= x | y | z
```

Figure 1.10: An unambiguous grammar for `if` expressions.

Likewise, if we decided to enforce parenthesis everywhere it might be a bit cumbersome:

```
<expr> ::= ( <op1> <expr> )
         | ( <expr> <op2> <expr> )
         | <var>

<op1> ::= not
<op2> ::= and | or
<var> ::= x | y | z
```

$\Rightarrow$

```
(((not x) and (not (not y))) or z)

((((x or y) or z) or x) or y)

(not ((not x) and (not y)) or (x and z))

(x and y)
```

Figure 1.11: An unambiguous grammar for `and or not` expressions, with parenthesis everywhere.

Theorem 1.2: The Ambiguity of Associativity & Precidence Rules

- **Associativity**: $a+b+c$ is ambiguous, as it could be $(a+b)+c$ or $a+(b+c)$.
- **Precedence**: $a+b*c+d$ is ambiguous, as it could be $(a+(b*c))+d$ or $a+((b*c)+d)$.

We can resolve this by breaking the symmetry of our grammar:

Theorem 1.3: Resolving Associativity: Breaking Symmetry

The reason for ambiguity often comes symmetry in grammar rules of form:

```
<expr> ::= <expr> <op> <expr>
  <op> ::= + | - | ...
```

This means that the left or right side can be expanded indefinitely. By breaking this symmetry, we can resolve ambiguity:

```
<expr> ::= <var> <op> <expr>
  <op> ::= + | - | ...
 <var> ::= x | y | z | ...
```

In the above we fixed the left side to be a `<var>` and the right side to be an `<expr>`. This makes our expression **right-associative**. Vice-versa, if we fixed the right side, it would be **left-associative**.

Theorem 1.4: Resolving Precedence: Factor Out Higher Precedence

Precedence issues arise when operations reside on the same level. For instance,

```
<expr>::= <expr> + <expr> | <expr> * <expr>
```

Here, it's unclear whether `+` or `-` should be evaluated first. By factoring out higher precedence operations, we can resolve this, as, **terms deeper in the parse tree are evaluated first**:

```
<expr>::= <expr> + <term> | <term>
<term>::= <term> * <var> | <var>
 <var>::= x | y | z | ...
```

Here, `*` has higher precedence than `+`, as it's deeper in the parse tree. To handle **parentheses**, we can introduce another factor:

```
<expr>::= <expr> + <term> | <term>
<term>::= <term> * <var> | <pars>
<pars>::= var | ( <expr> )
 <var>::= x | y | z | ...
```

Intuitively, in expressions like `(a + b) * c`, we want to treat `(a + b)` as a single unit/*variable*. **Note**, the parentheses are tokens surrounding the non-terminal `<expr>`.

Now consider the `and or not` expressions from Figure (1.7) resolving the associative ambiguity:

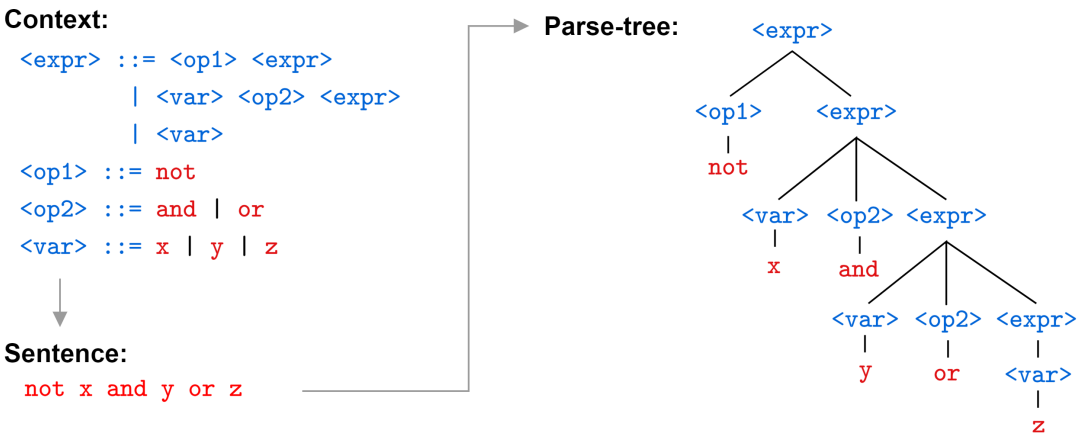

Figure 1.12: An unambiguous grammar for `and or not` expressions, by breaking symmetry.

Now we put all the rules together:

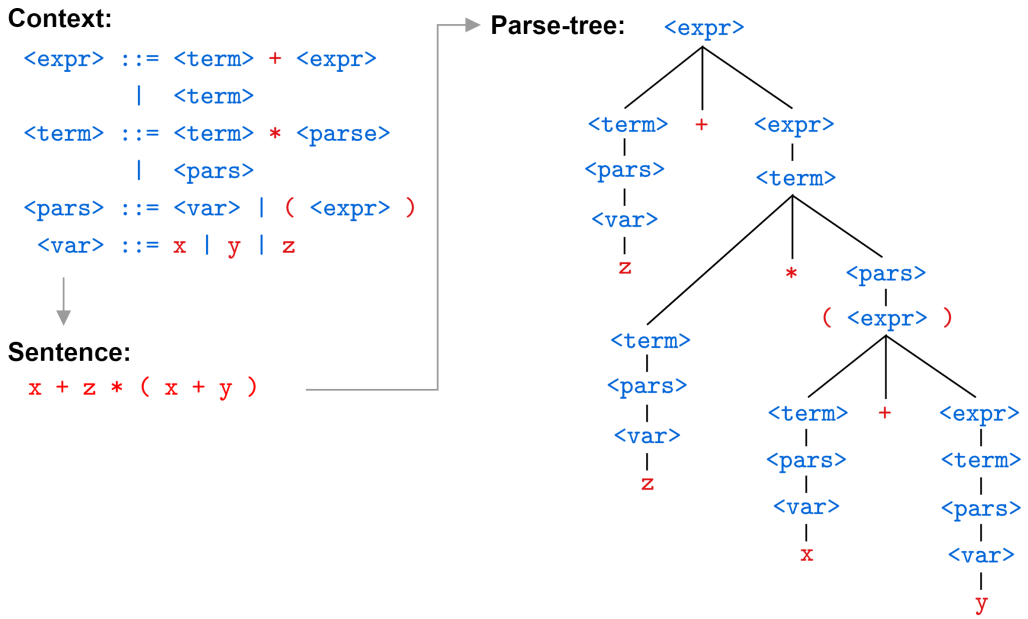

Figure 1.13: An unambiguous grammar for `+, *` expressions, by breaking symmetry and precedence.

Tip: The "Cult of Parentheses" in Lisp: Lisp-style languages rely heavily on **prefix notation**, where operators and function names appear **before** their arguments. This leads to deeply nested **parentheses**, often making Lisp code look overwhelming to newcomers.

For example, the Fibonacci function in Lisp is written as:

```
(defun fib (n)
  "Return the nth Fibonacci number."
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Since **everything** in Lisp is an expression and follows a **uniform syntax**, parentheses are required for every function call, condition, and operation. This often leads to **excessive nesting**, and the title, **"Cult of Parentheses"**.

Bibliography

[1] Nathan Mull. Cs320: Concepts of programming languages. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.