PARSER

Prepared by Rene Andre B. Jocsing September 8, 2025

Welcome to the second leg of your language hacking journey. Now that your scanner can transform source code into tokens, it's time to build the next component: the **parser**. Think of parsing as the step where we take those tokens and figure out what they actually *mean* in the context of your programming language.

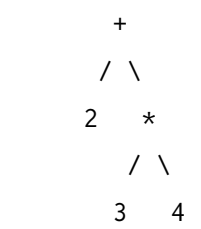
If the scanner answers, "what are the words?", then the parser answers, "how do the words fit together?" To do this, a parser takes the array of tokens from your scanner and organizes them into a tree that represents the grammatical structure of your source code.

From Token Arrays to Expression Trees

Consider this simple mathematical expression:

```
1 2 + 3 * 4
```

Your scanner may break this up into tokens: `NUMBER(2)`, `PLUS`, `NUMBER(3)`, `STAR`, `NUMBER(4)`. But how do you parse these tokens? Note that a linear sequence doesn't capture the crucial fact that, in math, multiplication has higher precedence than addition. With just an array of tokens, you could make the mistake of evaluating 2 plus 3 first (you probably even did this once in your math series!). Hence, it is the parser's job is to organize these tokens into a tree that reflects the correct mathematical meaning. The parser should produce something like this.



This tree structure, called an **Abstract Syntax Tree (AST)**, makes the precedence explicit. We go from the bottom-most, left-most leaves up to the root. In this example, we must evaluate the multiplication first (3 * 4), then add 2 to the result.

Context-Free Grammars

Before we can build a parser, we first need to formalize the rules that govern how tokens can be combined. We do this by using a **Context-Free Grammar (CFG)**. Don't let the fancy name intimidate you. In mathematics and, by extension, computer science, it's just a systematic way of describing the syntax rules of some language.

A CFG consists of:

- **Terminals**: The actual tokens from your scanner (like `NUMBER`, `PLUS`, `STAR`) (denoted by quotes)
- **Non-terminals**: Abstract categories that represent groups of tokens (like "expression" or "term", denoted by the absence of quotes)
- **Production rules**: Rules that define how non-terminals can be broken down into sequences of terminals and other non-terminals

Here's a simple grammar for arithmetic expressions, defined by production rules that yield both terminals and non-terminals.

```
expression → equality
equality   → comparison ( ( "!=" | "==" ) comparison )*
comparison → term ( ( ">" | ">=" | "<" | "<=" ) term )*
term       → factor ( ( "-" | "+" ) factor )*
factor     → unary ( ( "/" | "*" ) unary )*
unary      → ( "!" | "-" ) unary | primary
primary    → NUMBER | STRING | "true" | "false" | "nil" | "(" expression ")"
```

The fact that recursion is convenient allows us to form a combinatorically large number of strings from a limited grammar.

Each line is a production rule and yields a valid string in your programming language (a line of your source code). The symbol before the arrow is a non-terminal, and the stuff after the arrow (read: "produces") shows what it can be replaced with, separated by a pipe "|" (read: "or"). You can choose whichever terminal or non-terminal among the grouped, pipe-separated choices when you evaluate a production rule. So, if you were doing this by hand, you'd replace `expression` with `equality` in a statement, and keep on replacing non-terminals (starting from the left-most) until the string becomes composed entirely of terminals. The `*` means "zero or more", `+` means "one or more", and `?` means "zero or one" (optional). Parentheses group choices together. Note that a parenthesis in quotes is interpreted as the parenthesis *character*!

Let's trace through parsing `2 + 3` by hand (get a piece of paper so you can follow).

1. Start with `expression`
2. Replace with `equality` (only choice)
3. Replace with `comparison` (no equality operators present)
4. Replace with `term` (no comparison operators present)
5. Replace with `factor ( ( "-" | "+" ) factor )*`
6. The first `factor` becomes `unary`, then `primary`, then `NUMBER(2)`

7. We have a `+`, so we match the `*` part once
8. The second `factor` becomes `NUMBER(3)`
9. Final result: a `term` containing `NUMBER(2) + NUMBER(3)`

Note, however, that a grammar can be ambiguous. Consider this simpler grammar.

```
expression → expression "+" expression | NUMBER
```

For the input `1 + 2 + 3`, this grammar allows two different parse trees. Can you trace why?

```
    +           +
   / \    OR   / \
  1   +       +   3
     / \     / \
    2   3   1   2
```

The left tree represents `1 + (2 + 3)` while the right represents `(1 + 2) + 3`. This ambiguity makes the grammar unsuitable for a parser (because what if instead of plus it was multiply?). To solve ambiguity, we must structure our grammar carefully to enforce **precedence** and **associativity**.

Precedence and Associativity

Notice how the first grammar in the previous section is structured from lowest to highest precedence. Equality operations (like ==) have lower precedence than comparisons (like >), which have lower precedence than addition and subtraction. Addition and subtraction then have lower precedence than multiplication and division. Unary operations like "!" and "–" have the second highest precedence. Primary has the highest precedence among all because these are your literals like numbers, strings, booleans, etc. or specially enforced groupings via parenthesis.

The grammar's hierarchy ensures that `2 + 3 * 4` parses as `2 + (3 * 4)` rather than `(2 + 3) * 4`. It accomplishes this by separating each categorical operation into their own production rule (think separation of concerns from CMSC 22!) and arranging them according to precedence (think which symbol you'd read first if you were parsing it yourself).

The grammar also handles associativity. Most binary operators are left-associative, meaning `1 - 2 - 3` should parse as `(1 - 2) - 3`. The way we structure the recursive rules in our grammar naturally creates left-associative parsing (but you can try to structure right-associative parsing, if you wish).

Recursive Descent Parsing

At the heart of this lab, you'll implement what's called a **recursive descent parser**. This is a top-down parsing technique where each grammar rule becomes a method in your parser class.

The beauty of this approach is that your code mirrors your grammar almost exactly. Because of this, recursive descent parsers are the most beginner-friendly among parsing techniques. But don't underestimate it! Recursive descent parsers are used in production for gcc, JavaScript engines, and C# parsers because they're fast, robust, and easy to implement. If C++ can be parsed using a recursive descent parser, it should work for your programming language, too.

To parse a stream of tokens, we must transform our grammar rules into code. Consider that, in a recursive descent parser, the `term` rule from before,

```
term       → factor ( ( "-" | "+" ) factor )*
```

becomes a `term` method that:

1. Calls the `factor` method to parse the left operand
2. While the current token is `+` or `-`:
    - Saves the operator
    - Calls `factor` again to parse the right operand
    - Creates a binary expression node with left operand, operator, and right operand
    - Makes this new node the left operand for the next iteration
3. Returns the final expression

Translating Grammar Rules to Code

Each production rule in your grammar translates directly to a method in your parser. Here are some common patterns that you can adopt.

Simple sequence (`unary → ( "!" | "-" ) unary | primary`):

```
function unary():
    if current token is "!" or "-":
        save the operator
        recursively call unary() for operand
        return UnaryNode(operator, operand)
    else:
        return primary()
```

Repetition with `*` (`term → factor ( ( "-" | "+" ) factor )*`):

```
function term():
    expr = factor()

    while current token is "-" or "+":
        save the operator
        right = factor()
```

```
        expr = BinaryNode(expr, operator, right)

    return expr
```

Repetition with + (`arguments → expression ( "," expression )+`):

```
function arguments():
    args = []
    args.add(expression())  // First one required

    while current token is ",":
        consume ","
        args.add(expression())

    return args
```

Optional with ? (`function → "fun" IDENTIFIER "(" parameters? ")" block`):

```
function parseFunction():
    expect "fun"
    name = expect IDENTIFIER
    expect "("

    if current token is not ")":
        params = parameters()  // Optional - only if present
    else:
        params = empty list

    expect ")"
    body = block()
    return FunctionNode(name, params, body)
```

Notice that `*` becomes `while`, `+` becomes "do once, then while", and `?` becomes `if`.

The Abstract Syntax Tree

Your parser's output will be an Abstract Syntax Tree—a tree data structure where each node represents a construct in your programming language. You'll need classes for different types of expressions. For example:

- **Literal**: Represents literal values like numbers, strings, booleans
- **Unary**: Represents unary expressions like `-x` or `!flag`
- **Binary**: Represents binary expressions like `a + b` or `x == y`
- **Grouping**: Represents parenthesized expressions like `(2 + 3)`

Error Handling

Real-world code often has syntax errors, so your parser needs to handle them gracefully. When your parser encounters an unexpected token, it should:

1. Report a clear error message indicating what was expected
2. Avoid crashing the entire parsing process
3. Continue parsing to find additional errors (rather than stopping at the first one)

For now, focus on basic error detection and reporting. We'll tackle more sophisticated error recovery techniques in the interpreter labs.

Laboratory Deliverables

Context-Free Grammar

Draft a CFG for your programming language. To do this, **add a "Grammar" section to your README.md in your GitHub repository**. The CFG must be **unambiguous** and must capture the grammar rules of your programming language accurately.

Parser

Building upon your previous lab, implement a complete recursive descent parser that can parse a stream of tokens according to the grammar of your language. Your parser should:

- Take the list of tokens from your scanner as input
- Build an Abstract Syntax Tree representing the parsed expression
- Handle operator precedence correctly
- Support grouping with parentheses
- Include basic error reporting for malformed expressions (unbalanced parenthesis, etc.)

AST Printer

Create an AST printer that can traverse your syntax tree and output a string representation. Pretty printing will help you visualize and debug your parser. The pretty printer should output expressions in a format like:

```
(+ 1.0 (* 2.0 3.0))
```

This parenthesized representation makes the tree structure explicit and is useful for testing.

Integration

Modify your main program to use both the scanner and parser together:

1. Scanner converts source code to tokens
2. Parser converts tokens to an AST
3. AST printer displays the parsed structure

Your REPL should now show the parsed representation of expressions rather than just the raw tokens. This improved REPL is the expected, testable output for this laboratory.

Testing

Create comprehensive test cases that demonstrate:

- Correct precedence handling
- Left associativity of binary operators
- Proper parsing of literals and grouping
- Error handling for malformed expressions

This time, you'll be providing the test cases and you'll be demonstrating the correctness of your parser during defense.

To close, remember to commit your changes to your GitHub repository with meaningful messages. Continue building on the same repository from your scanner lab. Good luck!

Expected Output

```
> 42
42.0
> "Hello"
Hello
> true
true
> nil
nil
> 1 + 2
(+ 1.0 2.0)
> 3 - 4 * 5
(- 3.0 (* 4.0 5.0))
> --10
(- (- 10.0))
> (1 + 2) == (3 - 0)
(== (group (+ 1.0 2.0)) (group (- 3.0 0.0)))
> (1 + 2) * 3
(* (group (+ 1.0 2.0)) 3.0)
> -(4 + 5)
(- (group (+ 4.0 5.0)))
> 1 + 2 * 3 - 4
(- (+ 1.0 (* 2.0 3.0)) 4.0)
> 10 + -5 * 15 - 6
(- (+ 10.0 (* (- 5.0) 15.0)) 6.0)
> 1 <= 2 != 3 >= 4
(!= (<= 1.0 2.0) (>= 3.0 4.0))
> 5 > 3
(> 5.0 3.0)
> !true
(! true)
> !!false
(! (! false))
> "hello" != "world"
(!= hello world)
> "CMSC 124 is the best"
CMSC 124 is the best
> (1 + 2 + 3 + 4 + 5
[line 1] Error at end: Expect ')' after expression.
> this is nonsense
[line 1] Error at 'this': Expect expression.
> // just a comment, no code
[line 1] Error at end: Expect expression.
```

Rubric for Programming Exercises

Table 1: Rubric for Programming Exercises (50 pts)

Criteria (10 Points Each)	Excellent (9 – 10)	Good (6 – 8)	Fair (3 – 5)	Poor (0 – 2)
Program Correctness	Program executes correctly with no syntax or runtime errors, meets/exceeds specifications, and displays correct output	Program executes and outputs with minor errors, yet meets specifications	Program executes and outputs with major errors, yet somehow meets specifications	Program does not execute or does not meet specs
Logical Design	Program is logically well-designed with excellent structure and flow	Program has slight logic errors that do not significantly affect the results	Program has significant logic errors affecting functionality	Program logic is fundamentally incorrect
Code Mastery	Programmer demonstrates excellent mastery over the program's code	Programmer demonstrates adequate mastery over the program's code	Programmer demonstrates fair mastery over the program's code	Programmer demonstrates poor mastery over the program's code
Engineering Standards	Program is stylistically well designed from an engineering standpoint	Slight inappropriate design choices (i.e., poor variable names, improper indentation)	Severe inappropriate design choices (i.e., code repetition, redundancy)	Program is poorly written
Documentation*	Program is well-documented: comments exist for clarity, not redundancy	Missing one required comment or some redundant comments	Missing two or more required comments or many redundant comments	Most documentation missing or most documentation is redundant

Remember:* *"Code tells you how, comments tell you why."*** *— Jeff Atwood, co-founder of Stack Overflow and Discourse*