Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

Contents **1**

1 The Interpretation Pipeline **4**
1.1 Formal Grammars . . . 4
1.1.1 Defining a Language . . . 4
1.1.2 Ambiguity in Grammars . . . 9
1.2 Formal: Parsing, Lexing, Semantics, & Grammar Types . . . 14
1.2.1 Parsing Generators & Grammar Types . . . 14
1.2.2 Menhir: Simple Lexing & Parsing . . . 19

Bibliography **26**

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.

— 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:

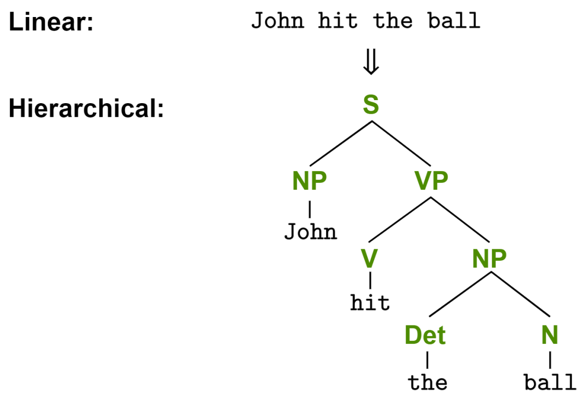


Figure 1.1: The sentence "John hit the ball" has an underlying hierarchical structure of a tree. Here, **S**: Sentence (the root of the tree), **NP**: Noun Phrase (a phrase centered around a noun), **VP**: Verb Phrase (a phrase centered around a verb), **V**: Verb (the action in the sentence), **Det**: 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:

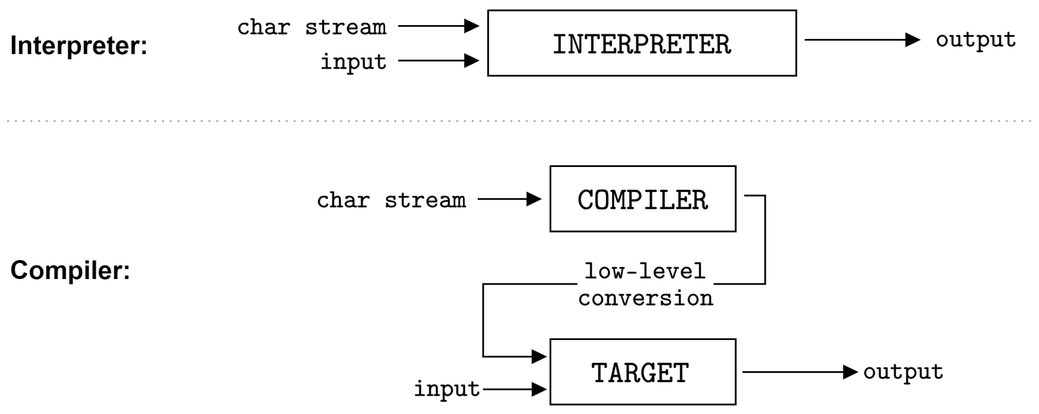


Figure 1.2: The high-level processes of a compiler and an interpreter.

The flow of a program through a compiler or interpreter is as follows:

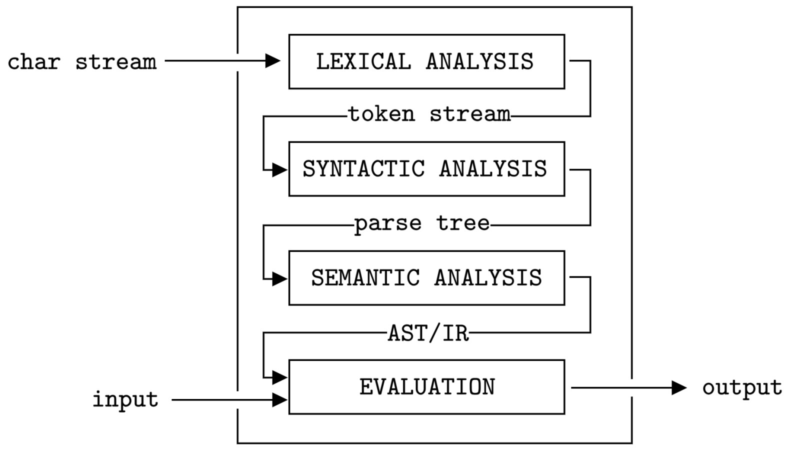


Figure 1.3: Stages of program processing: A character stream undergoes lexical, syntactic, and semantic analysis, transforming into an AST or IR before evaluation. Interpreters execute the AST/IR directly, while compilers translate it into machine code.

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 (**??**).

```
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:

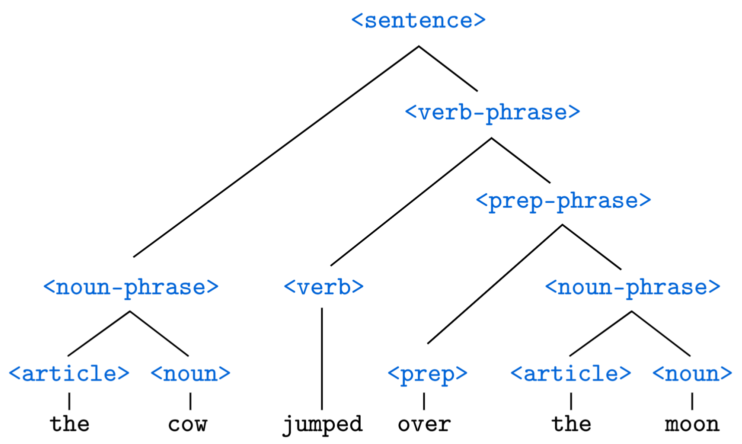


Figure 1.5: The sentence "The cow jumped over the moon," represented as a **parse-tree**.

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.

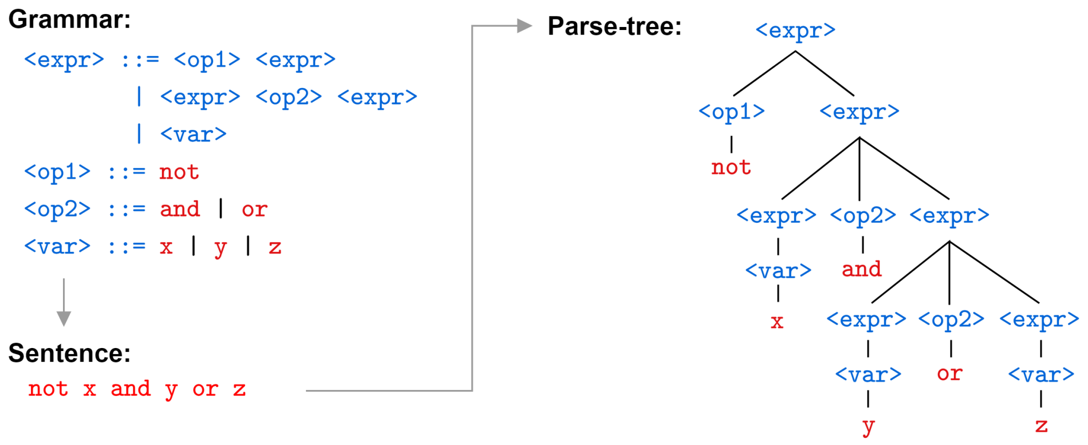
Definition 1.3: Backus-Naur Form (BNF)

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:



Grammar:

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

Sentence:

```
not x and y or z
```

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.

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."

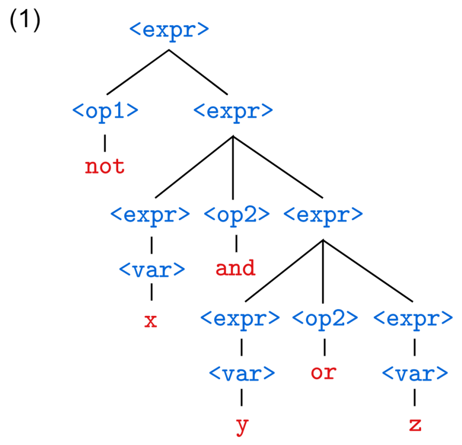
Natural questions may arise:

- Was the duck ready to be eaten?
- Was the duck a servant 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 or not` operations from Figure (1.7):



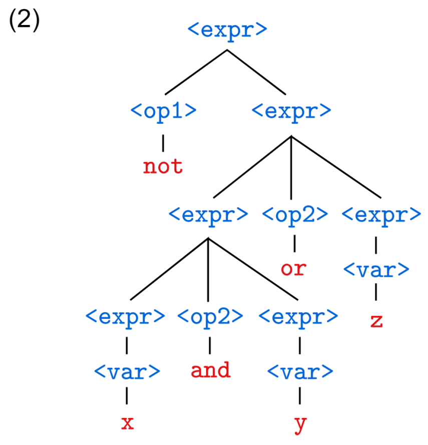


Figure 1.8: Two possible parse-tree derivations based off of Figure (1.7). (1) Shows our previous intention, while (2) shows a different interpretation, if parsing order was not specified.

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)` and `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 to a computer, as it could be $(a+b)*c+d$.

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 Precedences

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**:

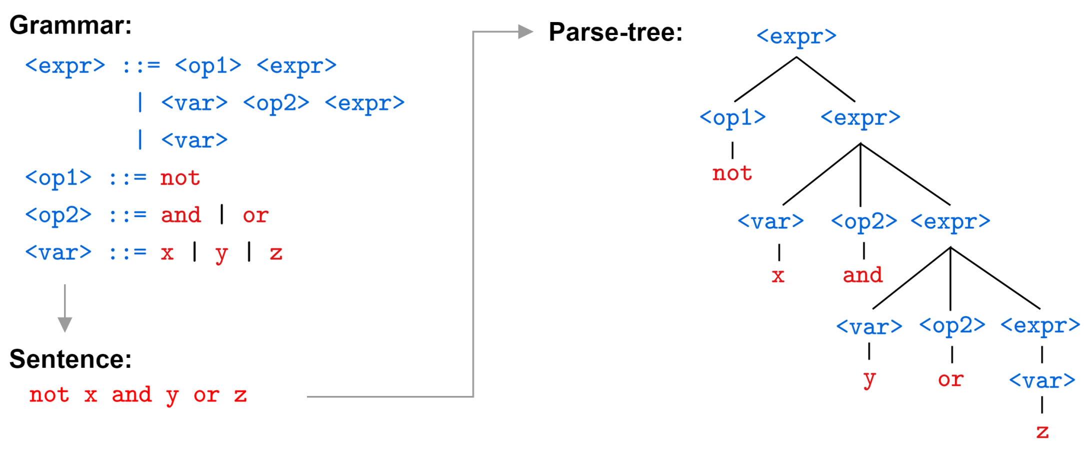
```
<expr>::= <expr> + <term> | <term>
<term>::= <term> * <var> | <var>
 <var>::= x | y | z | ...
```

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> * <pars> | <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:



Grammar:

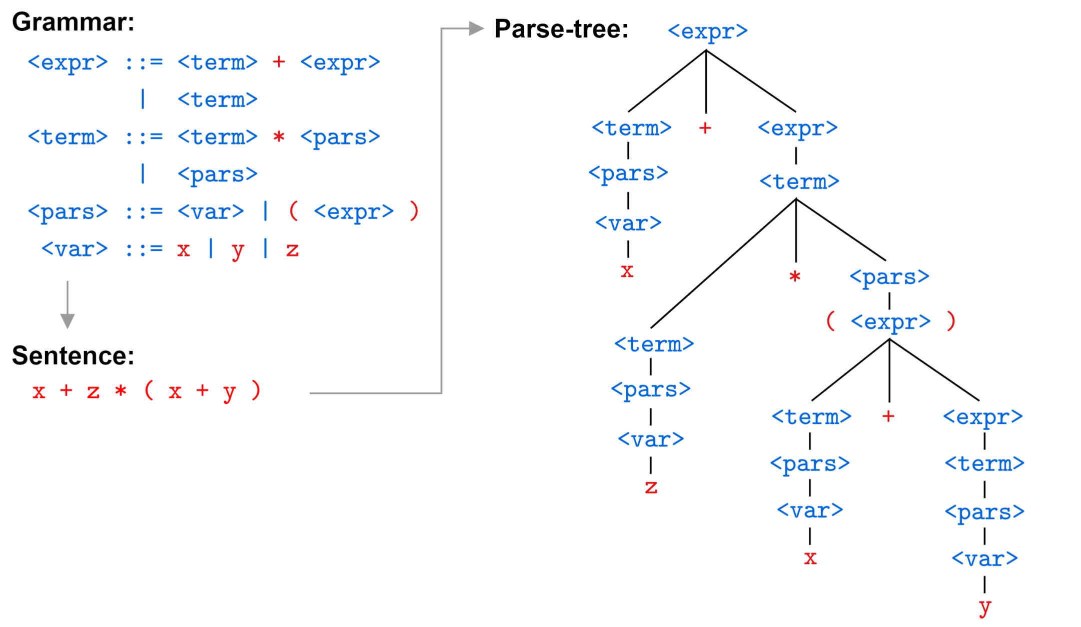
```
<expr> ::= <op1> <expr>
         | <var> <op2> <expr>
         | <var>
<op1> ::= not
<op2> ::= and | or
<var> ::= x | y | z
```

Sentence:

```
not x and y or z
```

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

Now we put all the rules together:



Grammar:

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

Sentence:

```
x + z * ( x + y )
```

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**.

1.2 Formal: Parsing, Lexing, Semantics, & Grammar Types

1.2.1 Parsing Generators & Grammar Types

Lexical analysis and parsing are closely linked to compiler design. Though instead of writing such parsers by hand, we will use **parser generators**:

Definition 2.1: Parser Generators

Parser generators are programs that take a representation of formal grammar as input and produces a parser as output.

We’ll use **Menhir** for OCaml. There are other options but this suffices for our needs.

Definition 2.2: Menhir

Menhir is an LR(1) parser generator for OCaml. LR(1) stands for:

- **L**: Left-to-right scanning of the input.
- **R**: Rightmost derivation in reverse. Meaning, it builds the parse tree from the leaves up to the root.
- **(1)**: One symbol of lookahead. This means that the parser can look at one token ahead in the input stream to make parsing decisions.

Users input small OCaml snippets to define the production rules of the grammar.

A quick aside on Domain Specific Languages (DSLs):

Definition 2.3: Domain Specific Language (DSL)

A domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain. An example of a DSL is SQL, as its only purpose is to query databases. This is in contrast to general-purpose languages like Python, which can be used for a wide range of applications.

DSL's require custom parsers to be written. This can be done by hand, or using parser generators.

Now we discuss **Extended BNF (EBNF)**, which allows us to be more expressive in our grammars with less noise.

Definition 2.4: Extended BNF (EBNF)

Extended Backus-Naur Form (EBNF) is a notation for specifying formal grammars, building upon BNF (Backus-Naur Form) by introducing syntactic extensions for greater clarity and brevity. EBNF includes the following additional constructs:

- `|` — alternation: denotes choice between patterns.
  - **E.g.,** `<letter> ::= A | B | C`, means that <letter> can be A, B, or C.
- `{…}` — repetition: the enclosed pattern may repeat **zero** or more times.
  - **E.g.,** `<digits> ::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}`, match:
    > `"0"`, `"320"`, `"11235813"`
- `[…]` — optionality: the enclosed pattern may appear zero or one time.
  - **E.g.,** `<number> ::= ["-"] <digits>`, match:
    > `"1"`, `"-320"`
- `(…)` — grouping: used to group sequences or alternatives.
  - **E.g.,** `<sum> ::= <number> ("+" | "-") <number>`, matches:
    > `"1 + 2 - 3"`, `"-1 + 2"`, `"5"`
- `"…"` — terminal strings may be written in quotes. This clears ambiguity between terminal and non-terminal symbols.
  - **E.g.,** in `<digit> ::= A | B | C`, it's clear that A, B, and C refer to non-terminal symbols. However, in the expression `<expr> ::= <expr> | "(" <expr> ")"`, the parenthesis are **crucial** in indicating that it is not grouping notation.

Let's elaborate more on `<digits>` from above:

Example 2.1: Dealing with Optionality

In our previous grammar from Definition (2.4), we had:

```
<digit> ::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
```

which matched:

> "0", "320", "11235813"

But also unintentionally matched:

> "" *(empty string)*

> "000", "0123" *(leading zeros)*

Let's first fix the empty string case:

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digits> ::= <digit> {<digit>}
```

To restrict leading zeros (e.g., for natural numbers), we refine further:

```
<nonzero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digits> ::= "0" | <nonzero> {<digit>}
```

Adding quotes around `0` is a stylistic choice and does not change the meaning of the grammar. We could have written:

```
<nonzero> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<digits> ::= "0" | <nonzero> {<digit>}
```

Or without quotes, it doesn't matter as long as there is no ambiguity. ■

To be more precise with our language we briefly discuss **Automata Theory** and types of grammars:

Definition 2.5: Automata Theory

In mathematics, an automaton (Plural form: automata) is an abstract machine that follow a sequence of states (similar to a flowchart). A finite number of states describe a **finite automaton (FA)** or **finite-state machine (FSM)**.

Tip: Automata comes from the Greek word *αυτόματος* (autómatos), which means "self-acting, self-willed, self-moving"

Consider the following automaton described by a state machine diagram:

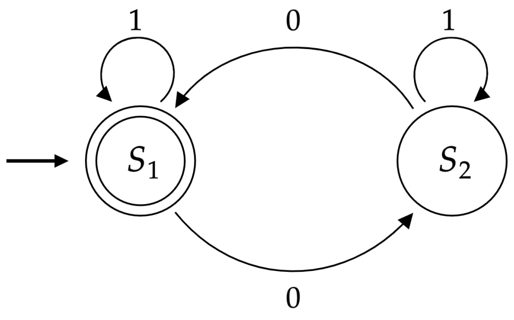


Figure 1.14: An automaton represented as a state-machine diagram. This state-machine generates binary strings with an even number of 0's. There are two states, S_1 and S_2. The automaton starts in state S_1 as shown by the arrow coming from blank space. The double circle indicates the **accepting state**, at which the automaton may terminate.

Definition 2.6: Grammar Types

In formal language theory, grammars are classified into a hierarchy based on their generative power. This classification, known as the **Chomsky hierarchy**, comprises four primary types that utilize BNF:

- **Type 3: Regular Grammars**
  Generate **regular languages**, which can be recognized by finite automata. These grammars can describe simple flat patterns, but cannot handle nested or recursive structures.

- **Type 2: Context-Free Grammars (CFGs)**
  Generate **context-free languages** and are recognized by pushdown automata. A **push-down automaton** is a finite automaton extended with a stack—a Last-In, First-Out (LIFO) memory structure. This allows it to handle nested and recursive structures, such as balanced parentheses.

- **Type 1: Context-Sensitive Grammars**
  Generate **context-sensitive languages** and are recognized by linear bounded automata. A **linear bounded automaton** is a Turing machine where the tape is limited to a length proportional to the input size. This constraint gives it more power than pushdown automata but less than unrestricted Turing machines.

- **Type 0: Unrestricted Grammars**
  Generate **recursively enumerable languages** and are recognized by Turing machines. These grammars have no restrictions on production rules and can describe any computable language, though some of these languages are undecidable.

We can visualize the Chomsky hierarchy as follows:

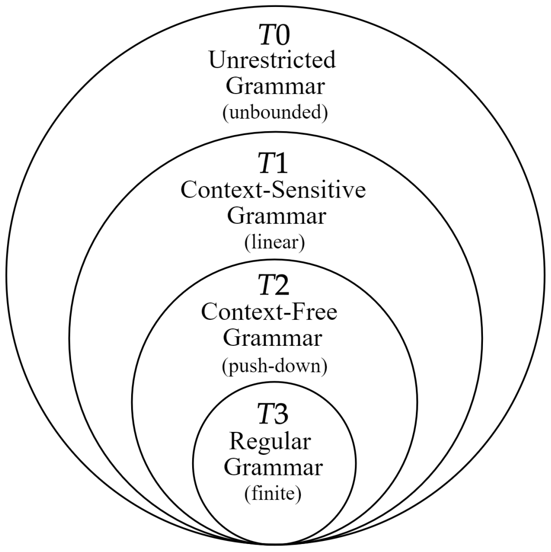


Figure 1.15: The Chomsky hierarchy: T_3 finite automata (regular expressions), T_2 pushdown automata (context-free grammars), T_1 linear bounded automata (context-sensitive grammars), and T_0 Turing machines (unrestricted grammars).

Definition 2.7: Regular Expressions (Regex)

Regular expressions (Regex) provide a compact way to describe patterns in regular grammars:

- `a` — a single **terminal** symbol is itself a regex.
- `[t1 t2 …tk]` — character class: matches any one of the listed characters.
- `(e1 | e2 | …| ek)` — alternation: matches any one of the enclosed expressions.
- `exp*` — repetition: matches **zero or more occurrences** of `exp`.
- `exp+` — repetition: matches **one or more occurrences** of `exp`.
- `exp?` — optional: matches **zero or one occurrence** of `exp`.

E.g., the regex `(a|b)*c` matches any string of a's and b's followed by a c, such as:

> `"abc"`, `"aabbbbc"`, `"c"`

This is just a small subset of the Regex syntax. To learn more, consider the following resource: https://regexlearn.com/

1.2.2 Menhir: Simple Lexing & Parsing

To formally retouch on the definitions of lexing:

Definition 2.8: Lexical Analysis

Lexical analysis concerns itself with the **tokenization** of a program. In particular, converting a stream of characters into a stream of tokens (whitespace and comments are ignored).

Lexing (to lex) works by scanning the input for the next longest valid token. Then it returns that token and the remaining input to lex.

E.g., `" let@#_)($#@_J_@O#GKJ"` gives us `(LET, "@#_)($#@_J_@O#GKJ")`, where `LET` is the token and `"@#_)($#@_J_@O#GKJ"` is the remaining input, which is most likely to result in an error (unless the lexer is designed to handle such cases).

We will now create a lexer and parser for a simple toy-language that only has addition, subtraction, multiplication, and division. Menhir should already be installed from the onboarding Section (**??**).

Example 2.2: Simple Lexer & Parser (Part 1)

Let's create a simple lexer and parser for a toy language that only has addition, subtraction, multiplication, and division. We will use Menhir to generate the parser.

Let's create the project:

```
        dune init proj small_arith
        cd small_arith
```

This should create:

```
small_arith/
├── _build/
│   └── log
├── bin/
│   ├── dune
│   └── main.ml
├── dune-project
├── lib/
│   └── dune
├── small_arith.opam
└── test/
    ├── dune
    └── test_small_arith.ml
```

■

Example 2.3: Simple Lexer & Parser (Part 2)

Create these new files, `lib/ast.ml`, `lib/lexer.mll`, and `lib/parser.mly`:

```
touch lib/dune lib/ast.ml lib/lexer.mll lib/parser.mly
```

- `ast.ml` — A regular OCaml source file (`.ml`) where we define the core data structures via ADTs, which will generate the **abstract syntax tree (AST)** for our language.
- `lexer.mll` — The (`.mll`) extension to signal it will be processed by **OCamllex**, a lexer generator. Here, we write rules using regular-expression-like syntax to convert raw strings into **tokens** like `NUM`, `ADD`, and `LPAREN`.
- `parser.mly` — The (`.mly`) extension is handled by **Menhir**. It defines the **grammar** of our language. The result of parsing is an AST defined in `ast.ml`.

Together: `Input String` → `Lexer (mll)` → `Tokens` → `Parser (mly)` → `AST (ml)`

Lets begin by defining the AST in `ast.ml`:

```
(* ast.ml *)
type bop = Add | Sub | Mul | Div

type expr =
  | Var of string
  | Num of int
  | Let of string * expr * expr
  | Bop of bop * expr * expr

type prog = expr
```

The above code defines the abstract syntax tree (AST) for our toy language. It includes:

- `bop` — a type for binary operators (addition, subtraction, multiplication, division).
- `expr` — a type for expressions, which can be a variable, number, let-binding, or binary operation.
- `prog` — a type for the program, which is just an expression.

This should be familiar from the BNF grammar we've discussing. ■

Example 2.4: Simple Lexer & Parser (Part 4)

Now let's define our lexer in `lib/lexer.mll`:

```
(* lexer.mll *)
{
    open Parser
}

(* Regex for tokens *)
let whitespace = [' ' '\t' '\n' '\r']+
let var = ['a'-'z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_' '\'']*
let num = ['0'-'9']+

rule read = parse
    | whitespace { read lexbuf }
    | "let"      { LET }
    | "in"       { IN }
    | "="        { EQ }
    | "+"        { ADD }
    | "-"        { SUB }
    | "*"        { MUL }
    | "/"        { DIV }
    | "("        { LPAREN }
    | ")"        { RPAREN }
    | num        { NUM (int_of_string (Lexing.lexeme lexbuf)) }
    | var        { VAR (Lexing.lexeme lexbuf) }
    | eof        { EOF }
```

- `lexbuf`: short for **"lexing buffer,"** is an object created by the OCaml stdlib. It stores the input string and tracks the position of the lexer as it reads through.
- `read lexbuf`: A recursive call to the lexer function. It's used to **skip over whitespace**. When the lexer sees a space or newline, it just calls itself again to scan the next part of the input.
- `Lexing.lexeme lexbuf`: This function returns the **exact substring** that was matched by our Regex. For example, if the input is `"42 + 1"` and we match `num`, then `Lexing.lexeme lexbuf` returns `"42"` as a string. We then convert it to an integer.
- `rule read = parse`: This defines the main lexer function. The `read` is the entry point of the lexer. The `parse` keyword indicates parsing rules for the input to match against.

■

Example 2.5: Simple Lexer & Parser (Part 5)

Finally the parser in `lib/parser.mly` utilizing our `ast.ml` (Moreover on the next page.):

```
%{
  open Ast
%}

%token LET EQ IN
%token ADD SUB MUL DIV
%token LPAREN RPAREN
%token <int> NUM
%token <string> VAR
%token EOF

%left ADD SUB
%left MUL DIV

%start <Ast.prog> prog
%%

prog:
  | e = expr; EOF { e }

expr:
  | LET; x = var; EQ; e1 = expr; IN; e2 = expr
    { Let (x, e1, e2) }
  | e = expr1 { e }

%inline bop:
  | ADD { Add }
  | SUB { Sub }
  | MUL { Mul }
  | DIV { Div }

expr1:
  | e1 = expr1; op = bop; e2 = expr1
    { Bop (op, e1, e2) }
  | n = num { Num n }
  | v = var { Var v }
  | LPAREN; e = expr; RPAREN { e }

num:
  | n = NUM { n }

var:
  | x = VAR { x }
```

■

Example 2.6: Simple Lexer & Parser (Part 5a)

The `parser.mly` file looks quite different from regular OCaml code because it follows the syntax of Menhir (an LR(1) parser generator). Let's walk through its structure:

- `%{ ... %}` — Opens a block of OCaml code. We use it to import our AST module.
- `%token` — Declares the tokens the parser will receive from the lexer. If tokens carry data integer or string, we write it as:

```
%token <int> NUM
%token <string> VAR
```

- `%left` — Defines left-associativity, and precedence is by order of declaration:

```
%left ADD SUB
%left MUL DIV
```

  means that `MUL` and `DIV` bind more tightly than `ADD` and `SUB`. All are left-associative. This prevents ambiguity in expressions like `3 + 4 * 2`.

- `%start <Ast.prog> prog` — The entry point of the parser is the rule `prog`. The return type is `Ast.prog` from our AST module.
- `<name>:` — Each rule has the form:

```
name:
  | pattern1 { action1 }
  | pattern2 { action2 }
```

  Think of it as EBNF, but instead we use OCaml pattern matching syntax.

- `%inline <name>:` — A helper rule that lets us **macro** grammar patterns into a single variable without generating a new nonterminal or type (cannot be recursive):

```
%inline bop:
  | ADD { Add }
  | SUB { Sub }
  | ...
```

■

Example 2.7: Simple Lexer & Parser (Part 5b)

Let’s look closely at the following grammar rule:

```
expr:
    | LET; x = var; EQ; e1 = expr; IN; e2 = expr
    { Let (x, e1, e2) }
```

- Each item before the ‘`{`’ is a **token** (e.g., `LET`, `EQ`, `IN`) or a **nonterminal rule** (e.g., `var`, `expr`).
- When a nonterminal is matched, we can bind its result using `name = rule`. For example, `x = var` means: “run the `var` rule and bind its result to `x`.”
- The block in `{ }` is an OCaml expression that builds an AST node using the bound variables. Notice, `Let (x, e1, e2)` our ADT constructor from `ast.ml`.

When we parse a string like:

```
let x = 3 + 4 in x * 2
```

The lexer produces a stream of tokens:

```
[LET; VAR("x"); EQ; NUM(3); ADD; NUM(4); IN; VAR("x"); MUL; NUM(2); EOF]
```

Then, the parser:

- Sees `LET`, then `VAR("x")` → runs the `var` rule, binds to `x`.
- Sees `EQ`, then parses the expression `3 + 4` using the `expr` rule, binds to `e1`.
- Sees `IN`, then parses the expression `x * 2`, binds to `e2`.

Finally, it constructs:

```
Let (x, e1, e2)
```

This is now a fully constructed OCaml value using the constructors we defined in `ast.ml`.

■

Example 2.8: Setting Up Dune and Testing in utop

Now that we've written our AST, lexer, and parser, we need to set up Dune so we can compile everything and test it in `utop`.

First, open your `dune-project` file and add the following line at the bottom to enable Menhir:

```
(using menhir 3.0)
```

Your complete `dune-project` should now look like:

```
(lang dune 3.0)
(name small_arith)
(using menhir 3.0)
```

Next, open `lib/dune`:

```
(library
    (name small_arith))

(ocamllex lexer)
(menhir (modules parser))
```

Create a `lib/demo.ml` file to test the lexer and parser:

```
let parse s =
    try Some (Parser.prog Lexer.read (Lexing.from_string s))
    with _ -> None
```

From the root of your project, run:

```
dune build
dune utop lib
```

In `utop`, you can test the parser like this:

```
# Demo.parse "let x = 3 + 4 in x * 2";;
- : Demo__Ast.prog option =
Some
(Demo__.Ast.Let ("x",
Demo__.Ast.Bop (Demo__.Ast.Add, Demo__.Ast.Num 3, Demo__.Ast.Num 4),
Demo__.Ast.Bop (Demo__.Ast.Mul, Demo__.Ast.Var "x", Demo__.Ast.Num 2)))
```

■

Bibliography

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