

# Compilers

Muchang Bahng

Spring 2026

## Contents

<b>1 Scanning</b>	<b>3</b>
1.1 Token Types . . . . .	4
1.2 Handwritten Lexer . . . . .	9
<b>2 Generated Lexers</b>	<b>12</b>
<b>3 Parsing</b>	<b>13</b>
3.1 Precedence and Associativity . . . . .	13
3.2 Abstract Syntax Trees . . . . .	13
3.3 Recursive Descent Parsing . . . . .	15
3.4 Bytecode and Pratt Parsing . . . . .	15
3.5 LL Parsing . . . . .	15
3.6 LR Parsing . . . . .	18
<b>4 Resolution</b>	<b>19</b>
<b>5 Type Checking</b>	<b>20</b>
<b>6 Tree Walk Interpreters</b>	<b>21</b>
6.1 Evaluation of Expressions . . . . .	21
6.2 Environments . . . . .	21
<b>7 Bytecode Interpreters</b>	<b>22</b>
7.1 Stack-Based Virtual Machines . . . . .	23
7.2 Register-Based Virtual Machines . . . . .	24

Compiling is the process of converting code that you write (essentially a giant string) into assembly, for a specific ISA. You can use a *cross-compilation toolchain*, where you have a machine for one ISA (e.g. x86) but compile it into ARM with—say the `gccarm` package.

Languages themselves are not compiled or interpreted. Their implementations are. Second, the dichotomy between compiling and interpreting is much more grey.

1. *Lexer*. Convert a sequence of characters into a sequence of tokens. Whitespace and comments are not tokens, since they get dropped out. Need to talk about DFA, NFA, and regular expressions.
2. *Parsing*. Building the abstract syntax tree. e.g. LL parsing (what we are doing) vs LR parsing. MLYac. Trees make everything explicit and so is easier to work with.
3. *Type Checking*. Just a bit of work.
4. *IR*. Top of the mountain.
5. *Instruction Selection*.
6. *Liveness Analysis*. Data flow analysis, which is at the core of a lot of optimization.
7. *Register Allocation*. This gives the MIPS assembly, which is text.

# 1 Scanning

If we look at our source code—whether it'd be in Python, Java, or C—they are all just a giant string. Every program is just a string composed of characters in our alphabet.<sup>1</sup> This motivates the following definitions.

## Example 1.1 (C Identifiers)

The set  $\Sigma = \{\_, a, \dots, z, A, \dots, Z, 0, \dots, 9\}$  can be the alphabet of the formal language  $L$  representing all variable identifiers in the C programming language.

## Example 1.2 (Implementations in Languages)

While the concept of “importing” other files into the current file is similar, the process is very different across languages.<sup>a</sup>

1. In C, preprocessing directives are a part of lexing. For example, `#include` literally copies and pastes the contents of the target module into the current file. With `#IFDEF`, `#IFNDEF`, `#ENDIF`, we can delete tokens from our file to do *conditional compilation*.
2. In Python, the `import` keyword is a runtime behavior done during execution of the statement. That is, doing something like

```
1 import math
2 math.sqrt(9)
```

can be thought of as doing something like

```
1 math = load_module("math")
2 math.sqrt(9)
```

It is basically like executing a function or instantiating a class.

3. In Java, the `import` keyword is closer to a name shortcut, done during resolution. These import statements allow us to write

```
1 import java.util.ArrayList;
2
3 public static void main(String arg) {
4     ArrayList<Integer> x = new ArrayList<>();
5 }
```

rather than having to do

```
1 public static void main(String arg) {
2     java.util.ArrayList<Integer> x = new java.util.ArrayList<>();
3 }
```

Basically, it just tells Java that when I write `ArrayList`, I mean `java.util.ArrayList`. Now, it is clear that this is more of a name resolution process, where we are binding the `ArrayList` name to `java.util.ArrayList`.

<sup>a</sup><https://chatgpt.com/share/6a155f53-3404-83a7-8349-03818c8b23de>

<sup>1</sup>Technically, not just English letters, but other ASCII characters plus other UTF-8.

## 1.1 Token Types

The immediate problem with this representation of our code is that our alphabet is too *fine*.<sup>2</sup> We need a slightly more restrictive alphabet, which give us a better building blocks of our language. In fact, choosing the set of token types is one of the first things you might want to think about when making a new language. Since we want to convert a series of characters (our program) into tokens, what tokens should we have in our language? Think about all the characters you have seen so far in programs, plus how they are used within and across different languages.

We start off with the single character tokens.

### Example 1.3 (Parentheses)

Parentheses are used universally in almost all languages. We will label them `<LEFT_PAREN>` and `<RIGHT_PAREN>`.

1. *Precedence*. In C, Python, and Java, parentheses indicate some precedence in the order of operations, e.g. `(3 + 4) * 5`.
2. *Function Calls*. They also represent function calls, e.g. `int x = myFunc()`.
3. *Function Declaration*. They are used in function declarations, e.g. `public void myFunc(int x, int y) { ... }`.

### Example 1.4 (Braces)

Braces are also pretty universal. We will denote them `<LEFT_BRACE>` and `<RIGHT_BRACE>`.

1. *Block Delimiters*. In C, C++, and Java, braces represent the start and end of a block, whether it'd be in a function declaration, an if/else statement, or a loop.
2. *F-Strings*. In Python, braces can be used in f-strings, e.g. `name = "Bob"; print(f"Hello {name}")`.

### Example 1.5 (Brackets)

For brackets, we will denote them `<LEFT_BRACKET>` and `<RIGHT_BRACKET>`.

1. *Indexing*. They are used to retrieve a value stored in the *i*th position of an array (in C, Java) or a list (Python).
2. *List Creation*. In Python, they are used to initialize a list, e.g. `x = [1, 2, 3]`.

### Example 1.6 (Comma)

The comma token is denoted `<COMMA>`.

1. *Function Parameters*. You use commas to separate parameters in function declarations, e.g. `def myPythonFunction(x, y, z): ...`
2. *Function Arguments*. You use commas to separate arguments in function calls, e.g. `int y = myFunc(4, 3);`.
3. *Unpacking*. When returning an iterable in Python, you can unpack them with commas, e.g. `x, *, y = someList`.

<sup>2</sup>In English, sentences are indeed made up of letters, but it would be better to represent them as a list of words.

**Example 1.7 (Dot)**

The dot token is denoted <DOT>.

1. *Attribute Retrieval.* In OOP, you can use it to retrieve an attribute from an object, e.g. `student.name`.
2. *Method Call.* In OOP, you can use it to call an instance or static method, e.g. `MyClass.Stuff()`.

**Example 1.8 (Plus)**

The plus token is denoted <PLUS>.

1. *Addition.* In almost all languages, you use this to add two numbers (int, double) together, e.g. `int x = y + z;`
2. *Concatenation.* In Python, you use this to concatenate two strings into a longer string.

**Example 1.9 (Minus)**

The minus token is denoted <MINUS>.

1. *Subtraction.*

**Example 1.10 (Star)**

The star is denoted <STAR>.

1. *Multiplication.*
2. *Repeat.* In Python, you can multiply a string by an int to concatenate the string with itself multiple times, e.g. `"hello" * 4`.

**Example 1.11 (Star Star)**

The double star token is denoted <STAR\_STAR>.

1. *Exponent.* In Python, it indicates an exponent, e.g. `3 ** 2`.

**Example 1.12 (Slash)**

The slash is denoted <SLASH>.

1. *Division.*

**Example 1.13 (In-Place Arithmetic)**

The following tokens are used commonly for in-place arithmetic.

1. *Increment.* We use the <PLUS\_PLUS> token to represent increments, e.g. `i++`.
2. *Decrement.* We use the <MINUS\_MINUS> token to represent decrements, e.g. `i--`.
3. *In-Place Addition.* We use the <PLUS\_EQUAL> token to represent in-place addition, e.g. `i += 2`.
4. *In-Place Subtraction.* We use the <PLUS\_EQUAL> token to represent in-place subtraction, e.g. `i -= 2`.
5. *In-Place Multiplication.* We use the <PLUS\_EQUAL> token to represent in-place multiplication, e.g. `i *= 2`.
6. *In-Place Division.* We use the <PLUS\_EQUAL> token to represent in-place division, e.g. `i /= 2`.

**Example 1.14 (Backslash)**

The backslash is denoted <BACKSLASH>.

1. *Escape Sequence.*

**Example 1.15 (Hashtag)**

The <HASHTAG> token is used to represent #.

1. *Comment.* In Python, this is used to represent single line comments, e.g. `# comment`.
2. *Directives.* In C and C++, it is used to represent preprocessing directive, e.g. `#include <stdio.h>` or `#IFDEF`.

**Example 1.16 (At)**

The <AT> token is used to represent .

1. *Macros.* In Python and Java, this is used to represent macros for methods and functions, e.g. `@property` or `@Override`.

**Example 1.17 (Semicolon)**

The semicolon is denoted <SEMICOLON>.

Next, we continue onto tokens that are one-character or two characters long.

**Example 1.18 (Logic Operators)**

The logic operators may be represented as these character sequences (like in C++ or Java), or as keywords (like in Python).

1. *Negation.* The `!` character represents the <BANG> token. In C, C++, and Java, we can take a boolean value and negate it, e.g. `!true`.
2. *Or.* The `||` characters often represent the <OR> token.
3. *And.* The `&&` characters often represent the <AND> token.

**Example 1.19 (Equal)**

The <EQUAL> token almost universally represents assignment, e.g. `x = 4`.

**Example 1.20 (Greater, Less)**

The <LESS> and <GREATER> tokens can represent the following.

1. *Comparison.* In almost all languages, these are used as comparison operators, e.g. `4 < 5`.
2. *Template Parameters.* In C++, you can use them as template parameters, e.g. `template <typename T>`.
3. *Element Type.* In Java, the types of the elements in the standard library data structures are specified within these angle brackets, e.g. `HashMap<String, Integer>`.

**Example 1.21 (Equal Equal, Not Equal, Less Equal, Greater Equal)**

The following four mean the same thing across all languages that I have seen so far.

1. *Equal Equal*. The `<EQUAL_EQUAL>` token is used to determine whether two values are equal.
2. *Not Equal*. The `<BANG_EQUAL>` token is used to determine whether two values are not equal.<sup>a</sup>
3. *Less Equal*. The `<LESS_EQUAL>` token is used to determine whether the previous token is less than the token after.
4. *Greater Equal*. The `<GREATER_EQUAL>` token is used to determine whether the previous token is greater than the token after.

---

<sup>a</sup>Note that this is completely separate from the concatenation of tokens `<NOT><EQUAL_EQUAL>`.

**Example 1.22 (Whitespace)**

In most languages, whitespace are not tokens. The exception is Python, which treat tabs as significant for identifying scope. We can label them as `<TAB>`.

Next, we talk about variable-length token types.

**Example 1.23 (Identifiers)****Example 1.24 (String)****Example 1.25 (Numbers)**

We can represent multiple types of numbers.

1. *Number*. The `<NUMBER>` token can be used to generically represent any number in your language. In Lox, this is the only token type.
2. *Integer*. The `<INTEGER>` token is used to represent integers.
3. *Float*. The `<FLOAT>` or `<DOUBLE>` token is used to represent floating point numbers.

Next, we talk about the keywords.

**Example 1.26 (Boolean)**

We can implement token types to represent boolean values `<TRUE>` and `<FALSE>`.

**Example 1.27 (Logic Keywords)**

Just as we have logic tokens, we have

1. *And*.
2. *Or*.
3. *Not*.

**Example 1.28 (Null Value)**

The null value may be represented depending on the language. It may be called `<NULL>`, `<NONE>`, or `<NIL>`.

**Example 1.29 (Control Flow)**

Control flow also has a bunch of keywords.

1. *Conditionals*. We have token types like `<IF>`, `<THEN>`, `<ENDIF>`, `<ELSEIF>`, and `<ELSE>`.
2. *For Loops*. We have token types like `<FOR>`, `<DO>`, `<ENDFOR>`.
3. *While Loops*. We have token types like `<WHILE>`, `<DO>`, `<ENDFOR>`.

**Example 1.30 (Functions)**

We have stuff like.

1. *Declaration*. Python uses `<DEF>` and JavaScript uses `<FUN>`.
2. *Return*. We have `<RETURN>`.

**Example 1.31 (Variable Declaration)**

We can use `<VAR>` or `<LET>`.

**Example 1.32 (Classes)**

In object oriented languages, we can't forget about

1. `<CLASS>` used to declare a new class.
2. `<SUPER>` used to reference the parent class.
3. `<THIS>` used to reference the current instance of the class.

**Example 1.33 (Print)**

The `<PRINT>` token type may be used as a keyword. Often though, it is implemented as a native function.<sup>a</sup>

---

<sup>a</sup>More on this later.

**Example 1.34 (Begin, End)**

The `<BEGIN>` and `<END>` tokens are often used as delimiters of blocks rather than `<LEFT_BRACE>` and `<RIGHT_BRACE>`.

**Example 1.35 (Comment)**

Most of the times, comments are thrown away, but in certain cases one may want to keep them around as tokens. Then, we will have a `<COMMENT>`.

**Example 1.36 (End of File)**

Often, languages would implement an end-of-file token, denoted <EOF>.

The whole process of lexing is to convert a giant string (your code) into a sequence of tokens. But this requires us to classify which token the next substring is, i.e. whether it's an identifier, a keyword, a literal, etc. So basically, we need to match these incoming words to their respective token type, and we can do this by pattern matching. If you have thought of regular expressions to do this, you are exactly on the right track.

```
1  typedef enum {
2      // Single-character tokens.
3      TOKEN_LEFT_PAREN, TOKEN_RIGHT_PAREN,
4      TOKEN_LEFT_BRACE, TOKEN_RIGHT_BRACE,
5      TOKEN_COMMA, TOKEN_DOT, TOKEN_MINUS, TOKEN_PLUS,
6      TOKEN_SEMICOLON, TOKEN_SLASH, TOKEN_STAR,
7      // One or two character tokens.
8      TOKEN_BANG, TOKEN_BANG_EQUAL,
9      TOKEN_EQUAL, TOKEN_EQUAL_EQUAL,
10     TOKEN_GREATER, TOKEN_GREATER_EQUAL,
11     TOKEN_LESS, TOKEN_LESS_EQUAL,
12     // Literals.
13     TOKEN_IDENTIFIER, TOKEN_STRING, TOKEN_NUMBER,
14     // Keywords.
15     TOKEN_AND, TOKEN_CLASS, TOKEN_ELSE, TOKEN_FALSE,
16     TOKEN_FOR, TOKEN_FUN, TOKEN_IF, TOKEN_NIL, TOKEN_OR,
17     TOKEN_PRINT, TOKEN_RETURN, TOKEN_SUPER, TOKEN_THIS,
18     TOKEN_TRUE, TOKEN_VAR, TOKEN_WHILE,
19
20     TOKEN_ERROR, TOKEN_EOF
21 } TokenType;
```

Figure 1

Then, we can define a Token.

```
1  typedef struct {
2      TokenType type;
3      const char* start;
4      int length;
5      int line;
6  } Token;
```

## 1.2 Handwritten Lexer

Now that we have regex's, we can just pattern match for each substring. We employ this using a classic two pointer strategy.<sup>3</sup>

<sup>3</sup>I initially tried it using a stack, but I found out later than a two pointer was much easier and faster.

**Example 1.37 (Ambiguities)**

If you are trying to lex the string:

```
1  !=
```

You can either tokenize it as `<BANG_EQUAL>` or `<BANG><EQUAL>`. This creates an ambiguity in our tokenizer.

**Definition 1.1 (Maximal Munch)**

The principle of tokenizing the maximum length substring at a given time is called **maximal munch**.

**Algorithm 1.1 (Scanner)**

The general idea is to use a two pointer approach, where `i` represents the start index of the current token and `j` represents the current index of the current token. Some tips for implementation:

```

1: procedure SCAN(source)
2:   tokens ← []
3:   i ← 0
4:   j ← 0
5:   while j < source.length() do
6:     c = jth character of source
7:     if c ∈ { ( , ) , { , } , , , . , - , + , ; , * } then                                ▷1-char tokens
8:       convert c to corresponding token and append it to tokens
9:       increment j
10:    else if c == / then                                                                ▷It could be a division, comment, or multiline comment
11:      c2 ← jth character of source  ▷We peek ahead to find out which token / is a part of
12:      if c2 = / then                                                                    ▷If this is single-line comment
13:        Keep incrementing j until we hit a newline.
14:        Increment j once more
15:      else if c2 = * then                                                                ▷If this is multiline comment
16:        Keep incrementing j until we hit */
17:        Increment j once more
18:      else                                                                                ▷This must be a division
19:        Add <SLASH> token
20:        Increment j
21:      end if
22:    else if c = ! then
23:      Peek ahead and add either <BANG_EQUAL> or <BANG>
24:      Increment j past this token.
25:    else if c = = then
26:      Peek ahead and add either <EQUAL_EQUAL> or <EQUAL>
27:      Increment j past this token.
28:    else if c = > then
29:      Peek ahead and add either <GREATER_EQUAL> or <GREATER>
30:      Increment j past this token.
31:    else if c = < then
32:      Peek ahead and add either <LESS_EQUAL> or <LESS>
33:      Increment j past this token.
34:    else if c = " then
35:      Keep incrementing j until we hit "

```

```
36:         Increment j past this token.
37:     else if c is a digit then
38:         Keep incrementing j until we hit the end of the number.
39:         Increment j past this token.
40:     else if c is a letter then
41:         Keep incrementing j as long as the character is alphanumeric or underscore.
42:         If substring from i to j is a keyword, add the corresponding token.
43:         Otherwise, this is an identifier.
44:     end if
45: end while
46: Append <EOF> token to tokens.
47: return tokens
48: end procedure
```

Note that the maximal munch principle is always applied in every conditional statement except for the first branch with 1-character tokens.

1. For the 1-character tokens, you can just use a switch statement which is faster, and to detect the identifiers/keywords/numbers, we can use an if statement in the default.
2. To see keyword vs identifier, you usually use a hashmap to check for keywords (e.g. `if` → `<IF>`), and if there are no matches, then you catch it with a regex.
3. The current index `j` keeps track of the column, but it might be nice to keep a second variable `line` tracking the current line.

## 2 Generated Lexers

It turns out that using handwritten lexers is not the most powerful nor practical way to do this. Remember that all scanning does is pattern matching using regular expressions. In modern days, we have software that can take in whatever set of regular expressions you give it, and it will convert it to an equivalent DFA. Therefore, to detect tokens, we run it through a DFA and see the accepting state it lands on.

Again, with multiple matches, the problem is manifested with multiple dead states. If it is in a dead state, then output the most recent<sup>4</sup> accepting state.

### Example 2.1 (Ambiguities in Tokenizing)

Say that we have a language with the following tokens.

1. `<x>`: `x`.
2. `<nxy1>`: `(x*)y`
3. `<z>`: `z`

This leads to the following DFA.

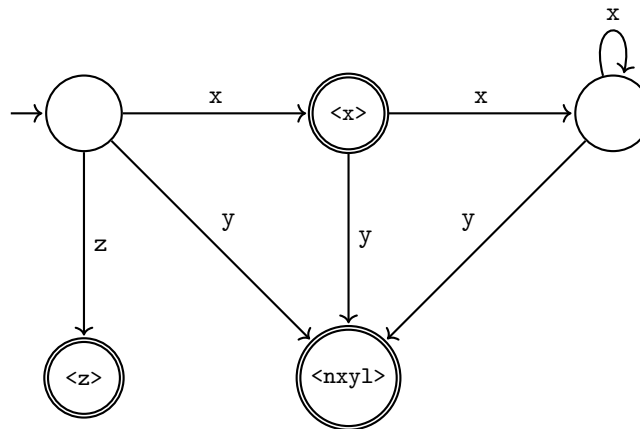


Figure 2

Then, to parse “`xxxzxxxxy`,” we have 3 choices, which leads to ambiguity.

1. `<x>` `<x>` `<x>` `<z>` `<nxy1>`
2. `<x>` `<x>` `<x>` `<z>` `<x>` `<nxy1>`
3. `<x>` `<x>` `<x>` `<z>` `<x>` `<x>` `<nxy1>`

Again, we can use maximal munch.

Lexers can have multiple DFAs, and depending on context (through start state), you traverse different DFAs (e.g. code vs comments).

<sup>4</sup>due to principle of maximal munch

### 3 Parsing

Once we are done scanning, we now have a list of tokens ( $t_n$ ). The next job is to determine if this list of tokens is valid syntax for our language, i.e.  $t_1 t_2 \dots t_n \in L$ . So, the parser's first job is to determine membership of our sequence of tokens. This seems like a pretty hard problem since  $L$  can be very complex.

NOTE: Introduce parsing, then next 2 subsections should be the final result of what we want the parsed output to look like, i.e. either an AST or bytecode.

We basically want not only inclusion but a parse tree. But a sequence of tokens may have 2 valid parse trees.

#### Example 3.1

For example,  $2 + 3 * 5$  should really be translated to

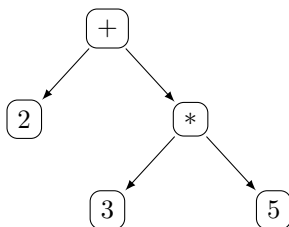


Figure 3: Syntax tree for the expression  $2 + 3 * 5$ .

We could sometimes refactor our grammar or sometimes use precedence directives.

#### 3.1 Precedence and Associativity

##### Definition 3.1 (Precedence)

**Precedence** is an ordering on operators that determines which operator becomes deeper in the parse tree when an expression contains multiple possible parses. When an operator has higher precedence than another, it is said to **bind more tightly**.

##### Example 3.2 (Assignment)

Assignment is usually right associative.

$$a = b = c = d \iff a = (b = (c = d)) \quad (1)$$

#### 3.2 Abstract Syntax Trees

Parse trees are grammar level structure while AST is implementation level structure. The three major AST node types are:

##### Definition 3.2 (Expression)

An **expression** is a piece of syntax that can be *evaluated* to produce a value.

**Definition 3.3 (Statement)**

A **statement** is a piece of syntax that can be *executed* to produce an effect.

**Definition 3.4 (Declaration)**

A **declaration** is a piece of syntax that is used to introduce a name or binding.

**Definition 3.5 (Categories of Words)**

In a language,

1. a **literal** is a word in  $L$  that represents a fixed value directly in the source code.
2. an **identifier** is a word in  $L$  used to name entities such as variables, functions, and types.
3. a **reserved word**, or **keyword**, is an identifier that has a fixed meaning in the language's grammar.

**Example 3.3 (Grammar Rule for jLox)**

```

1 program    -> declaration* EOF
2 declaration -> varDecl
3           | funDecl
4           | statement;
5 varDecl   -> "var" IDENTIFIER = exprStmt
6 funDecl   -> "fun" function;
7 function  -> IDENTIFIER "(" parameters? ")" block ;
8 parameters -> IDENTIFIER ("," IDENTIFIER)* ;
9 statement -> exprStmt
10          | printStmt
11          | Block
12          | ifStmt
13          | whileStmt
14          | forStmt
15          | returnStmt
16 block     -> "{" declaration* "}";
17 exprStmt  -> expression ";"
18 printStmt -> "print" expression ";"
19 ifStmt    -> "if" "(" expression ")" statement ("else" statement )? ;
20 whileStmt -> "while" "(" expression ")" statement;
21 forStmt   -> "for" "(" (varDecl | exprStmt | ) ";" expression? ";" expression? ")"
           statement;
22 returnStmt -> "return" expression? ";" ;
23 expression -> assignment;
24 assignment -> IDENTIFIER "=" assignment
25           | logic_or;
26 logic_or   -> logic_and ( "or" logic_and )*
27 logic_and  -> equality ( "and" equality )*
28 equality   -> comparison ( ( "==" | "!=" ) comparison )*
29 comparison -> term      ( ( ">" | ">=" | "<" | "<=" ) term )*
30 term       -> factor    ( ( "-" | "+" ) factor )*
31 factor     -> unary    ( ( "/" | "*" ) unary )*
32 unary      -> ( "!" | "-" ) unary
33           | call
34 call       -> primary ( "(" arguments? ")"*);

```

```

35 arguments  -> expression ("," expression)*;
36 primary   -> NUMBER
37           | STRING
38           | "true"
39           | "false"
40           | "nil"
41           | "(" expression ")"
42           | IDENTIFIER

```

### 3.3 Recursive Descent Parsing

The simplest form of parsing, depending on how you do this, this will determine associativity for sure. To resolve ambiguities, recursive descent mostly does grammar refactoring.

Parsing expressions. Should be okay since we defined precedence and associativity before. Parsing statements. Parsing declarations.

### 3.4 Bytecode and Pratt Parsing

Another top-down (LL-like) parsing method that outputs an array of bytecode rather than an AST.

```

1  typedef struct {
2      Token previous;
3      Token current;
4  } Parser;

```

### 3.5 LL Parsing

The whole purpose of parsing is to derive an order of operations of our tokens. There are two types of parsing. We will start out with *LL parsing*, which is easier to construct by hand.

#### Definition 3.6 (LL Parsing)

**LL parsing** refers to parsing while reading the input from left-to-right, with leftmost derivation, aka “top-down” or “recursive descent.” As the name suggests, we write a bunch of mutually recursive functions—one function per non-terminal symbol. Any time we need to parse the nonterminal symbol, we call that function.

Let us take the grammar  $G = (V, \Sigma, R, S)$  with nonterminals  $V = \{S, C, A, B, Q\}$ , terminals  $\Sigma = \{x, a, d, b, q\}$ , start symbol  $S$ , and production rules. Note that for any symbol  $X$ ,  $X \rightarrow$  is shorthand for  $X \rightarrow \epsilon$ , i.e. the rule that can “delete”  $X$  without any cost. Furthermore,  $\$$  indicates an *end of input*.

1.  $S \rightarrow AC\$$
2.  $C \rightarrow c$
3.  $C \rightarrow$
4.  $A \rightarrow aBCd$
5.  $A \rightarrow BQ$
6.  $B \rightarrow bB$
7.  $B \rightarrow$
8.  $Q \rightarrow q$

9.  $q \rightarrow$ 

Ideally, we would like an algorithm to parse these things similar to the functions below (`eat(c)` is a function that consumes the next token and requires it to match `c`).

<pre> 1 parseS(): 2   parseA() 3   parseC() 4   eat("\$") 5 parseC(): 6   if (peek() == "x"): 7     eat("x") </pre>	<pre> 1 parseA(): 2   if (peek() == "a"): 3     parseB() 4     parseC() 5     eat("d") 6   else: 7     parseB() 8     parseQ() </pre>
---	---

The potential problem is that there may be multiple choices for rules. To fix this, we talk about 4 things: *SDE*, *RDE*, *First Set*, *Follow Set*

**Definition 3.7 (Symbol Derives Empty)**

A **SDE** is a function that takes in a non-terminal and returns a boolean, indicating true if the non-terminal symbol could derive the empty string.

**Example 3.4**

In the example above,  $B, Q$  can both derive empty since  $B \Rightarrow \epsilon, Q \Rightarrow \epsilon$ . Consequently, since  $A \Rightarrow BQ$ ,  $A$  can also derive empty.  $S$  cannot since it has a "\$."

**Definition 3.8 (Rule Derives Empty)**

An **RDE** is a function that takes in a rule ( $\alpha \rightarrow \beta$ ) and returns a boolean, indicating true if the rule can be used to derive the empty string.

**Theorem 3.1 (Recursive Definition of SDE, RDE)**

Let  $\wedge, \vee$  is the logical AND and OR operations, respectively. Then,

$$\text{SDE}(S) = \bigvee_{S \rightarrow *} \text{RDE}(S \rightarrow *) \quad (2)$$

For nonterminal  $n$  and terminal  $t$ ,

$$\text{RDE}(N \rightarrow *t*) = 0 \quad (3)$$

since  $t$  must be in all future derivations. Given nonterminals  $R_0, \dots, R_n$ , we have

$$\text{RDE}(n \rightarrow R_0 \dots R_n) = \bigwedge_{i=0}^n \text{SDE}(R_i) \quad (4)$$

This is a good definition in itself mathematically, but this has no guarantee of termination. We can implement this fix by using a graph algorithm, keeping track of the visited states, and saying that if a symbol ever derives back to itself, then we return False. A different way is to iterate to a *fixed point*, which is a great algorithmic approach in general.

**Algorithm 3.1 (Iterative Computation of SDE, RDE)**

The general idea is that you have a system of equations and want a solution to it. You start with a solution and keep refreshing it until it doesn't change. Start by assuming no symbols derive empty and no rules deriving empty. You iteratively correct this assumption by looking at the rules, starting with the simplest (e.g.  $C \rightarrow \epsilon$  and  $Q \rightarrow \epsilon$ ).

Basically, this is a 0th order optimization problem in  $\{0, 1\}^N$ . This will terminate since once you switch over to True, you can't go back to false.

**Definition 3.9 (First Set)**

Given nonterminal  $n$  and terminal  $t$ , the `FirstSet` function takes in any string of nonterminals/terminals and returns a set of terminals. More specifically, a terminal  $t'$  is in

$$\text{FirstSet}((n|t)^*) \tag{5}$$

if and only if there exists some derivation that has its first character as  $t'$ .

**Example 3.5**

In the example above, we can compute

$$\text{FirstSet}(aBqD) = \{a\} \tag{6}$$

$$\text{FirstSet}(B) = \{b, q\} \tag{7}$$

$$\text{FirstSet}(BqD) = \{b, q\} \tag{8}$$

**Theorem 3.2 (Recursive Definition of FirstSet)**

We can define

$$\text{FirstSet}() = \{\} \tag{9}$$

$$\text{FirstSet}(t(n|t)^*) = \{t\} \tag{10}$$

$$\text{FirstSet}(n(n|t)^*) = \begin{cases} \bigcup_{n \rightarrow *}\text{FirstSet}(*) & \text{if } \text{SDE}((n|t)^*) = 0 \\ \text{FirstSet}((n|t)^*) \cup \bigcup_{n \rightarrow *}\text{FirstSet}(*) & \text{if } \text{SDE}((n|t)^*) = 1 \end{cases} \tag{11}$$

Note that the  $\bigcup_{n \rightarrow *}\text{FirstSet}(*)$  may infinitely recurse. So to actually compute this, we again do an iterative approach, designed so that it is guaranteed to converge.

**Algorithm 3.2 (Iterative Computation of FirstSet)****Example 3.6**

So after computing this, we can see that a string in this language must start with an  $a$ ,  $b$ ,  $q$ ,  $c$ , or  $\$$ , but not with a  $d$ !

The next function answers the following. Given that I have a nonterminal, what can legally come after it for any valid word in my language?

**Definition 3.10 (Follow Set)**

Let  $G = (V, \Sigma, R, S)$  be a grammar. Then for any nonterminal  $n \in V$ ,  $\text{FollowSet}(n) \subset \Sigma$  is the set of terminals satisfying the following property:  $t \in \text{FollowSet}(n)$  if there exists a some word  $w$  derived from  $S$  that contains a  $n$  immediately followed by a  $t$ . That is,

$$S \implies w, \quad w = \dots nt\dots \tag{12}$$

### 3.6 LR Parsing

Shift (=incrementing) reduce (package into Expr) conflict. For example, `if then ... if then ... else ...`. Is this shift or reduce?

Parser generators allow a compact ambiguous grammar, but they use precedence (and associative) directives, which is an easier way to annotate this rather than writing down 10 more layers of grammar rules. This resolves shift-reduce conflicts.

## 4 Resolution

The scope refers to where a name is visible.

<b>Definition 4.1 (Scope)</b>
The <b>scope</b> of a variable is the region of code where the name is visible.

The resolution is basically a syntactical analysis step where we make sure which value does a given variable refer to? We call this *resolving a variable*.

Later on, we will see this implemented through an **environment**, which is a runtime data structure that stores the name-to-value bindings.

## 5 Type Checking

This isn't needed for dynamically typed languages, but for statically typed ones, we need this. This is *semantic analysis*.

## 6 Tree Walk Interpreters

This is the simplest way to run source code, but it is so slow and memory heavy. Very few modern language implementations use source code interpreters. What is extremely common is that they are compiled to an intermediate form, and then the intermediate form is interpreted in some form of VM.

### 6.1 Evaluation of Expressions

### 6.2 Environments

## 7 Bytecode Interpreters

Java and Python run on a bytecode VM. The process is that after you scan, you take the list of tokens and convert them into bytecode.

### Definition 7.1 (Bytecode)

**Bytecode** is a low-level compact instruction sequence—where each instruction/**opcode** is usually encoded in a byte—produced from source code and consumed by a virtual machine or interpreter.

The fact that each instruction is a byte limits us to a total of 256 instructions. Usually, this is enough but sometimes, we may need to apply variable-length encoding to instructions.

### Example 7.1 (Lox Implementation of Bytecode Chunks)

In the Lox language, a *chunk* is an object that stores a piece of compiled bytecode. The two most important parts are that it contains

1. `ValueArray constants`, which is a dynamically resizable array of heap-allocated elements of type `Value` (which is basically just a double). It serves to contain elements that may be bigger than 1 byte and is most analogous to the read-only data that C uses to store literals.
2. `uint8_t* code`, which is an array of bytes. Each byte can represent either an opcode or the index in `constants` where the literals are stored.

```

1  typedef double Value;
2
3  typedef struct {
4      int capacity;
5      int count;
6      Value* values;
7  } ValueArray;
8
9  typedef struct {
10     int count;      // number of bytes in our array 'code'
11     int capacity;  // number of bytes we have allocated in 'code'
12     uint8_t* code; // bytes of opcode or the index in 'constants'
13     int* lines;    // array keeping track of each byte's line number
14     ValueArray constants; // constant pool containing objects bigger than 1 byte
15 } Chunk;
16

```

Figure 4

Ultimately, the compiler must try to convert.

### Definition 7.2 (Virtual Machine)

A **virtual machine** is a tuple  $(T, ip)$  that contains

1. a pointer to the sequence of tokens generated by the scanner.
2. an **instruction pointer** `ip` that points to the next opcode.<sup>a</sup>

<sup>a</sup>Note that this instruction pointer is specific to the VM, not the actual CPU.

There are two main types of virtual machines: stack-based and register-based. These are based off of the

execution model, not whether the final machine code uses stacks or registers.

## 7.1 Stack-Based Virtual Machines

The key to stack-based virtual machines is the realization that traversing a tree (AST) is really just the same thing as DFS, which uses a stack. Therefore, all the evaluations and executions can be done with a stack.

### Definition 7.3 (Stack-Based Virtual Machine)

A **stack-based virtual machine** is a virtual machine  $(T, ip, S)$  that contains an extra stack  $S$ .

```

1  #define STACK_MAX 256
2
3  typedef struct {
4      Chunk* chunk;
5      uint8_t* ip;
6      Value stack[STACK_MAX];
7      Value* stackTop;
8  } VM;
9

```

Figure 5: C implementation of a stack-based virtual machine.

This allows us to define the relevant opcodes.

### Definition 7.4 (OP\_CONSTANT)

The `OP_CONSTANT` is a bytecode instruction that tells the VM to load a constant value from the chunk's constant pool and push it on to the VM stack.

### Definition 7.5 (Arithmetic)

The arithmetic operators all bytecode instructions that tell the VM to do the same thing.

1. `OP_ADD` says to pop 2 values, add them, and push the result.
2. `OP_SUBTRACT` says to pop 2 values, subtract them, and push the result.
3. `OP_MULTIPLY` says to pop 2 values, multiply them, and push the result.
4. `OP_DIVIDE` says to pop 2 values, divide them, and push the result.

### Definition 7.6 (Negation)

The `OP_NEGATE` is a bytecode instruction that tells the VM to pop 1 value, negate it, and push the result to to stack.

It is often nice to see the state of the stack within each instruction. This is called the **stack trace**.

### Example 7.2 (Adding 2 Numbers)

Instruction	Stack
<code>OP_CONSTANT 0</code>	[1]

```
4 OP_CONSTANT 1 [1, 2]
5 OP_ADD [3]
```

**Definition 7.7 (OP\_RETURN)**

The OP\_RETURN is a bytecode instruction that tells the VM to return the current value(?)

## 7.2 Register-Based Virtual Machines

**Definition 7.8 (Register-Based Virtual Machine)**

A **register-based virtual machine** is a virtual machine  $(T, ip, R)$  that contains an extra set of *registers*.

```
1 #define REGISTER_MAX 256
2
3 typedef struct {
4     Chunk* chunk;
5     uint8_t* ip;
6     Value registers[REGISTER_MAX];
7 } VM;
8
```

Figure 6: C implementation of a register-based virtual machine.

**Example 7.3 (Adding 2 Numbers)**

```
1 LOAD_CONST r0, 0
2 LOAD_CONST r1, 1
3 ADD r2, r0, r1
4 RETURN r2
```