04834580 Software Engineering (Honor Track) 2025-26

# Parsing

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

**Definition ([1])**

Parsing is a process of analyzing a string of symbols, either in programming languages or data structures, conforming to the rules of a formal grammar by breaking it into parts.

# Where Parsing Is Used

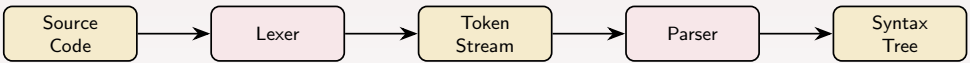


Parsing is fundamental in many software systems:

- **Compilers and interpreters**: translating source code
- **Data formats**: reading JSON, XML, YAML, CSV
- **Configuration files**: INI, TOML, dotfiles
- **Query languages**: SQL, GraphQL, regular expressions
- **Command-line tools**: shells, argument parsers
- **Protocols**: HTTP headers, email (MIME), URLs

# Parsing Pipeline






- **Lexer** (lexical analysis): breaks character stream into tokens
- **Parser** (syntactic analysis): organizes tokens into a tree structure according to grammar rules

The context-free grammar [2] describing arithmetic expressions with addition, subtraction, and parentheses:

$$
\begin{aligned}
E &\to E + T \\
&\mid E - T \\
&\mid T \\
T &\to (E) \\
&\mid \text{id}
\end{aligned}
$$

Here:

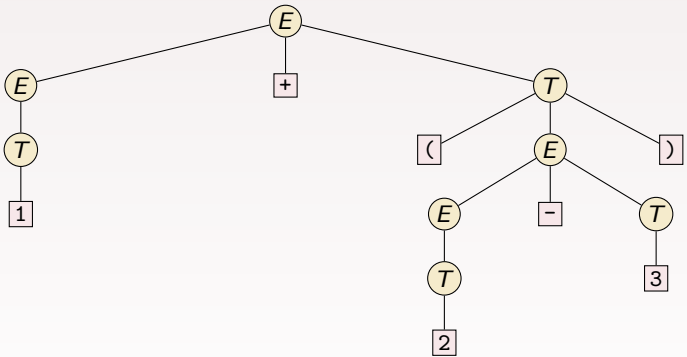
- $E$, $T$: **nonterminals** (syntactic categories)
- $+$, $-$, ( ), id: **terminals** (tokens)
- $E$: the **start symbol**
- Each line is a **production rule**

# Derivation

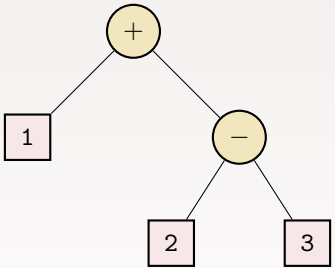
A **derivation** applies production rules to transform the start symbol into a string of terminals. Leftmost derivation of `1 + (2 - 3)`:

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow \mathsf{id} + T \\
&\Rightarrow 1 + T \\
&\Rightarrow 1 + (E) \\
&\Rightarrow 1 + (E - T) \\
&\Rightarrow 1 + (T - T) \\
&\Rightarrow 1 + (2 - T) \\
&\Rightarrow 1 + (2 - 3)
\end{aligned}
$$

Parse tree for `1 + (2 - 3)` records every step of the derivation:



The AST [3] for `1 + (2 - 3)` discards syntactic details (parentheses, intermediate nonterminals):




- Parentheses are gone — precedence is encoded by tree structure
- Internal nodes are operators, leaves are values
- Used by compilers, interpreters, and analysis tools

# Concrete Syntax Tree vs Abstract Syntax Tree

北京大学

**Concrete Syntax Tree (CST)**

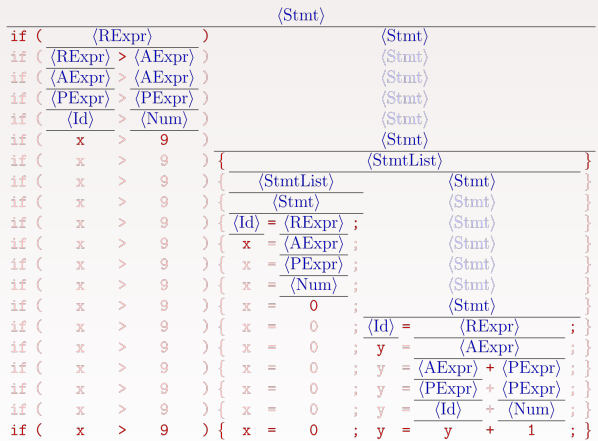
- Preserves all grammar details
- Includes every token (parentheses, keywords, delimiters)
- One node per grammar symbol
- Mirrors the grammar exactly

**Abstract Syntax Tree (AST)**

- Retains only semantic content
- Drops redundant syntactic tokens
- Simplified structure
- Used for downstream processing (compilation, analysis)

# A Fragment of C Grammar

$$
\begin{array}{rcl}
\langle \text{Stmt} \rangle & \to & \langle \text{Id} \rangle \texttt{ = } \langle \text{RExpr} \rangle \texttt{ ;} \\
\langle \text{Stmt} \rangle & \to & \texttt{\{ } \langle \text{StmtList} \rangle \texttt{ \}} \\
\langle \text{Stmt} \rangle & \to & \texttt{if (} \langle \text{RExpr} \rangle \texttt{ )} \langle \text{Stmt} \rangle \\
\langle \text{StmtList} \rangle & \to & \langle \text{Stmt} \rangle \\
\langle \text{StmtList} \rangle & \to & \langle \text{StmtList} \rangle \ \langle \text{Stmt} \rangle \\
\langle \text{RExpr} \rangle & \to & \langle \text{RExpr} \rangle \texttt{ > } \langle \text{AExpr} \rangle \\
\langle \text{RExpr} \rangle & \to & \langle \text{RExpr} \rangle \texttt{ < } \langle \text{AExpr} \rangle \\
\langle \text{RExpr} \rangle & \to & \langle \text{RExpr} \rangle \texttt{ >= } \langle \text{AExpr} \rangle \\
\langle \text{RExpr} \rangle & \to & \langle \text{RExpr} \rangle \texttt{ <= } \langle \text{AExpr} \rangle \\
\langle \text{RExpr} \rangle & \to & \langle \text{AExpr} \rangle \\
\langle \text{AExpr} \rangle & \to & \langle \text{AExpr} \rangle \texttt{ + } \langle \text{PExpr} \rangle \\
\langle \text{AExpr} \rangle & \to & \langle \text{AExpr} \rangle \texttt{ - } \langle \text{PExpr} \rangle \\
\langle \text{AExpr} \rangle & \to & \langle \text{PExpr} \rangle \\
\langle \text{PExpr} \rangle & \to & \langle \text{Id} \rangle \\
\langle \text{PExpr} \rangle & \to & \langle \text{Num} \rangle \\
\langle \text{Id} \rangle & \to & \texttt{x} \\
\langle \text{Id} \rangle & \to & \texttt{y} \\
\langle \text{Num} \rangle & \to & \texttt{0} \\
\langle \text{Num} \rangle & \to & \texttt{1} \\
\langle \text{Num} \rangle & \to & \texttt{9}
\end{array}
$$

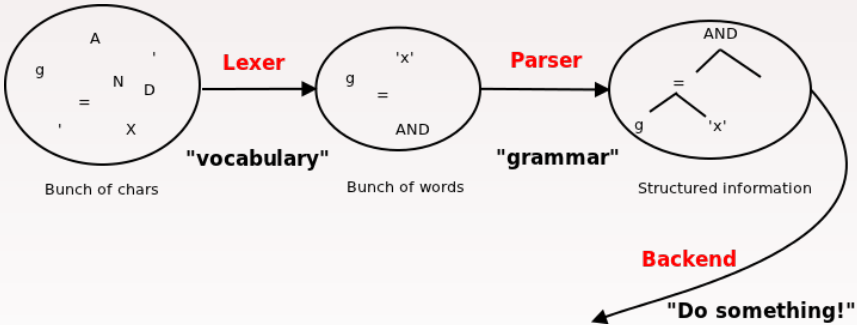


# Backus–Naur Form

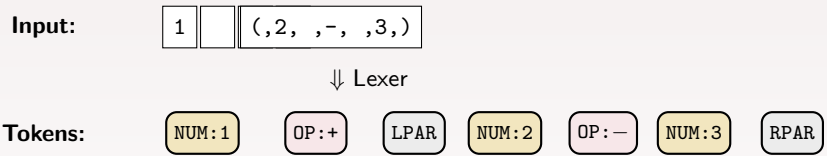
The Backus–Naur Form (BNF) [4] is a notation for defining programming language syntax:

```
E ::= T (('+' | '-') T)*
T ::= '(' E ')' | id
```

- `::=` means "is defined as"
- `|` means "or" (alternative)
- `*` means "zero or more" (repetition)
- `'...'` denotes terminal symbols (literal characters)
- Unquoted names are nonterminals


A
'
g
N
D
=
'
X
Lexer
"vocabulary"
Bunch of chars
'x'
g
=
AND
Bunch of words
Parser
"grammar"
AND
=
g
'x'
Structured information
Backend
"Do something!"

The lexer converts a character stream into a sequence of tokens:




- Whitespace is consumed but not emitted as tokens
- Each token has a **type** (NUM, OP, LPAR, ...) and a **value**

# Lexer Example



A lexer represents a stream of tokens:

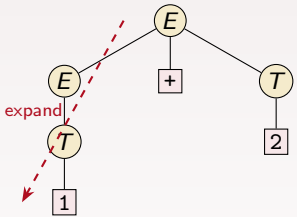
```
import re

class Lexer:
    def __init__(self, input_text):
        self.tokens = re.findall(r'\d+|[()+\-]', input_text)
        self.position = 0

    def get_next_token(self):
        if self.position < len(self.tokens):
            token = self.tokens[self.position]
            self.position += 1
            return token
        return None

    def peek(self):
        if self.position < len(self.tokens):
            return self.tokens[self.position]
        return None
```

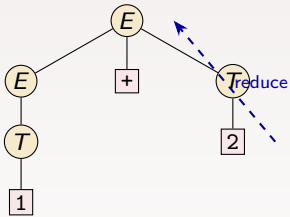
### Top-Down Parsing




Start from root (start symbol), expand nonterminals to match input

Example: recursive descent

### Bottom-Up Parsing




Start from input tokens, reduce to start symbol

Example: shift-reduce (LR)

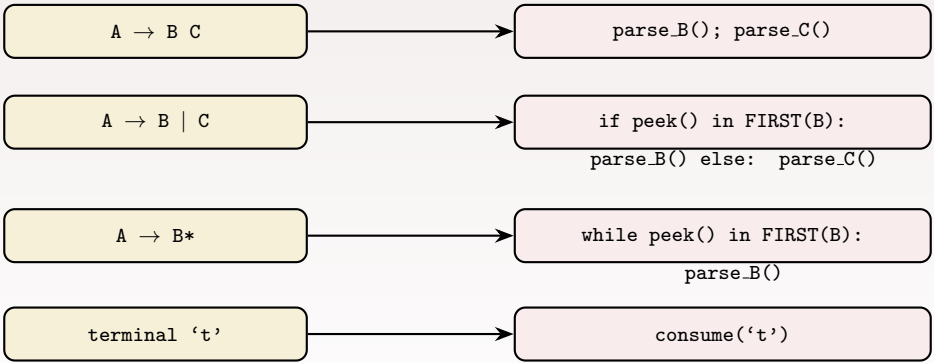
**Definition ([5])**

A kind of top-down parser built from a set of mutually recursive procedures where each such procedure implements one of the nonterminals of the grammar.

Key idea: one function per nonterminal

- `parse_E()` implements the production rules for $E$
- `parse_T()` implements the production rules for $T$
- Functions call each other following the grammar structure

# Grammar Rules to Code



# Recursive Descent Parser Example

北京大学

```
class Parser:
    def __init__(self, lexer):
        self.lexer = lexer
        self.current_token = self.lexer.get_next_token()

    def consume(self):
        self.current_token = self.lexer.get_next_token()

    def parse_E(self):
        node = self.parse_T()
        while self.current_token in ('+', '-'):
            op = self.current_token
            self.consume()
            node = (op, node, self.parse_T())
        return node

    ...
```

# Recursive Descent Parser Example (Continued)

北京大学

```
...

def parse_T(self):
    if self.current_token == '(':
        self.consume()  # Consume '('
        node = self.parse_E()
        if self.current_token == ')':
            self.consume()  # Consume ')'
            return node
        else:
            raise SyntaxError("Missing closing parenthesis")
    elif self.current_token.isdigit():
        node = int(self.current_token)  # Convert id (number) to integer
        self.consume()
        return node
    else:
        raise SyntaxError(f"Unexpected token: {self.current_token}")
```

# Tracing: Parsing `1 + (2 - 3)`

北京大学

| Step | Call | Token | Action |
|---|---|---|---|
| 1 | `parse_E()` | `1` | calls `parse_T()` |
| 2 | `  parse_T()` | `1` | consume `1`, return `1` |
| 3 | `parse_E()` | `+` | matches `+`, consume |
| 4 | `  parse_T()` | `(` | consume `(`, calls `parse_E()` |
| 5 | `    parse_E()` | `2` | calls `parse_T()` |
| 6 | `      parse_T()` | `2` | consume `2`, return `2` |
| 7 | `    parse_E()` | `-` | matches `-`, consume |
| 8 | `      parse_T()` | `3` | consume `3`, return `3` |
| 9 | `    parse_E()` | `)` | no more ops, return `('-', 2, 3)` |
| 10 | `  parse_T()` | `)` | consume `)`, return `('-', 2, 3)` |
| 11 | `parse_E()` | — | return `('+', 1, ('-', 2, 3))` |

# Recursive Descent Parser Example

北京大学

Client code:

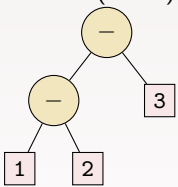
```
if __name__ == "__main__":
    input_text = "1 + (2 - 3)"
    lexer = Lexer(input_text)
    parser = Parser(lexer)
    syntax_tree = parser.parse_E()
    print(syntax_tree)
```

Output:

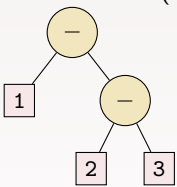
```
('+', 1, ('-', 2, 3))
```

A grammar is **ambiguous** if a string has more than one parse tree. Consider the expression `1 - 2 - 3` with the grammar $E \to E - T \mid T$:

**Left-associative**: $(1 - 2) - 3 = -4$




**Right-associative**: $1 - (2 - 3) = 2$




Same expression, different results — ambiguity must be resolved!

Ambiguity is resolved by encoding **precedence** and **associativity** into the grammar:

**Ambiguous:**

$$E \to E + E \mid E * E \mid \text{id}$$

Does 1 + 2 * 3 mean $(1 + 2) * 3$ or $1 + (2 * 3)$?

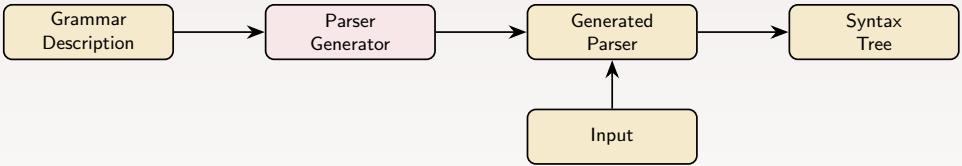
**Unambiguous:**

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to \text{id} \mid (E)$$

Separate nonterminals enforce $*$ binds tighter than $+$.

More nonterminal levels = more precedence levels

Parser generators automatically produce parsers from grammar descriptions [6]:




Popular parser generators:

- **Bison** for C/C++
- **ANTLR** for Java and other languages
- **Lark** for Python

# ANTLR Notation

北京大学

```
grammar ExpressionGrammar;

// parser

expr  : left=expr op=('*'|'/') right=expr #opExpr
      | left=expr op=('+'|'-') right=expr #opExpr
      | '(' expr ')'                      #parenExpr
      | atom=INT                          #atomExpr
      ;

// lexer

INT : ('0' .. '9') +;

WS : [ \r\n\t] + -> skip ;
```

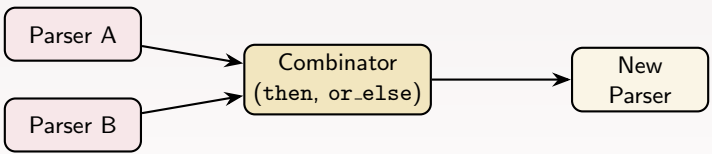
# Lark Notation



```
?start: product
    | start "+" product -> add
    | start "-" product -> sub
?product: atom
    | product "*" atom  -> mul
    | product "/" atom  -> div
?atom: NUMBER           -> number
    | "-" atom          -> neg
    | "(" start ")"
%import common.NUMBER
%import common.WS_INLINE
%ignore WS_INLINE
```

**Definition ([7])**

A parser combinator is a higher-order function that accepts several parsers as input and returns a new parser as its output.




Build complex parsers by composing simple ones — no grammar file needed.

# Parser Combinator Primitives



A parser is a function: takes a string, returns `(result, remaining)` or `None`:

```
def char(c):
    """Parse a single expected character."""
    def parser(s):
        if s and s[0] == c:
            return (c, s[1:])
        return None
    return parser

def digit(s):
    """Parse a single digit."""
    if s and s[0].isdigit():
        return (s[0], s[1:])
    return None
```

Usage: `char('+')("+-3")` returns `('+', '-3')`

# Combining Parsers



Higher-order functions compose parsers into larger ones:

```
def then(p1, p2):
    """Sequence: match p1 then p2."""
    def parser(s):
        r1 = p1(s)
        if r1:
            r2 = p2(r1[1])
            if r2:
                return ((r1[0], r2[0]), r2[1])
        return None
    return parser

def or_else(p1, p2):
    """Alternative: try p1, fall back to p2."""
    def parser(s):
        return p1(s) or p2(s)
    return parser
```

Composition: `then(digit, then(char('+'), digit))` parses strings like `"1+2"`

# References I


[1] Wikipedia Authors.
Parsing.
https://en.wikipedia.org/wiki/Parsing, 2025.

[2] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman.
Introduction to automata theory, languages, and computation.
*Acm Sigact News*, 32(1):60–65, 2001.

[3] Wikipedia Authors.
Abstract syntax tree.
https://en.wikipedia.org/wiki/Abstract_syntax_tree, 2025.

[4] John W Backus.
The syntax and the semantics of the proposed international algebraic language of the zurich acm-gamm conference.
In *ICIP Proceedings*, pages 125–132, 1959.

# References II

[5] Wikipedia Authors.
Recursive descent parser.
https://en.wikipedia.org/wiki/Recursive_descent_parser, 2025.

[6] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman.
*Compilers: Principles, Techniques, and Tools*.
Addison-Wesley, 2nd edition, 2006.

[7] Wikipedia Authors.
Parser combinator.
https://en.wikipedia.org/wiki/Parser_combinator, 2025.