04834580 Software Engineering (Honor Track) 2024-25

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.

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

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

Here:

- E: Represents an expression.
- T: Represents a term.
- id: Represents an identifier, which can be a number or variable.
- The operators $+$ and $-$ denote addition and subtraction respectively.
- Parentheses (E) group sub-expressions for precedence.

A Fragment of C Grammar

北京大学

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

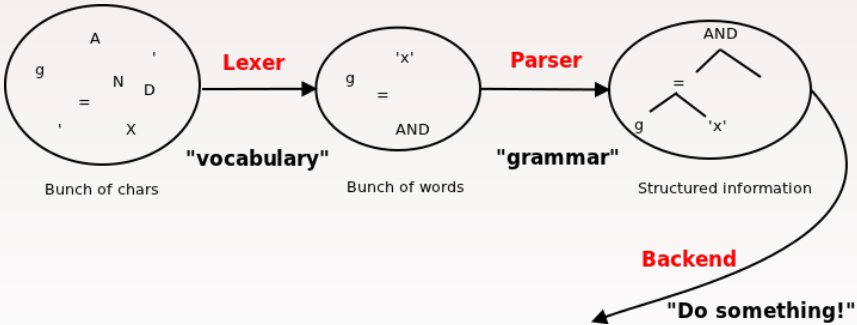
$$
\begin{array}{l}
\langle \text{Stmt} \rangle \\
\texttt{if (} \langle \text{RExpr} \rangle \texttt{)} \quad \langle \text{Stmt} \rangle \\
\texttt{if (} \langle \text{RExpr} \rangle \texttt{ > } \langle \text{AExpr} \rangle \texttt{)} \quad \langle \text{Stmt} \rangle \\
\texttt{if (} \langle \text{AExpr}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

Backus–Naur Form

The Backus–Naur Form (BNF) [3] is a notation system for defining the syntax of programming languages:

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

- `E ::= T (('+' | '-') T)` an expression E consists of a term T optionally followed by zero or more sequences of addition "+" or subtraction "-" operators, each paired with another term T. The "*" indicates repetition (zero or more occurrences).
- `T ::= '(' E ')' | id` means a term T can either be an expression E enclosed in parentheses to group subexpressions (E), or an identifier id, which represents variables or literals like numbers.


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!"

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
```

Definition ([4])

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.

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}")
```

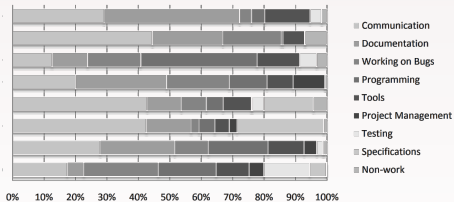
Client code:

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

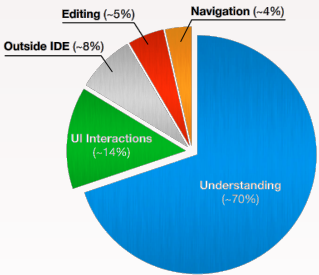
How Developers Spend Their Time

北京大学

2008's study with 8 new Microsoft developers showed that programming occupies only 10–20% of their time. [5]



2015's study with 18 developers showed that they spend 70% of their time on understanding. [6]



Parser Generator

Parser generators such as generate parsers based on grammar descriptions such as BNF. Popular generations include:

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

ANTRL Notation

```
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

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

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] 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.

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

References II

[5] Andrew Begel and Beth Simon.
Novice software developers, all over again.
In *International Workshop on Computing Education Research*, pages 3–14, 2008.

[6] Roberto Minelli, Andrea Mocci, and Michele Lanza.
I know what you did last summer-an investigation of how developers spend their time.
In *International Conference on Program Comprehension*, pages 25–35. IEEE, 2015.