CP471 - Intro to Compiling
Phase #1

Topic: Lexical Analysis

Aidan Traboulay

Wednesday 15th February, 2023

1 Introduction

The initial stage of a compiler is called lexical analysis, and it has a fundamental role in converting a high-level programming language into machine language that can be understood by computers. This stage involves examining the program's source code and breaking it down into distinct "tokens," which are the fundamental components of the programming language. These tokens are then evaluated and sorted according to their classification, including keywords, identifiers, operators, and literals. The primary objective of this phase is to transform the original source code into a well-organized set of tokens that can be effortlessly handled by the following phase of the compiler, which is the parser.

2 Grammar

```
<program> ::= <fdecls> <declarations> <statement_seq>.
<fdecls> ::= <fdec>; | <fdecls> <fdec>; |
<fdec> ::= def <type> <fname> ( <params> ) <declarations> <statement_seq> fed
<params> ::= <type> <var> | <type> <var> , <params> |
<fname> ::= <id>
<declarations> ::= <decl>; | <declarations> <decl>; |
<decl> := <type> <varlist>
<type> := int | double
<varlist> ::= <var>, <varlist> | <var>
<statement_seq> ::= <statement> | <statement>; <statement_seq>
<statement> ::= <var> = <expr> |
if <bexpr> then <statement_seq> fi |
if <bexpr> then <statement_seq> else <statement_seq> fi |
while <bexpr> do <statement_seq> od |
print <expr> |
return <expr> |
<expr> ::= <expr> + <term> | <expr> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <term> % <factor> |
<factor>
<factor> ::= <var> | <number> | (<expr>) | <fname>(<exprseq>)
<exprseq> ::= <expr>, <exprseq> | <expr> |
<bexpr> ::= <bexpr> or <bterm> | <bterm>
<bterm> ::= <bterm> and <bfactor> | <bfactor>
<bfactor> ::= (<bexpr>) | not <bfactor> | (<expr> <comp> <expr>)
<comp> ::= < | > | == | <= | >= | <>
<var> ::= <id> | <id>[<expr>]
<letter> ::= a | b | c | ... | z
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
<id> ::= <letter> | <id><letter> | <id><digit>
<number> ::= <integer> | <double>...
```

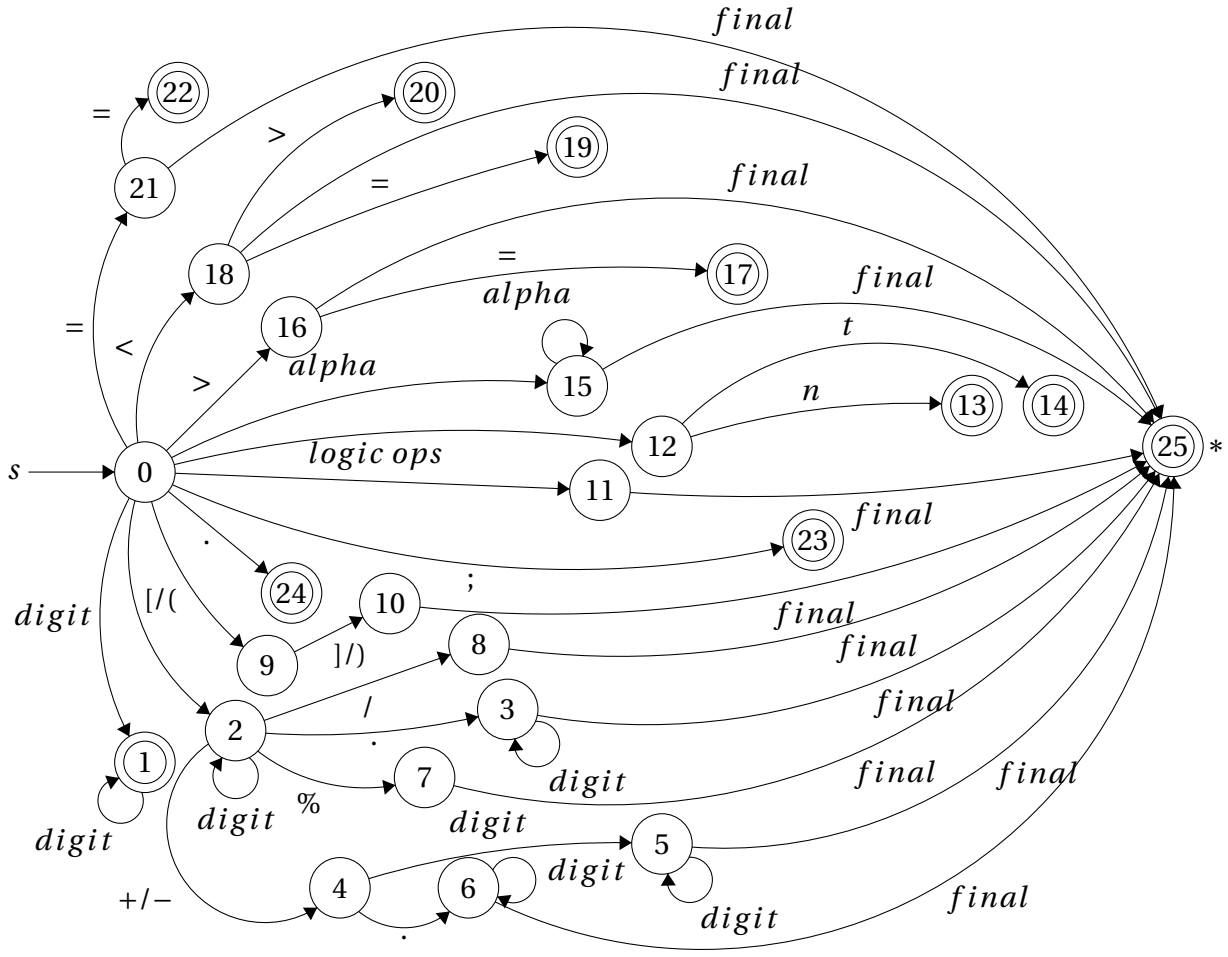
3 Terminals & Non-Terminals

Non-Terminals
<program>
<fdecls>
<fdec>
<params>
<fname>
<declarations>
<decl>
<type>
<varlist>
<statement_seq>
<statement>
<expr>
<term>
<factor>
<exprseq>
<bexpr>
<bterm>
<bfactor>
<comp>
<var>
<letter>
<digit>
<id>
<number>

Terminals
def
fed
(
)
,
int
double
=
if
then
else
while
do
od
print
return
/
%
or
and
not
<
==
<=
=
<>
[
]
a, b, c, ..., z (lowercase)
1, 2, 3, 4, 5, 6, 7, 8, 9, 0 (digits)

4 Deterministic Finite Automaton

4.1 State Diagram



4.2 Transition Table

State	0...9	a...z	=	<	>	\	[	]	(	)	/	.	%	+	-	;
0	1	15	21	18	16	12	9	10	9	10	3	6	7	4	4	23*
1	1*	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	2	-	-	-	-	-	-	-	-	-	8	3	-	-	-	-
3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	5	-	-	-	-	-	-	-	-	-	-	6	-	-	-	-
5	5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	10	10	10	10	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	13, 14	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	-	15	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	-	-	17	-	-	-	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
18	-	-	19	20	-	-	-	-	-	-	-	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
21	-	-	22	-	-	-	-	-	-	-	-	-	-	-	-	-
22	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
23	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

5 Implementation

```
use std::fs::File;
use std::io::prelude::*;
use std::fmt;
use comfy_table::Table;

/*
    @Description: Enum of all possible tokens
    @Params: None
    @Returns: None
*/
#[derive(Debug, PartialEq)]
pub enum TokenTypes {
    Def,
    Type(String),
    Ident(String),
    LParen,
    RParen,
    Comma,
    Semicolon,
    Assign,
    Plus,
    PlusAssign,
    Minus,
    MinusEqual,
    Asterisk,
    AsteriskEqual,
    Divide,
    DivideEqual,
    Modulo,
    ModuloEqual,
    If,
    Then,
    Else,
    Fi,
    While,
    Do,
    Od,
    Print,
    Return,
    Eof,
```

```
    IntegerLiteral(i32),
    DoubleLiteral(f64),
    Or,
    And,
    Not,
    Less,
    Greater,
    Equal,
    LessEqual,
    GreaterEqual,
    NotEqual,
    LBracket,
    RBracket,
    Error,
}

/*
    @Description: String representation of the token types
    @Params: None
    @Returns: None
*/
impl fmt::Display for TokenTypes {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            TokenTypes::Def => write!(f, "Def"),
            TokenTypes::Type(ref s) => write!(f, "Type({})", s),
            TokenTypes::Ident(ref s) => write!(f, "Ident({})", s),
            TokenTypes::LParen => write!(f, "LParen"),
            TokenTypes::RParen => write!(f, "RParen"),
            TokenTypes::Comma => write!(f, "Comma"),
            TokenTypes::Semicolon => write!(f, "Semicolon"),
            TokenTypes::Assign => write!(f, "Assign"),
            TokenTypes::Plus => write!(f, "Plus"),
            TokenTypes::PlusAssign => write!(f, "PlusAssign"),
            TokenTypes::Minus => write!(f, "Minus"),
            TokenTypes::MinusEqual => write!(f, "MinusEqual"),
            TokenTypes::Asterisk => write!(f, "Asterisk"),
            TokenTypes::AsteriskEqual => write!(f, "AsteriskEqual"),
            TokenTypes::Divide => write!(f, "Divide"),
            TokenTypes::DivideEqual => write!(f, "DivideEqual"),
            TokenTypes::Modulo => write!(f, "Modulo"),
            TokenTypes::ModuloEqual => write!(f, "ModuloEqual"),
            TokenTypes::If => write!(f, "If"),
```

```
                TokenTypes::Then => write!(f, "Then"),
                TokenTypes::Else => write!(f, "Else"),
                TokenTypes::Fi => write!(f, "Fi"),
                TokenTypes::While => write!(f, "While"),
                TokenTypes::Do => write!(f, "Do"),
                TokenTypes::Od => write!(f, "Od"),
                TokenTypes::Print => write!(f, "Print"),
                TokenTypes::Return => write!(f, "Return"),
                TokenTypes::Eof => write!(f, "Eof"),
                TokenTypes::IntegerLiteral(ref i) => write!(f, "IntegerLiteral({})",
                    i),
                TokenTypes::DoubleLiteral(ref d) => write!(f, "DoubleLiteral({})",
                    d),
                TokenTypes::Or => write!(f, "Or"),
                TokenTypes::And => write!(f, "And"),
                TokenTypes::Not => write!(f, "Not"),
                TokenTypes::Less => write!(f, "Less"),
                TokenTypes::Greater => write!(f, "Greater"),
                TokenTypes::Equal => write!(f, "Equal"),
                TokenTypes::LessEqual => write!(f, "LessEqual"),
                TokenTypes::GreaterEqual => write!(f, "GreaterEqual"),
                TokenTypes::NotEqual => write!(f, "NotEqual"),
                TokenTypes::LBracket => write!(f, "LBracket"),
                TokenTypes::RBracket => write!(f, "RBracket"),
                TokenTypes::Error => write!(f, "Error"),
            }
        }
    }

    /*
        @Description: Struct for tokens
        @Params: None
        @Returns: None
    */
    #[derive(Debug)]
    pub struct Token {
        pub token_type: TokenTypes,
        pub line_number: usize,
        pub column_number: usize,
        pub lexeme: String,
    }

    /*
```

```
    @Description: Lexical Analyzer method
    @Param: input - String
    @Return: Token on different line with line number
*/
pub fn get_next_token(input: &str) -> Result<Vec<Token>, String> {
    let mut tokens = Vec::new();
    let mut chars = input.chars().peekable();
    let mut line_number = 1;
    let mut column_number = 0;

    while let Some(c) = chars.next() {
        if !(c == ' ' || c == '\t' || c == '\r') {
            column_number += 1;

            if c == '\n' {
                column_number = 0;
            }
        }

        match c {
            '\n' => line_number += 1,
            ' ' | '\t' | '\r' => continue,
            'a'..='z' | 'A'..='Z' => {
                let mut ident = String::new();
                ident.push(c);
                while let Some(&c) = chars.peek() {
                    match c {
                        'a'..='z' | 'A'..='Z' | '0'..='9' => {
                            ident.push(c);
                            chars.next();
                        }
                        _ => break,
                    }
                }

                let token_type = match ident.as_str() {
                    "def" => TokenTypes::Def,
                    "type" => TokenTypes::Type(ident.clone()),
                    "if" => TokenTypes::If,
                    "then" => TokenTypes::Then,
                    "else" => TokenTypes::Else,
                    "fi" => TokenTypes::Fi,
                    "while" => TokenTypes::While,
```

```
                "do" => TokenTypes::Do,
                "od" => TokenTypes::Od,
                "print" => TokenTypes::Print,
                "return" => TokenTypes::Return,
                "or" => TokenTypes::Or,
                "and" => TokenTypes::And,
                "not" => TokenTypes::Not,
                "int" => TokenTypes::Type(ident.clone()),
                "double" => TokenTypes::Type(ident.clone()),
                "bool" => TokenTypes::Type(ident.clone()),
                "string" => TokenTypes::Type(ident.clone()),
                "void" => TokenTypes::Type(ident.clone()),
                _ => TokenTypes::Ident(ident.clone()),
            };

            tokens.push(Token {
                token_type,
                line_number,
                column_number,
                lexeme: ident.as_str().to_string(),
            });
        }

        '0'..='9' => {
            let mut number = String::new();
            number.push(c);
            while let Some(&c) = chars.peek() {
                match c {
                    '0'..='9' => {
                        number.push(c);
                        chars.next();
                    }
                    _ => break,
                }
            }

            if let Some(&'.') = chars.peek() {
                number.push('.');
                chars.next();
                while let Some(&c) = chars.peek() {
                    match c {
                        '0'..='9' => {
                            number.push(c);
```

```
                        chars.next();
                    }
                    _ => break,
                }
            }
            tokens.push(Token {
                token_type: TokenTypes::DoubleLiteral(
                    number.parse().expect("Unable to parse double"),
                ),
                line_number,
                column_number,
                lexeme: number.clone(),
            });
        } else {
            tokens.push(Token {
                token_type: TokenTypes::IntegerLiteral(
                    number.parse().expect("Unable to parse integer"),
                ),
                line_number,
                column_number,
                lexeme: number.clone(),
            });
        }

        if let Some(&c) = chars.peek() {
            if c.is_alphabetic() {
                tokens.push(Token {
                    token_type: TokenTypes::Error,
                    line_number,
                    column_number: column_number + 1,
                    lexeme: String::from("".to_owned() +
                        &c.to_string()),
                });
            }
        }
    }

    '(' => tokens.push(Token {
        token_type: TokenTypes::LParen,
        line_number,
        column_number,
        lexeme: String::from("("),
    }),
```

```
')' => tokens.push(Token {
    token_type: TokenTypes::RParen,
    line_number,
    column_number,
    lexeme: String::from(")"),
}),

'[' => tokens.push(Token {
    token_type: TokenTypes::LBracket,
    line_number,
    column_number,
    lexeme: String::from("["),
}),

']' => tokens.push(Token {
    token_type: TokenTypes::RBracket,
    line_number,
    column_number,
    lexeme: String::from("]"),
}),

',' => tokens.push(Token {
    token_type: TokenTypes::Comma,
    line_number,
    column_number,
    lexeme: String::from(","),
}),

';' => tokens.push(Token {
    token_type: TokenTypes::Semicolon,
    line_number,
    column_number,
    lexeme: String::from(";"),
}),

'=' => {
    if let Some(&'=') = chars.peek() {
        chars.next();
        tokens.push(Token {
            token_type: TokenTypes::Equal,
            line_number,
            column_number,
```

```
                    lexeme: String::from("=="),
               });
          } else {
               tokens.push(Token {
                    token_type: TokenTypes::Assign,
                    line_number,
                    column_number,
                    lexeme: String::from("="),
               });
          }
     },

     '+' => {
          if let Some(&'=') = chars.peek() {
               chars.next();
               tokens.push(Token {
                    token_type: TokenTypes::PlusAssign,
                    line_number,
                    column_number,
                    lexeme: String::from("+="),
               });
          } else {
               tokens.push(Token {
                    token_type: TokenTypes::Plus,
                    line_number,
                    column_number,
                    lexeme: String::from("+"),
               });
          }
     }

     '-' => {
          if let Some(&'=') = chars.peek() {
               chars.next();
               tokens.push(Token {
                    token_type: TokenTypes::MinusEqual,
                    line_number,
                    column_number,
                    lexeme: String::from("-="),
               });
          } else {
               tokens.push(Token {
                    token_type: TokenTypes::Minus,
```

```
                    line_number,
                    column_number,
                    lexeme: String::from("-"),
               });
          }
     }

     '*' => {
          if let Some(&'=') = chars.peek() {
               chars.next();
               tokens.push(Token {
                    token_type: TokenTypes::AsteriskEqual,
                    line_number,
                    column_number,
                    lexeme: String::from("*="),
               });
          } else {
               tokens.push(Token {
                    token_type: TokenTypes::Asterisk,
                    line_number,
                    column_number,
                    lexeme: String::from("*"),
               });
          }
     }

     '/' => {
          if let Some(&'=') = chars.peek() {
               chars.next();
               tokens.push(Token {
                    token_type: TokenTypes::DivideEqual,
                    line_number,
                    column_number,
                    lexeme: String::from("/="),
               });
          } else {
               tokens.push(Token {
                    token_type: TokenTypes::Divide,
                    line_number,
                    column_number,
                    lexeme: String::from("/"),
               });
          }
```

```
}

'%' => {
    if let Some(&'=') = chars.peek() {
        chars.next();
        tokens.push(Token {
            token_type: TokenTypes::ModuloEqual,
            line_number,
            column_number,
            lexeme: String::from("%="),
        });
    } else {
        tokens.push(Token {
            token_type: TokenTypes::Modulo,
            line_number,
            column_number,
            lexeme: String::from("%"),
        });
    }
}

'<' => {
    if let Some(&'=') = chars.peek() {
        chars.next();
        tokens.push(Token {
            token_type: TokenTypes::LessEqual,
            line_number,
            column_number,
            lexeme: String::from("<="),
        });
    }

    if let Some(&'>') = chars.peek() {
        chars.next();
        tokens.push(Token {
            token_type: TokenTypes::NotEqual,
            line_number,
            column_number,
            lexeme: String::from("<>"),
        });
    } else {
        tokens.push(Token {
            token_type: TokenTypes::Less,
```

```
                    line_number,
                    column_number,
                    lexeme: String::from("<"),
               });
          }
     }

     '>' => {
          if let Some(&'=') = chars.peek() {
               chars.next();
               tokens.push(Token {
                    token_type: TokenTypes::GreaterEqual,
                    line_number,
                    column_number,
                    lexeme: String::from(">="),
               });
          } else {
               tokens.push(Token {
                    token_type: TokenTypes::Greater,
                    line_number,
                    column_number,
                    lexeme: String::from(">"),
               });
          }
     }

     '!' => {
          if let Some(&'=') = chars.peek() {
               chars.next();
               tokens.push(Token {
                    token_type: TokenTypes::NotEqual,
                    line_number,
                    column_number,
                    lexeme: String::from("!="),

               });
          } else {
               tokens.push(Token {
                    token_type: TokenTypes::Not,
                    line_number,
                    column_number,
                    lexeme: String::from("!"),
               });
```

```
                    }
                }

                '.' => tokens.push(Token {
                    token_type: TokenTypes::Eof,
                    line_number,
                    column_number,
                    lexeme: String::from("."),
                }),

                _ => {
                    tokens.push(Token {
                        token_type: TokenTypes::Error,
                        line_number,
                        column_number,
                        lexeme: String::from("".to_owned() + &c.to_string()),
                    });
                }
            }
        }
    }
    Ok(tokens)
}

/*
    @Description: Runs the get_next_token function on two buffers of text and
        writes to file
    @Params: file_name: String
    @Returns: None
*/
pub fn run_lexical_analysis(file_name: String) {
    let mut error_file = File::create("data/error.log").expect("Unable to create
        file");
    let mut valid_file = File::create("data/valid.log").expect("Unable to create
        file");

    let mut valid_table = Table::new();
    let mut error_table = Table::new();

    let mut buffer1 = String::new();
    let mut buffer2 = String::new();
    let mut file = File::open(file_name).expect("Unable to open file");

    file.read_to_string(&mut buffer2).expect("Unable to read file");
```

```
    buffer1.push_str(&buffer2);

    valid_table.set_header(vec!["Token Type", "Line Number", "Column Number",
        "Lexeme"]);
    error_table.set_header(vec!["Token Type", "Line Number", "Column Number",
        "Lexeme"]);

    match get_next_token(&buffer1) {
        Ok(tokens) => {
            for token in tokens {
                if token.token_type == TokenTypes::Error {
                    error_table.add_row(vec![
                        token.token_type.to_string(),
                        token.line_number.to_string(),
                        token.column_number.to_string(),
                        token.lexeme.to_string(),
                    ]);
                }

                else {
                    valid_table.add_row(vec![
                        token.token_type.to_string(),
                        token.line_number.to_string(),
                        token.column_number.to_string(),
                        token.lexeme.to_string(),
                    ]);
                }
            }
        }
        Err(e) => {
            writeln!(error_file, "{}", e).expect("Unable to write to file");
        }
    }

    // let tokens = get_next_token(&buffer1);
    // println!("{:?}", tokens.unwrap());
    writeln!(valid_file, "{}", valid_table).expect("Unable to write to file");
    writeln!(error_file, "{}", error_table).expect("Unable to write to file");
}
```

6 Analysis

The lexer was implemented in Rust for a numerous amount of reasons. Firstly, the language has an extremely built out robust type system. This allows ease of manipulation and linking of tokens to their given lexemes; this is showcased in the use with the definition of `TokenTypes`, a set of values, stored in an enum. Futuremore t hese values are then linked to the `Token` struct. Another important use case of Rust was its memory safety, it guarantees prevention of a buffer overflow, which can easily occur when passing in the target language to the lexer. It does this through its ownership and borrowing system, which enfores the safety guarantees at compile-time without compromising on performance - it took $0.15s$ to build and compile. The biggest use case in this implementation was the pattern matching feature, allowing ease of use to match input from the grammar to its desired token, which is of particular importance in building the lexer.

References

[1] CS143 Lecture 3 - Stanford University. (n.d.). Retrieved February 24, 2023, from *https://web.stanford.edu/class/cs143/lectures/lecture03.pdf*

[2] Aho, A. V. (2008). Compilers : Principles, Techniques, Tools. Greg Tobin.