Tecnológico de Monterrey

Campus Monterrey

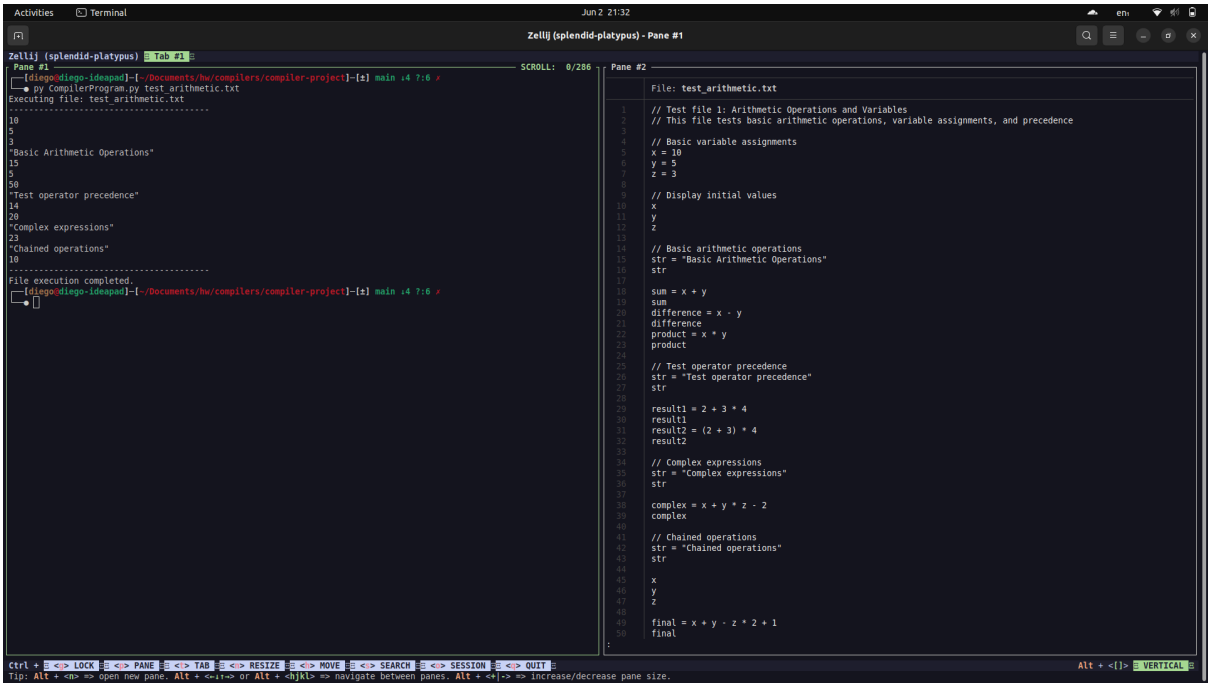
Development of advanced computer science applications

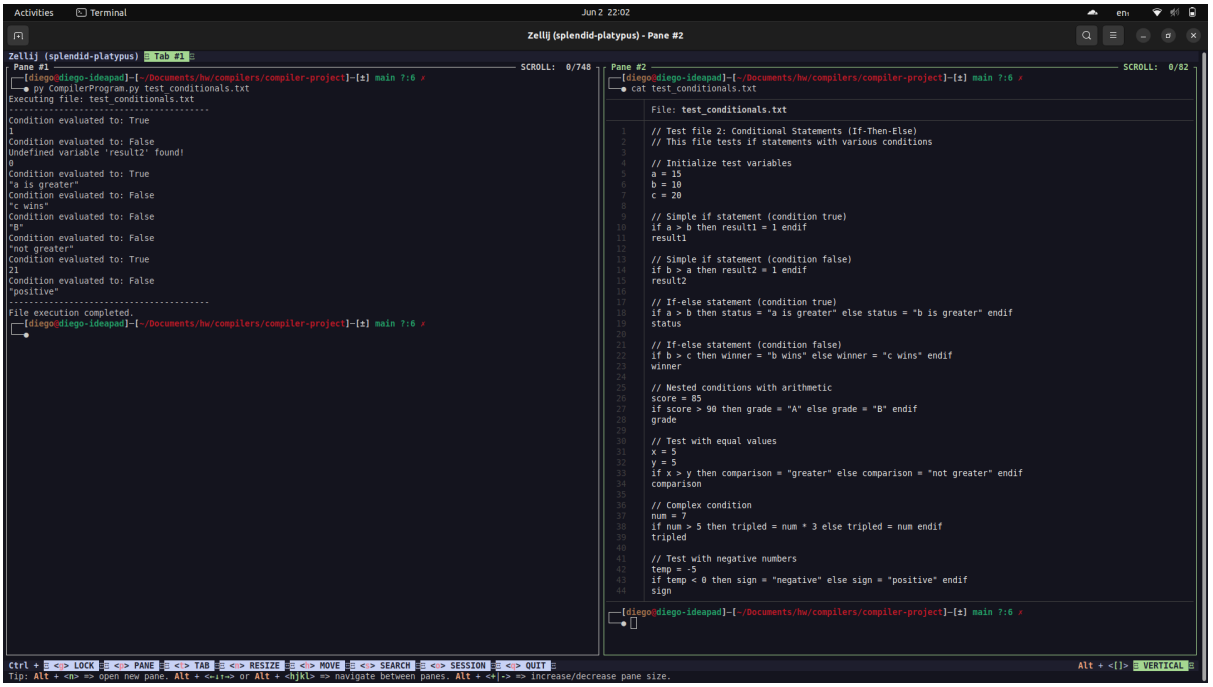
Compiler

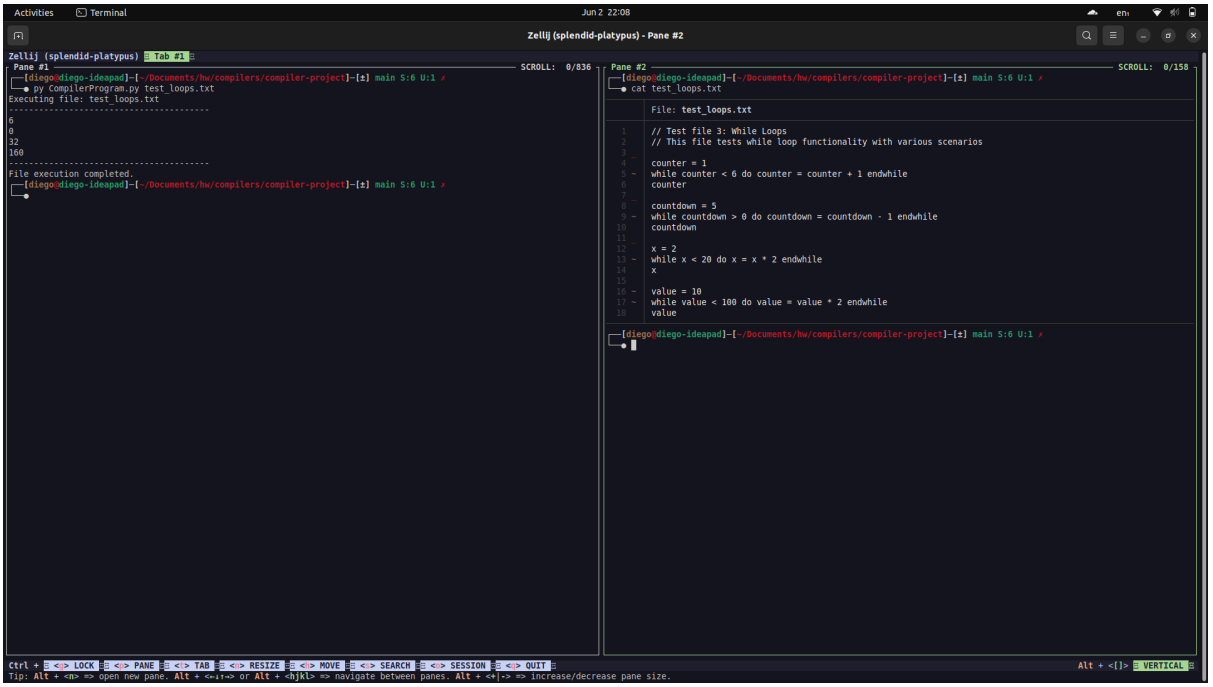
A00833623 Iván Alberto Romero Wells
A00834015 Diego Eduardo Hernández Cadena

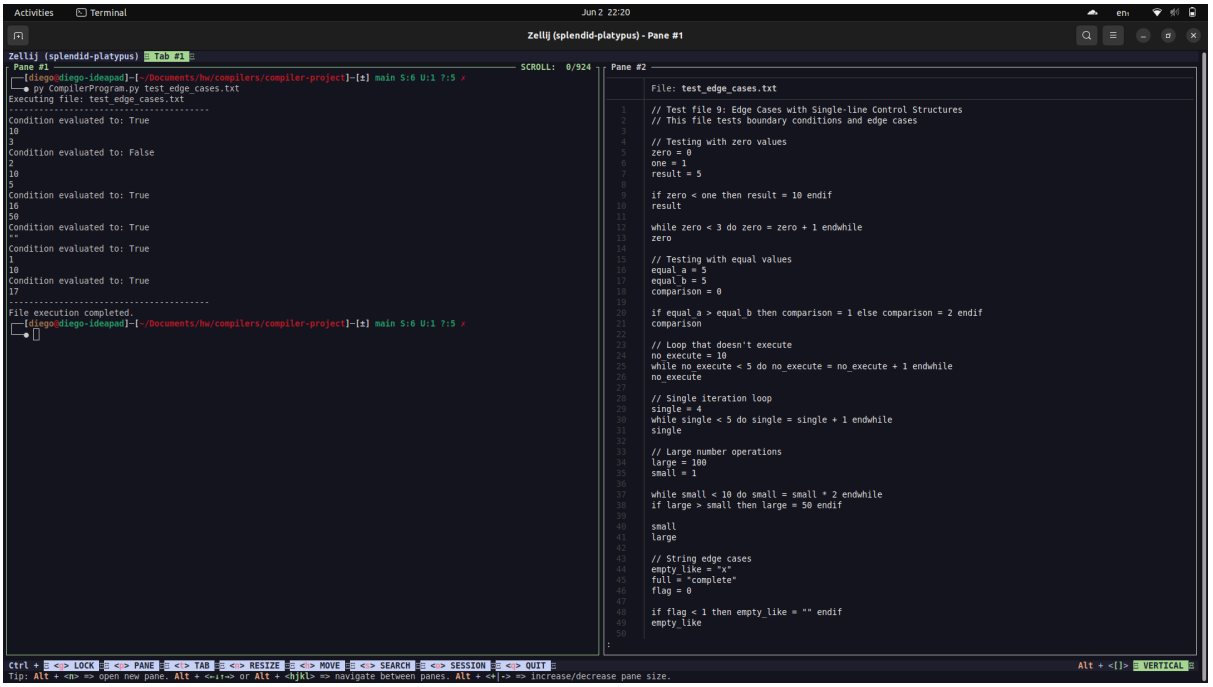
June 2, 2025

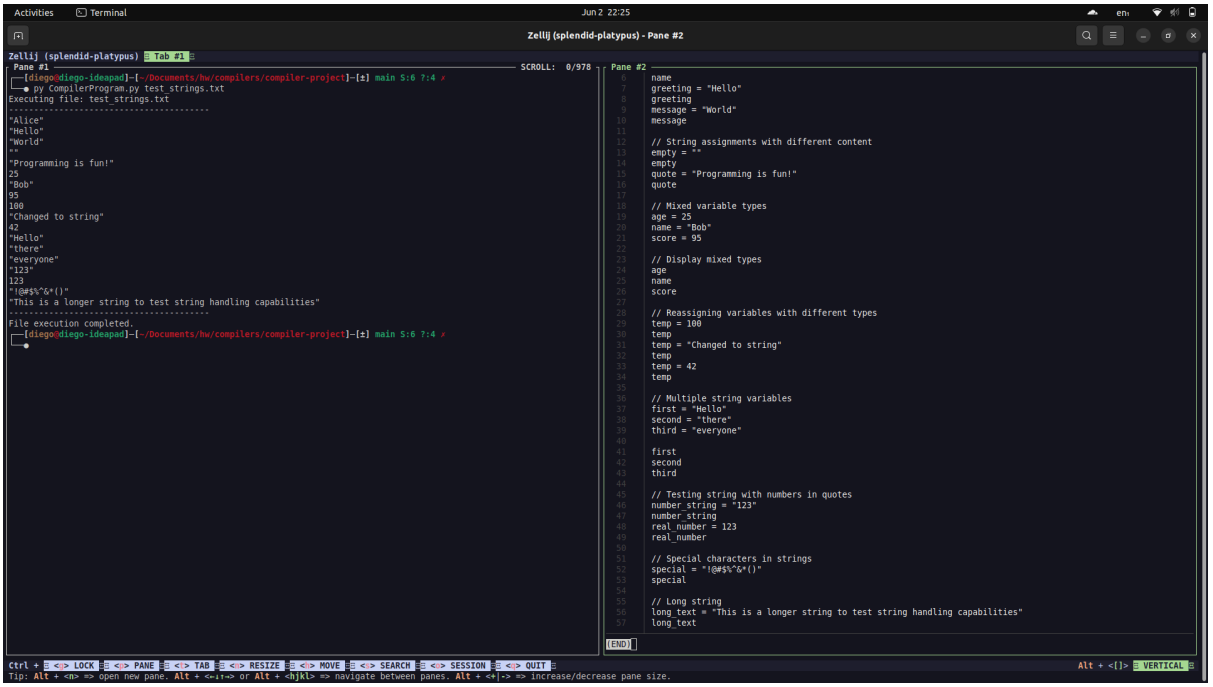
Brief and Phases..... 9
Lexical Analysis..... 10
Semantic Analysis..... 12
Execution..... 17
User Manual..... 22

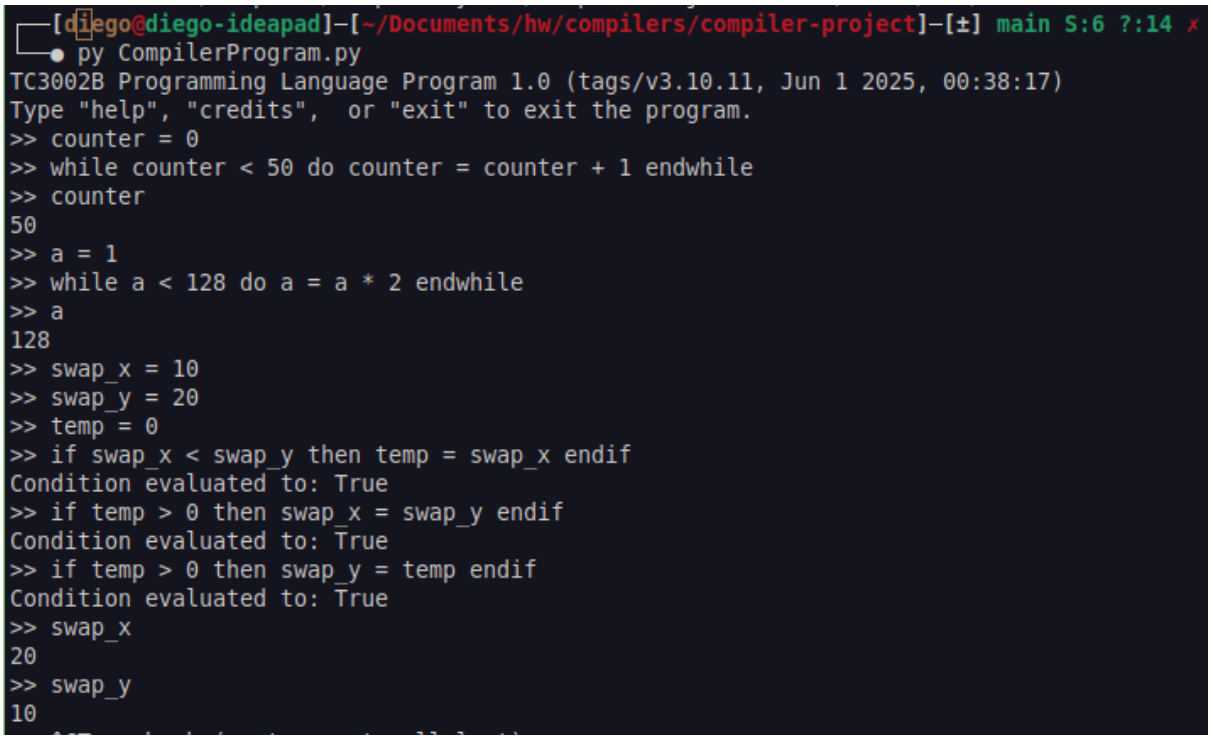

Activities
Terminal
Jun 2 21:32
Zellij (splendid-platypus) - Pane #1
Zellij (splendid-platypus) Tab #1
Pane #1
SCROLL: 0/286
[diego@diego-ideapad]-[~/Documents/hw/compilers/compiler-project]-[±] main ↓4 ?:6 ✗
py CompilerProgram.py test_arithmetic.txt
Executing file: test_arithmetic.txt
10
5
3
"Basic Arithmetic Operations"
15
5
50
"Test operator precedence"
14
20
"Complex expressions"
23
"Chained operations"
10
File execution completed.
Pane #2
File: test_arithmetic.txt
// Test file 1: Arithmetic Operations and Variables
// This file tests basic arithmetic operations, variable assignments, and precedence
// Basic variable assignments
x = 10
y = 5
z = 3
// Display initial values
x
y
z
// Basic arithmetic operations
str = "Basic Arithmetic Operations"
str
sum = x + y
sum
difference = x - y
difference
product = x * y
product
// Test operator precedence
str = "Test operator precedence"
str
result1 = 2 + 3 * 4
result1
result2 = (2 + 3) * 4
result2
// Complex expressions
str = "Complex expressions"
str
complex = x + y * z - 2
complex
// Chained operations
str = "Chained operations"
str
x
y
z
final = x + y - z * 2 + 1
final
Ctrl + <o> LOCK <p> PANE <t> TAB <r> RESIZE <m> MOVE <s> SEARCH <o> SESSION <q> QUIT
Alt + <[]> VERTICAL
Tip: Alt + <n> => open new pane. Alt + <←↓↑→> or Alt + <hjkl> => navigate between panes. Alt + <+|-> => increase/decrease pane size.


Activities
Terminal
Jun 2 22:02
Zellij (splendid-platypus) - Pane #2
Zellij (splendid-platypus) Tab #1
Pane #1
SCROLL: 0/748
[diego@diego-ideapad]-[~/Documents/hw/compilers/compiler-project]-[±] main ?:6 ✗
py CompilerProgram.py test_conditionals.txt
Executing file: test_conditionals.txt
Condition evaluated to: True
1
Condition evaluated to: False
Undefined variable 'result2' found!
0
Condition evaluated to: True
"a is greater"
Condition evaluated to: False
"c wins"
Condition evaluated to: False
"B"
Condition evaluated to: False
"not greater"
Condition evaluated to: True
21
Condition evaluated to: False
"positive"
File execution completed.
Pane #2
SCROLL: 0/82
[diego@diego-ideapad]-[~/Documents/hw/compilers/compiler-project]-[±] main ?:6 ✗
cat test_conditionals.txt
File: test_conditionals.txt
// Test file 2: Conditional Statements (If-Then-Else)
// This file tests if statements with various conditions
// Initialize test variables
a = 15
b = 10
c = 20
// Simple if statement (condition true)
if a > b then result1 = 1 endif
result1
// Simple if statement (condition false)
if b > a then result2 = 1 endif
result2
// If-else statement (condition true)
if a > b then status = "a is greater" else status = "b is greater" endif
status
// If-else statement (condition false)
if b > c then winner = "b wins" else winner = "c wins" endif
winner
// Nested conditions with arithmetic
score = 85
if score > 90 then grade = "A" else grade = "B" endif
grade
// Test with equal values
x = 5
y = 5
if x > y then comparison = "greater" else comparison = "not greater" endif
comparison
// Complex condition
num = 7
if num > 5 then tripled = num * 3 else tripled = num endif
tripled
// Test with negative numbers
temp = -5
if temp < 0 then sign = "negative" else sign = "positive" endif
sign
[diego@diego-ideapad]-[~/Documents/hw/compilers/compiler-project]-[±] main ?:6 ✗
Ctrl + <o> LOCK <p> PANE <t> TAB <r> RESIZE <m> MOVE <s> SEARCH <o> SESSION <q> QUIT
Alt + <[]> VERTICAL
Tip: Alt + <n> => open new pane. Alt + <←↓↑→> or Alt + <hjkl> => navigate between panes. Alt + <+|-> => increase/decrease pane size.


Activities
Terminal
Jun 2 22:08
Zellij (splendid-platypus) - Pane #2
Zellij (splendid-platypus) Tab #1
Pane #1
SCROLL: 0/836
[diego@diego-ideapad]-[~/Documents/hw/compilers/compiler-project]-[±] main S:6 U:1 ✗
py CompilerProgram.py test_loops.txt
Executing file: test_loops.txt
6
0
32
160
File execution completed.
Pane #2
SCROLL: 0/158
cat test_loops.txt
File: test_loops.txt
// Test file 3: While Loops
// This file tests while loop functionality with various scenarios
counter = 1
while counter < 6 do counter = counter + 1 endwhile
counter
countdown = 5
while countdown > 0 do countdown = countdown - 1 endwhile
countdown
x = 2
while x < 20 do x = x * 2 endwhile
x
value = 10
while value < 100 do value = value * 2 endwhile
value
Ctrl + LOCK PANE TAB RESIZE MOVE SEARCH SESSION QUIT
Alt + VERTICAL
Tip: Alt + <n> => open new pane. Alt + <←↓↑→> or Alt + <hjkl> => navigate between panes. Alt + <+|-> => increase/decrease pane size.


Activities
Terminal
Jun 2 22:20
Zellij (splendid-platypus) - Pane #1
Zellij (splendid-platypus) Tab #1
Pane #1
SCROLL: 0/924
[diego@diego-ideapad]-[~/Documents/hw/compilers/compiler-project]-[±] main S:6 U:1 ?:5 ✗
py CompilerProgram.py test_edge_cases.txt
Executing file: test_edge_cases.txt
Condition evaluated to: True
10
3
Condition evaluated to: False
2
10
5
Condition evaluated to: True
10
50
Condition evaluated to: True
""
Condition evaluated to: True
1
10
Condition evaluated to: True
17
File execution completed.
Pane #2
File: test_edge_cases.txt
// Test file 9: Edge Cases with Single-line Control Structures
// This file tests boundary conditions and edge cases
// Testing with zero values
zero = 0
one = 1
result = 5
if zero < one then result = 10 endif
result
while zero < 3 do zero = zero + 1 endwhile
zero
// Testing with equal values
equal_a = 5
equal_b = 5
comparison = 0
if equal_a > equal_b then comparison = 1 else comparison = 2 endif
comparison
// Loop that doesn't execute
no_execute = 10
while no_execute < 5 do no_execute = no_execute + 1 endwhile
no_execute
// Single iteration loop
single = 4
while single < 5 do single = single + 1 endwhile
single
// Large number operations
large = 100
small = 1
while small < 10 do small = small * 2 endwhile
if large > small then large = 50 endif
small
large
// String edge cases
empty_like = "x"
full = "complete"
flag = 0
if flag < 1 then empty_like = "" endif
empty_like
Ctrl + LOCK PANE TAB RESIZE MOVE SEARCH SESSION QUIT
Alt + VERTICAL
Tip: Alt + <n> => open new pane. Alt + <←↓↑→> or Alt + <hjkl> => navigate between panes. Alt + <+|-> => increase/decrease pane size.





CompilerTests output:

```
Unset
Running compiler on: test_files/test_arithmetic.txt

Executing file: test_files/test_arithmetic.txt
```

```
----------------------------------------
10
5
3
"Basic Arithmetic Operations"
15
5
50
"Test operator precedence"
14
20
"Complex expressions"
23
"Chained operations"
10
----------------------------------------
File execution completed.
Running compiler on: test_files/test_loops.txt
Executing file: test_files/test_loops.txt
----------------------------------------
6
0
32
160
----------------------------------------
File execution completed.
Running compiler on: test_files/test_conditionals.txt
Executing file: test_files/test_conditionals.txt
```

```
----------------------------------------
Condition evaluated to: True
1
Condition evaluated to: False
Undefined variable 'result2' found!
0
Condition evaluated to: True
"a is greater"
Condition evaluated to: False
"c wins"
Condition evaluated to: False
"B"
Condition evaluated to: False
"not greater"
Condition evaluated to: True
21
Condition evaluated to: False
"positive"
----------------------------------------
File execution completed.
Running compiler on: test_files/input.txt
Executing file: test_files/input.txt
----------------------------------------
5
10
----------------------------------------
File execution completed.
Running compiler on: test_files/test.txt
```

```
Executing file: test_files/test.txt
----------------------------------------
Condition evaluated to: True
3
----------------------------------------
File execution completed.
Running compiler on: test_files/test_edge_cases.txt
Executing file: test_files/test_edge_cases.txt
----------------------------------------
Condition evaluated to: True
10
3
Condition evaluated to: False
2
10
5
Condition evaluated to: True
16
50
Condition evaluated to: True
""
Condition evaluated to: True
1
10
Condition evaluated to: True
17
----------------------------------------
File execution completed.
```

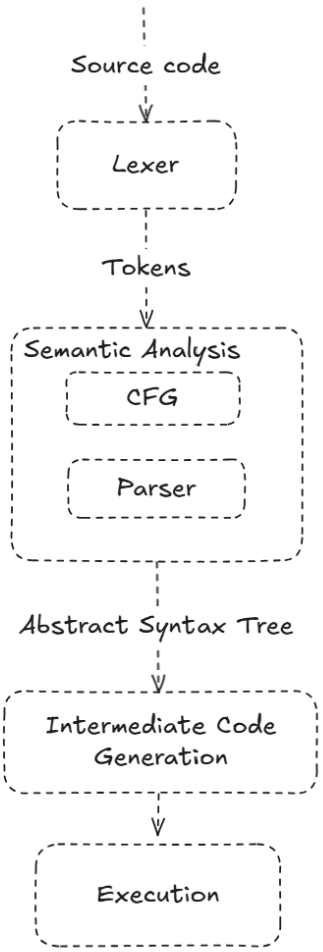
```
Running compiler on: test_files/test_strings.txt
Executing file: test_files/test_strings.txt
----------------------------------------
"Alice"
"Hello"
"World"
""
"Programming is fun!"
25
"Bob"
95
100
"Changed to string"
42
"Hello"
"there"
"everyone"
"123"
123
"!@#$%^&*()"
"This is a longer string to test string handling capabilities"
----------------------------------------
File execution completed.
```

Brief and Phases

The current document aims to document and explain all the phases and implementation of this simple compiler that achieves simple and minimal functionality demonstrating the knowledge of the different parts and implementations of a compiler.

The different phases of our implementation include:

1. Lexical Analysis (Lexer) -> Tokens
2. Syntax Analysis (CFG|Parser) -> AST
3. Intermediate code generation -> intermediate code
4. Execution



Lexical Analysis

The lexical analyzer (lexer) breaks down the input source code into a sequence of tokens. It's the first phase of compilation that converts the character stream into meaningful symbols.

This is achieved here by defining literal characters, and keywords, and recognizing identifiers, numbers, and strings. It also handles comments and whitespace.

In our implementation, the regex patterns are defined to match if, then, else, endif; while, do, endwhile; variable names, numbers, and string literals.

```python
class BasicLexer(Lexer):

    tokens = {NAME, NUMBER, STRING, IF, THEN, ELSE,

              ENDIF, WHILE, DO, ENDWHILE}

    ignore = "\t "

    literals = {"=", "+", "-", "/", "*", "(", ")", ",", ";", "<", ">", "!"}


       # regex being defined for keywords variables ints and strings

       # Tokenize directly the regex

    IF = r"if"

    THEN = r"then"

    ELSE = r"else"

    ENDIF = r"endif"

    WHILE = r"while"

    DO = r"do"

    ENDWHILE = r"endwhile"


    NAME = r"[a-zA-Z_][a-zA-Z0-9_]*"

    STRING = r"\".*?\""
```

```
    @_(r"\d+")
    def NUMBER(self, t):
        t.value = int(t.value)
        return t
    @_(r"//.*")
    def COMMENT(self, t):
        pass
    @_(r"\n+")
    def newline(self, t):
        self.lineno = t.value.count("\n")
```

Semantic Analysis

The Semantic Analysis takes as input the tokenized output that the lexer creates after taking the input of the source code. It consists in the CFG and the parser. The grammar rules that are implemented are the next ones:

```
Unset
# initial -> statement
1. statement -> ε                    (empty statement)
2. statement -> if_stmt              (if statement)
3. statement -> while_stmt           (while statement)
4. statement -> var_assign           (variable assignment)
5. statement -> expr                 (expression statement)
```

```
6. var_assign -> NAME "=" expr       (variable assignment with expression)
7. var_assign -> NAME "=" STRING     (variable assignment with string)

8. expr -> "(" expr ")"              (parenthesized expression)
9. expr -> expr "+" expr             (addition)
10. expr -> expr "-" expr            (subtraction)
11. expr -> expr "*" expr            (multiplication)
12. expr -> expr "/" expr            (division)
13. expr -> expr "<" expr            (less than comparison)
14. expr -> expr ">" expr            (greater than comparison)
15. expr -> "-" expr                 (unary minus)
16. expr -> NAME                     (variable reference)
17. expr -> NUMBER                   (numeric literal)

18. if_stmt -> IF expr THEN statement ENDIF                    (if without
else)
19. if_stmt -> IF expr THEN statement ELSE statement ENDIF    (if with else)

20. while_stmt -> WHILE expr DO statement ENDWHILE          (while loop)
```

The implementation of the parser first defines the tokens and the precedence for operations

Python

```
class BasicParser(Parser):

       tokens = BasicLexer.tokens

       precedence = (
```

```
            ("left", "<", ">"),           # Comparison operators
            ("left", "IF", "THEN", "ELSE", "ENDIF"),  # Conditional keywords
            ("left", "+", "-"),           # Addition/Subtraction
            ("left", "*", "/"),           # Multiplication/Division
            ("right", "UMINUS"),          # Unary minus
        )
```

Then we implemented all the Production Rules

Python

```
        def __init__(self):
            self.env = {}


    # Statement Rules
    @_("")
    def statement(self, p):
        pass


    @_("if_stmt")
    def statement(self, p):
        return p.if_stmt


    @_("while_stmt")
    def statement(self, p):
        return p.while_stmt


    @_("var_assign")
    def statement(self, p):
```

```
        return p.var_assign
    # Variable Assignment Rules
    @_('NAME "=" expr')
    def var_assign(self, p):
        return ("var_assign", p.NAME, p.expr)

    @_('NAME "=" STRING')
    def var_assign(self, p):
        return ("var_assign", p.NAME, p.STRING)

    # Expression Rules
    @_("expr")
    def statement(self, p):
        return p.expr

    @_('"(" expr ")"')
    def expr(self, p):
        return p.expr

    @_('expr "+" expr')
    def expr(self, p):
        return ("add", p.expr0, p.expr1)

    @_('expr "-" expr')
    def expr(self, p):
        return ("sub", p.expr0, p.expr1)

    @_('expr "*" expr')
```

```
    def expr(self, p):
        return ("mul", p.expr0, p.expr1)

    @_('expr "/" expr')
    def expr(self, p):
        return ("div", p.expr0, p.expr1)

    @_('expr "<" expr')
    def expr(self, p):
        return ("lt", p.expr0, p.expr1)

    @_('expr ">" expr')
    def expr(self, p):
        return ("gt", p.expr0, p.expr1)
    # If
    # Adding the if and while statements
    # IF works like: if expr then statement [else statement] endif
    @_("IF expr THEN statement ENDIF")
    def if_stmt(self, p):
        return ("if", p.expr, p.statement, None)

    @_("IF expr THEN statement ELSE statement ENDIF")
    def if_stmt(self, p):
        return ("if", p.expr, p.statement0, p.statement1)
    # While rules
    # WHILE works like: while expr do statement endwhile
    @_("WHILE expr DO statement ENDWHILE")
    def while_stmt(self, p):
```

```
        return ("while", p.expr, p.statement)
    # minus unary rule
    @_('"-" expr %prec UMINUS')
    def expr(self, p):
        return p.expr


    @_("NAME")
    def expr(self, p):
        return ("var", p.NAME)


    @_("NUMBER")
    def expr(self, p):
        return ("num", p.NUMBER)
```

Execution

For the execution the implementation uses a tree walk where recursively goes through the nodes until terminal states and defining expressions to resolve the main statement.

All the rules are executed using their equivalent in python as follows:

```
Python
def walkTree(self, node):
```

```
        if isinstance(node, int):
            return node
        if isinstance(node, str):
            return node

        if node is None:
            return None

        if node[0] == "program":
            if node[1] == None:
                self.walkTree(node[2])
            else:
                self.walkTree(node[1])
                self.walkTree(node[2])

        if node[0] == "num":
            return node[1]

        if node[0] == "str":
            return node[1]

        if node[0] == "add":
            return self.walkTree(node[1]) + self.walkTree(node[2])
        elif node[0] == "sub":
            return self.walkTree(node[1]) - self.walkTree(node[2])
        elif node[0] == "mul":
            return self.walkTree(node[1]) * self.walkTree(node[2])
        elif node[0] == "div":
```

```
            return self.walkTree(node[1]) / self.walkTree(node[2])


        if node[0] == "lt":
            return self.walkTree(node[1]) < self.walkTree(node[2])
        elif node[0] == "gt":
            return self.walkTree(node[1]) > self.walkTree(node[2])
```

For the more complex implementation of If we evaluate the condition's expression value. Then we execute depending on its bracket.

Python

```
      if node[0] == "if":
            condition = self.walkTree(node[1])
            print("Condition evaluated to:", condition)
            if condition:
                return self.walkTree(node[2])  # then branch
            elif node[3] is not None:
                return self.walkTree(node[3]) # else branch
```

Similarly for the while condition we execute in an internal python while loop, but we don't save the conditions value since each loop it changes its value.

```python
        if node[0] == "while":
            while self.walkTree(node[1]):
                self.walkTree(node[2])
            return None
```

Extras:

We also added some extra features including help and credits:

```
TC3002B Programming Language Program 1.0 (tags/v3.10.11, Jun 1 2025, 00:38:17)
Type "help", "credits",  or "exit" to exit the program.
>> help
This is a simple interpreter for a basic programming language.
You can use variables, arithmetic operations, and if statements.
Type 'credits' to see the developers.
Keywords: if, then, else, endif.
Operators: +, -, *, /, <, >, =.
Type 'exit' or 'quit' to exit the program.
Type 'help' to see this message again.
Type 'credits' to see the developers.
Type 'file <filename>' to execute a file.
Examples:
>> x = 5
>> y = "Hello"
>> if x < 10 then y = "Less than 10" else y = "10 or more" endif
>> y
"Less than 10"
In order to execute a file, run the program and then the <filename>'.
For example: python CompilerProgram.py test.txt
or: ./compiler test.txt
>>
```

```
TC3002B Programming Language Program 1.0 (tags/v3.10.11, Jun 1 2025, 00:38:17)
Type "help", "credits",  or "exit" to exit the program.
>> credits
Developed by Ivan Romero (A00833623) and Diego Hernandez (A00834015).
Class TC3002B - Programming Language. Compiler Module by Dr. Kingsley Okoye.
>>
```

Also file compiling.

```
PS C:\Users\iwell\Desktop\compiler-project> python.exe .\CompilerProgram.py .\input.txt
Executing file: .\input.txt
----------------------------------------
5
10
"Input from file execution"
----------------------------------------
File execution completed.
PS C:\Users\iwell\Desktop\compiler-project>
```

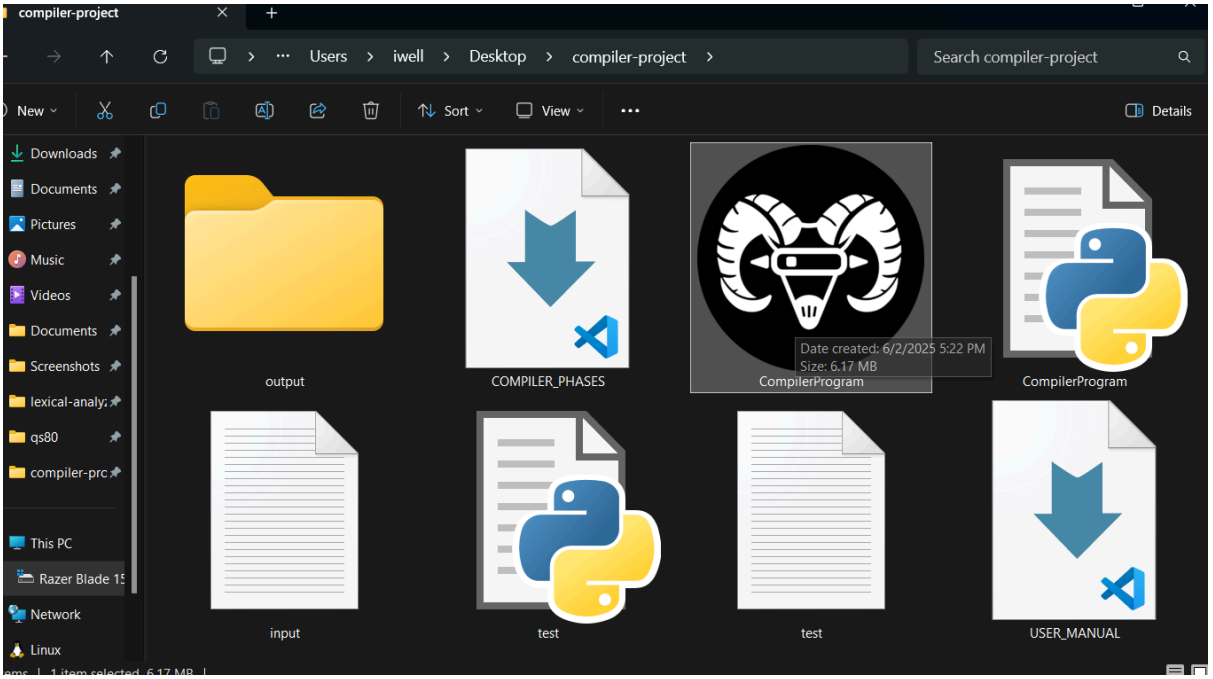
this was implemented with the next additions similar as the CLI implimentation:

Python

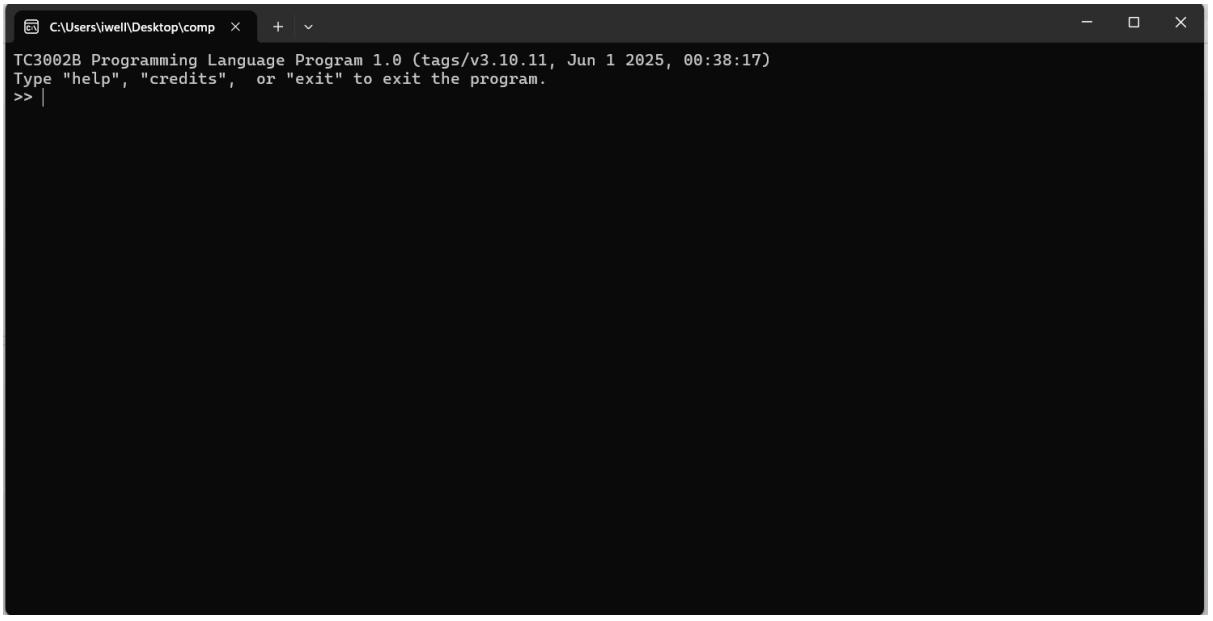
```python
def execute_file(filename):
        """Execute code from a file by using: python CompilerProgram.py
<filename> or ./compiler <filename>"""
    if not os.path.exists(filename):
        print(f"Error: File '{filename}' not found.")
        return

    try:
        with open(filename, "r") as file:
            content = file.read()

        lexer = BasicLexer()
        parser = BasicParser()
        env = {}
```

```
        print(f"Executing file: {filename}")
        print("-" * 40)

        # Split content into lines and execute each non-empty line
        lines = content.strip().split("\n")
        for line_num, line in enumerate(lines, 1):
            line = line.strip()
            if line:
                try:
                    tree = parser.parse(lexer.tokenize(line))
                    BasicExecute(tree, env)
                except Exception as e:
                    print(f"Error on line {line_num}: {e}")

        print("-" * 40)
        print("File execution completed.")

    except Exception as e:
        print(f"Error reading file '{filename}': {e}")
```

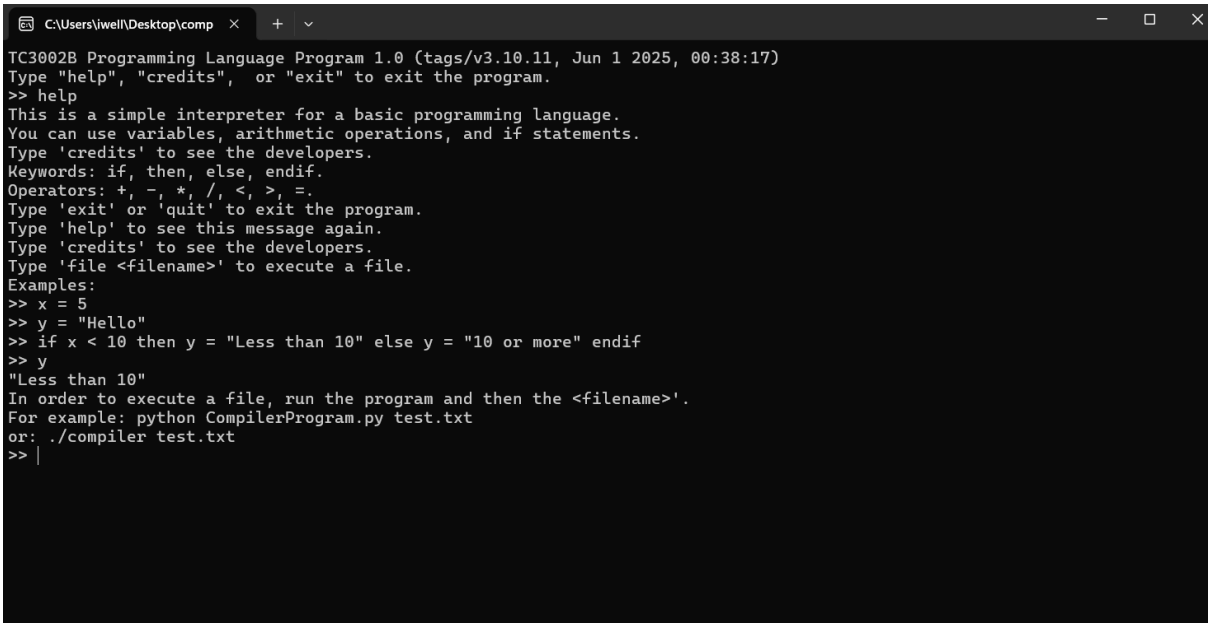
User Manual



Having our executable we are shown the cli:



we added a brief help display of the feature as shown below:



Language features

1. Data Types

Numbers

- Integer values are supported
- Examples: `42`, 0, -15

Strings

- String literals enclosed in double quotes
- Examples: "hello", "world"`, "123"

Variables

- Variable names must start with a letter or underscore
- Can contain letters, numbers, and underscores
- Examples: x, counter, my_var, _temp

2. Operators

Arithmetic Operators

- + : Addition
- - : Subtraction
- * : Multiplication
- - : Unary minus (negation)

Comparison Operators

- < : Less than
- > : Greater than

Assignment Operator

- = : Variable assignment

3. Language Constructs

Variable Assignment

```
Unset
variable_name = expression

variable_name = "string"
```

Arithmetic Expressions

```
Unset
5 + 3

10 - 4

6 * 7

20 / 4
```

```
(5 + 3) * 2
```

Comparison Expressions

```
Unset
5 > 3

10 < 20

x > y
```