# Project: Compiler and Virtual Machine for a Programming Language

# SparkUp

SER 502: Languages and Prog Paradigms

# TEAM-12

Pranav Kumar Jalagam

Naga Durga Amarnadh Reddy Dodda

Mokshith Sai Gadela

Harish Bandlamudi

# Features of SparkUp

**Modular Structure**:

- Components include **Initialization**, **Computation**, and **Conclusion**.

**Data Types**:

- Supports basic types: `int`, `float`, `str`, `bool`.

**Variable Management**:

- Variables are declared using `let` with type assignment and initialization.

**Expressions**:

- Arithmetic (`+`, `-`, `*`, `/`), Logical (`&&`, `||`, `==`, etc.), and **Ternary expressions** (`cond ? expr1 : expr2`).

**Control Flow**:

- **Conditionals**: `chk` (if) with optional `alt` (else).
- **Loops**: `while` and `for`.

# Features of SparkUp

**Input/Output**:

- Print functionality using `print(expr)`.

**Pre-Defined Literals**:

- **Integer**, **Float**, **String**, and **Boolean** literals (`true`, `false`).

**Logical Expressions**:

- Support for complex logic (`not`, comparisons, and logical operators).

**Encapsulation**:

- Components like loops and conditionals encapsulate logic within `{}` blocks.

**Program Termination**:

- Explicit conclusion using the `fin` keyword.

**Custom Syntax**:

- Unique keywords (`chk`, `alt`, `fin`, etc.) to simplify syntax.

# Components of SparkUp

**1. Initialization:**

**Syntax**: let <variable_type> <variable_name> = <expression>

- **Let**:Introduces a variable definition.
- **Variable_type**: Specifies the type such as int, float, str, or bool.
- **Variable_name**: Represents unique name for referencing the variable.
- **Expression**: The initial value assigned to the variable Example: let int x = 10
- **Example**:  let int x = 10

# Components of SparkUp

**2. Computation:**

**a. Print statements:**

**Syntax**: print(<expression>)

**b. Assignments:**

**Syntax**: <variable_name> = <expression>

# Components of SparkUp

**c. Conditionals:**

**Syntax:**

```
chk (<logical_expression>) {
<components>
}
alt {
<components>
}
```

# Components of SparkUp

**d. Loops:**

**while loop syntax**:

while (<logical_expression>)
{
 <components>
}

**for loop syntax:**

for (<assignment>; <logical_expression>; <assignment>)
{
 <components>
}

**3. Conclusion: Syntax:** fin

# Grammar

```
program: components EOF
components: initialization NEWLINE conclude* computation NEWLINE conclude*
""" PRE-DEFINITIONS """
"""
asnmt_op : '='
variable_type : 'int' | 'float' | 'str' | 'bool'
lowercase : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' |
'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
'y' | 'z'
digit : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
variable_name : lowercase (lowercase | digit)*
intLiteral : digit+
floatLiteral : intLiteral '.' intLiteral
stringLiteral : "'" (lowercase | digit | ' ')* "'" | '"' (lowercase | digit |
' ')* '"'
boolLiteral : 'true' | 'false'
"""
""" COMPONENTS """
initialization : 'let' variable_type variable_name asnmt_op expression
```

# Grammar

```
computation : operation+
operation : print_statement | assignment | conditional | loop
""" STATEMENTS """
print_statement : 'print' '(' expression ')'
assignment : variable_name asnmt_op expression
""" EXPRESSIONS """
expression : intLiteral
           | floatLiteral
           | stringLiteral
           | boolLiteral
           | variable_name
           | arithmeticExpression
           | logicalExpression
           | ternaryExpression
arithmeticExpression : expression ( '+' | '-' | '*' | '/' ) expression
logicalExpression : expression ( '&&' | '||' | '==' | '!=' | '>' | '<' | '>='
| '<=' ) expression
                  | 'not' expression
ternaryExpression : expression '?' expression ':' expression
""" CONDITIONALS AND LOOPS """
conditional : 'chk' '(' logicalExpression ')' '{' components '}' ('alt' '{'
components '}')?
loop : 'while' '(' logicalExpression ')' '{' components '}'
     | 'for' '(' assignment ';' logicalExpression ';' assignment ')' '{'
components '}'
conclude : 'fin'
```

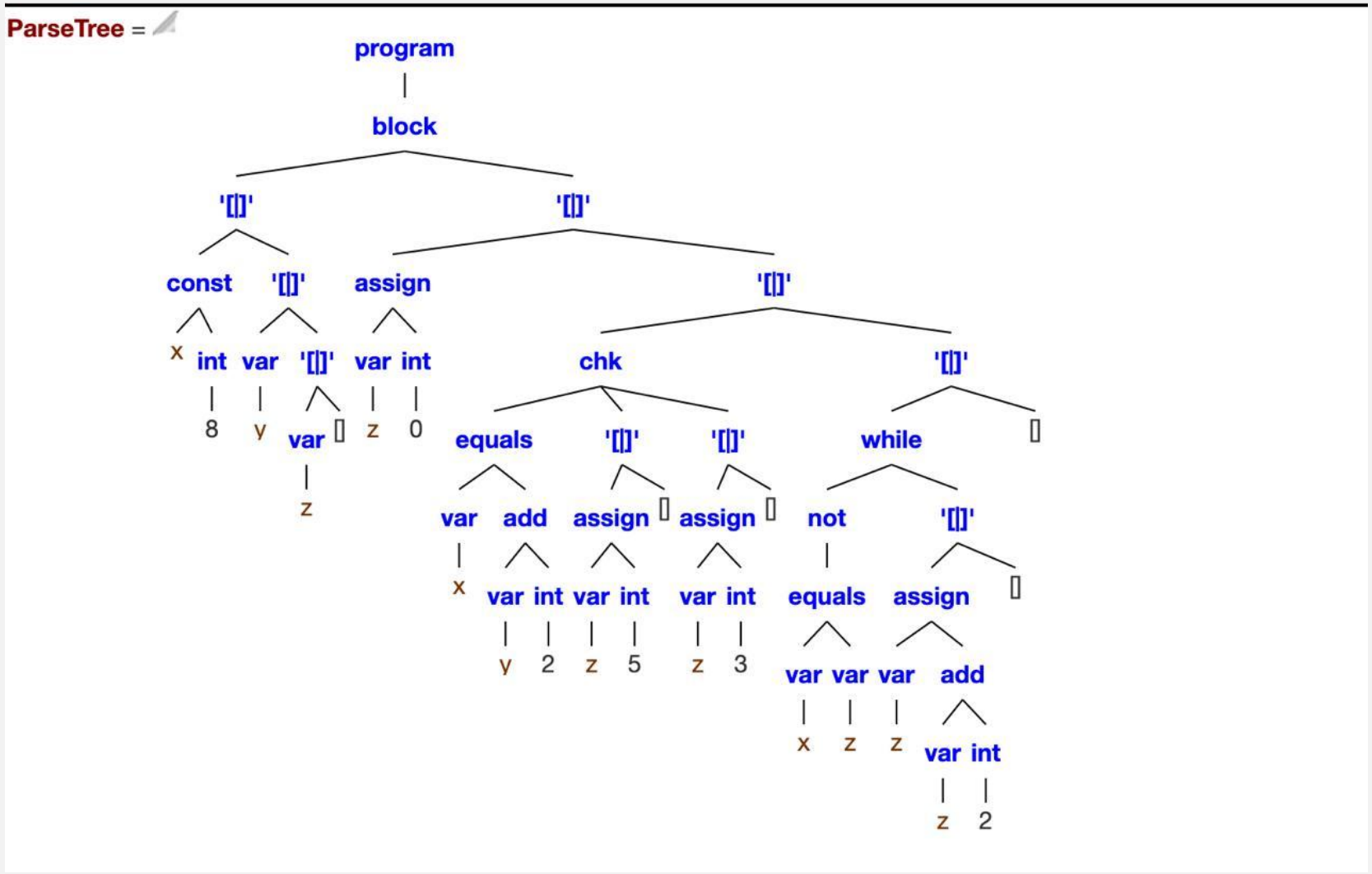
# Sample Code

```
let int x = 8
let int y
let int z
z=0

chk (x ==y+2) {
   z=5
} alt {
    z=3
}

while(not (x==z)){
   z=z+2
}
```

# Parse Tree


ParseTree =
program
block
'[]'
'[]'
const
'[]'
assign
'[]'
x
int
var
'[]'
var
int
chk
'[]'
8
y
var
z
0
equals
'[]'
'[]'
while
var
add
assign
assign
not
'[]'
x
var
int
var
int
var
int
equals
assign
y
2
z
5
z
3
var
var
var
add
x
z
z
var
int
z
2

# Interpreter

```python
import sys
import os

# Add the src folder to the Python path
script_dir = os.path.dirname(os.path.abspath(__file__))
src_path = os.path.join(script_dir, "../src")
sys.path.insert(0, src_path)

from sparkup_lexer import lexer
from sparkup_parser import parser

# Execution context (global variables)
execution_context = {}

def evaluate_expression(expression):
    """Evaluate an expression recursively."""
    if isinstance(expression, tuple):  # Binary or ternary operation
        op = expression[0]
        if op == 'ternary':  # Ternary operator
            condition = evaluate_expression(expression[1])
            if condition:
                return evaluate_expression(expression[2])
            else:
                return evaluate_expression(expression[3])
        elif op in ('+', '-', '*', '/', '<', '>', '<=', '>=', '==', '!='):
            left_val = evaluate_expression(expression[1])
            right_val = evaluate_expression(expression[2])
            if op == '+':
                return left_val + right_val
            elif op == '-':
                return left_val - right_val
            elif op == '*':
                return left_val * right_val
            elif op == '/':
                if right_val == 0:
                    raise ValueError("Division by zero")
                return left_val / right_val
            elif op == '<':
                return left_val < right_val
            elif op == '>':
                return left_val > right_val
            elif op == '<=':
                return left_val <= right_val
            elif op == '>=':
                return left_val >= right_val
```

# Interpreter

```
            elif op == '==':
                return left_val == right_val
            elif op == '!=':
                return left_val != right_val
    elif isinstance(expression, str):  # Variable reference or string literal
        if expression in execution_context:
            return execution_context[expression]
        elif expression.startswith('"') and expression.endswith('"'):  # String literal
            return expression[1:-1]  # Remove quotes
        else:
            return expression  # Treat as plain string if not a variable
    else:  # Literal value (int, float, bool)
        return expression

def execute_statement(statement):
    """Execute a single parsed statement."""
    if statement['type'] == 'assignment':
        execution_context[statement['var']] = evaluate_expression(statement['value'])
    elif statement['type'] == 'print':
        value = evaluate_expression(statement['value'])
        print(value)
    elif statement['type'] == 'conditional':
        condition = evaluate_expression(statement['condition'])
        if condition:
            execute_program(statement['then'])
        elif 'else' in statement:
            execute_program(statement['else'])
    elif statement['type'] == 'while':
        while evaluate_expression(statement['condition']):
            execute_program(statement['body'])
    elif statement['type'] == 'for':
        execute_statement(statement['init'])
        while evaluate_expression(statement['condition']):
            execute_program(statement['body'])
            execute_statement(statement['update'])

def execute_program(program):
    """Execute the parsed program."""
    for statement in program:
        execute_statement(statement)

def load_skp_file(filename):
    """Load and read a .skp file."""
    if not filename.endswith('.skp'):
```

# Interpreter

```
        raise ValueError("Invalid file extension. Please use a .skp file.")
    with open(filename, 'r') as file:
        code = file.read()
    return code

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: skp data/test.skp")
        sys.exit(1)

    filename = sys.argv[1]
    try:
        code = load_skp_file(filename)
    except Exception as e:
        print(f"Error: {e}")
        sys.exit(1)

    # Tokenizing and parsing
    lexer.input(code)
    program = parser.parse(code)

    if program:
        print("Program parsed successfully. Executing...")
        execute_program(program)
    else:
        print("Parsing failed.")
```

# Sample Code

```
let int x = 0
let int y = 5

while (x < y) {
    print("x value:")
    print(x)

    chk (x < 3) {
        print("x is less than 3")
    } alt {
        print("x is greater than or equal to 3")
    }

    x = x + 1
}

print("While loop completed.")
```

# Sample Run and Output

```
PS C:\Users\moksh\OneDrive\Desktop\SER502-Sparkup-Te
Initialization: x = 0
Components parsed
Initialization: y = 5
Components parsed
Print statement: x value:
Components parsed
Print statement: x
Components parsed
Print statement: x is less than 3
Components parsed
Print statement: x is greater than or equal to 3
Components parsed
Conditional statement parsed
Components parsed
Assignment: x = ('+', 'x', 1)
Components parsed
While loop parsed
Components parsed
Print statement: While loop completed.
Components parsed
Program successfully parsed!
Program parsed successfully. Executing...
```

```
x value:
Program successfully parsed!
Program successfully parsed!
Program parsed successfully. Executing...
x value:
0
x is less than 3
x value:
1
x is less than 3
x value:
2
x is less than 3
x value:
3
x is greater than or equal to 3
x value:
4
x is greater than or equal to 3
While loop completed.
```

# Future Scope

- **Advanced Data Types**: Add arrays, dictionaries, or objects.
- **Error Handling**: Introduce syntax and runtime error management.
- **Functions**: Enable reusability with user-defined functions.
- **File I/O**: Allow reading/writing external files and database interaction.

Thank You