SER 502: PROJECT MILESTONE - 02

Pranav Kumar Jalagam (pjalagam@asu.edu)
Naga Durga Amarnadh Reddy Dodda (ndodda4@asu.edu)
Mokshith Sai Gadela (mgadela@asu.edu)
Harish Bandlamudi (hbandlam@asu.edu)

Sparkup Programming Language Design:

Sparkup is a programming language designed to offer fundamental programming constructs, easy-to-read syntax, and support for essential variable types. The language structure is divided into three main components: Initialization, Computation, and Conclusion. Additionally, Sparkup supports common programming constructs like conditional statements, loops, and various expressions, aiming to enhance readability and simplicity.

Components of the Sparkup language:

1. **Initialization:** In this process we define new variables with a specific type and assign them values. Each variable must have a type and name. This process allows us to store and retrieve data using the variable names we defined.

   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

2. **Computation:** The computation component is where the program's main processing happens. It consists of a series of operations that include print statements, assignments, conditionals, and loops. This section handles calculations, data manipulation, and logical operations.
   a. **Print statements:** The print statement outputs the result of an expression, such as a variable's value, a calculation, or a logical result.
      Syntax: print(<expression>)

   b. **Assignments**: Assignments update the value of an already initialized variable with a new expression result.
      Syntax: <variable_name> = <expression>

c. **Conditionals:** Conditionals evaluate Boolean expressions and execute specific blocks of code based on the condition's result. If the expression is true, the code within the chk block runs; otherwise, it executes if an alt block is provided.

Syntax:

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

Example:

```
chk (x > 5) {
print("x is greater than 5")
}
alt {
print("x is 5 or less")
}
```

d. **Loops:** Loops repeat a set of operations as long as a condition holds. Sparkup supports for and while loops.

while loop syntax:

```
while (<logical_expression>)
{
 <components>
}
```

Example:

```
while (x < 5) {
print(x) x = x + 1
}
```

for loop syntax:

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

Example:

```
for (let int i = 0; i < 10; i = i + 1) {
print(i)
}
```

3. **Conclusion:** The conclusion component provides an optional part where final results are displayed or further computed. The keyword **fin(finish)** signifies the end of the computation block.

   Syntax: fin

   Example:

   ```
   print(x)
   Fin
   ```

Expressions and Operators:

Sparkup supports various types of expressions:

- **Arithmetic Expressions**: Combine values using +, -, *, or /.
- **Logical Expressions**: Evaluate Boolean results with &&, ||, ==, !=, <, >, <=, and >=.
- **Ternary Expressions**: Use ? : to return different results based on a Boolean condition.

Examples:

- **Arithmetic**: x + y * 3
- **Logical**: (x > 5) && (y < 10)
- **Ternary**: x > 0 ? x : -x

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
```

```
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'
```

INTERPRETER: Python was selected as the interpreter language for the Sparkup project due to its unique strengths in readability, support for parsing libraries, and suitability for rapid development. Here's a breakdown of the factors that make Python an ideal choice:

1. **Readability and Structure**: Python's clean syntax and straightforward structure make it a popular language for complex logic and parsing tasks. Its readability supports clear, maintainable code, which is especially beneficial in educational projects focused on interpreters and parsers.
2. **Robust Parsing Libraries**: Python includes well-established libraries, such as **PLY (Python Lex-Yacc)**, that are specifically designed for building lexers and parsers. Modeled after traditional Lex and Yacc tools, PLY streamlines both tokenization and parsing tasks and simplifies defining grammar rules and handling syntax errors.
3. **Ideal for Prototyping and Development**: Python's simple syntax and dynamic nature make it highly effective for rapid prototyping. This enables quick testing and refinement of

Sparkup's lexer and parser components, allowing developers to iterate faster and refine as needed.

4. **Comprehensive Documentation and Community Resources**: With Python's large user community and extensive resources, developers have access to abundant documentation and troubleshooting support. This support network allows issues to be addressed quickly, helping the project remain on schedule.
5. **Cross-Platform Accessibility**: As a cross-platform language, Python allows the Sparkup interpreter to run on various operating systems without special adjustments. This versatility is ideal for a project intended to be tested and accessed by users on different platforms.
6. **Memory Management and Built-In Data Structures**: Python's built-in data types, such as lists and dictionaries, along with automatic memory management, simplify many interpreter functions, including symbol table management, expression evaluation, and the construction of abstract syntax trees (ASTs). These features eliminate the need for low-level memory handling, making the development process more efficient.

In summary, Python combines simplicity, flexibility, and strong support for parsing and interpreter development, making it a practical choice for Sparkup. It allows the project team to concentrate on core parsing and interpretive functions without being burdened by low-level complexities, resulting in a functional and maintainable educational tool.

PARSER: The Sparkup project's parser is developed using **PLY (Python Lex-Yacc)**, a Python library inspired by traditional Lex and Yacc tools, which allows developers to build a clear and structured parser with ease. Here's why PLY was chosen and how it enhances the parser's functionality:

1. **Familiarity with Lex-Yacc Standards**: PLY mirrors the established syntax and design of Lex and Yacc, allowing developers to utilize classic parsing techniques in a Python-friendly way. This familiarity benefits those experienced with traditional parsing tools while leveraging Python's simplicity and readability.
2. **Ease of Tokenization and Grammar Definition**: PLY simplifies the creation of tokens and grammar rules by using regular expressions for tokens and functions for grammar definitions. With built-in support for token precedence and error handling, PLY enables developers to keep the lexer and parser organized and compact.
3. **Enhanced Error Handling**: PLY's error-handling capabilities are especially valuable for debugging, providing precise messages on syntax errors, including location and context. This feature is essential in educational projects like Sparkup, where understanding and troubleshooting syntax issues is central to the learning process.
4. **Efficient Abstract Syntax Tree (AST) Construction**: PLY supports structured representation of expressions and statements, allowing developers to build an Abstract Syntax Tree (AST) that can be further used for interpreting or generating code. By defining

grammar rules that return structured data (like tuples or dictionaries), PLY makes AST construction straightforward and organized for further processing.

5. **Flexibility and Extensibility**: Being Python-based, PLY allows for flexible rule definitions that can include custom actions or data transformations, giving developers the ability to manage parsing outcomes with greater precision. This adaptability supports unique language features and transformations within the Sparkup parser.
6. **Seamless Integration with Python's Tools**: Because PLY is native to Python, it integrates smoothly with other Python modules, such as those for file handling, data manipulation, and unit testing. This makes it easy to read source code, generate parse trees, and manage outputs in a single ecosystem, improving both development efficiency and maintainability.

Overall, PLY provides a practical, Python-based solution for building parsers that is ideal for Sparkup's educational goals. It combines Lex-Yacc's strengths with Python's flexibility, offering efficient token management, rule definition, and error handling, along with robust AST construction capabilities—all of which make it an excellent fit for interpreter projects and prototyping.

Parse Tree Generation:

```
sample_query(ParseTree) :-
    Tokens = [let, int, x, '=', 8, newline,
              let, int, y, newline,
              let, int, z, newline,
              z, '=', 0, newline,
              chk, '(', x, '==', y, '+', 2, ')', '{', z, '=', 5, '}',
              alt, '{', z, '=', 3, '}', newline,
              while, '(', not, '(', x, '==', z, ')', ')', '{', z, '=', z, '+',
2, '}',
              eof],
    phrase(program(ParseTree), Tokens).
```

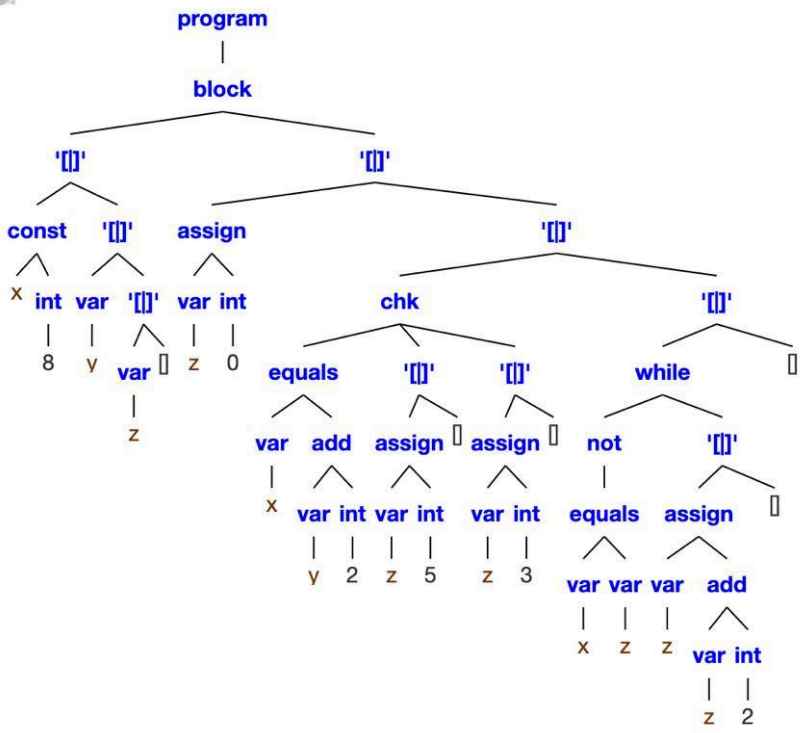
Input Query:

```
?- sample_query(ParseTree).
```

Output Parse Tree:

```
program(block([const(x,int(8)), var(y), var(z)],[assign(var(z),int(0)),
chk(equals(var(x),add(var(y),int(2))),[assign(var(z),int(5))],[assign(var(z),in
t(3))]),
while(not(equals(var(x),var(z))),[assign(var(z),add(var(z),int(2)))])]))
```

ParseTree =


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