PROGRAMMING LANGUAGE FOR MINECRAFT CRAFTING AUTOMATION

Gabriela Martínez Eslava - 20202020117
Luis Alejandro Cely Díez - 20211020093

FRANCISCO JOSÉ DE CALDAS DISTRICT UNIVERSITY

SYSTEMS ENGINEERING CURRICULUM PROJECT

COMPUTER SCIENCES III

Bogotá D.C.
January 2025

PROGRAMMING LANGUAGE FOR MINECRAFT CRAFTING AUTOMATION

Gabriela Martínez Eslava - 20202020117
Luis Alejandro Cely Díez - 20211020093

CLASS PROJECT FOCUSED ON THE CREATION OF A PROGRAMMING LANGUAGE FOR AUTOMATING MINECRAFT CREAFTING.

Director
CARLOS ANDRÉS SIERRA VIRGÜEZ

FRANCISCO JOSÉ DE CALDAS DISTRICT UNIVERSITY

SYSTEMS ENGINEERING CURRICULUM PROJECT

COMPUTER SCIENCES III

Bogotá D.C.
January 2025

Contents

Part I
INTRODUCTION

In the world of Minecraft, crafting is a core mechanic that plays a significant role in the gameplay experience. However, as players advance, the need for repetitive crafting tasks often becomes a challenge. To simplify and enhance this experience, we propose a programming language designed specifically for Minecraft crafting automation and customization. This language aims to address the growing need for more efficient crafting processes by allowing players to create custom recipes, automate systems, and optimize resource management in a structured and efficient way.

The scope of this language is broad and is intended for players of all level of experience, whether they are casual players, modders, or server administrators. The language is designed with a simple and intuitive syntax, ensuring that players can easily use it regardless of the complexity of the recipes they wish to automate or create. By providing a user-friendly approach, the language empowers players to better manage their crafting tasks and customize their Minecraft experience.

In order to develop this language, we undertook a comprehensive design process. We started by exploring the necessary steps to make this programming language feasible, focusing on the design and analysis of both its syntax and semantics. This involved creating a set of lexemes and developing a lexer and parser to analyze the grammatical structure of the language. Finally, we implemented the semantic rules and led the language through a compilation process, ensuring its functionality and usability within the Minecraft environment.

1 JUSTIFICATION

This programming language for Minecraft aims to solve issues related to the growing complexity of recipes, automation, and resource optimization, as mentioned earlier. It would be a powerful tool for every user, whether they are highly experienced or not, enhancing both gameplay and development potential. Its advantages consist of:

- **Decreasing complexity in Minecraft and mods:** As Minecraft and its mods introduce more items, resources, and recipes, managing them becomes more complex, especially manual manipulation. This language would solve this by enabling automated workflows, allowing players to define optimized crafting sequences, saving time and effort.
- **Automation and Resource optimization:** Although Minecraft offeres some automation tools like redstone and command blocks, they are limited and often require manual effort. This language would let players automate and optimize complex crafting systems, reducing time spent on repetitive tasks and improving resource efficiency.
- **Customization and Balance:** Mantaining balance can be challenging with certain items being too easy to obtain, specially in multiplayer servers. The language would allow server administrators and developers to adjust crafting recipes, creating a fair and dynamic gameplay. Custom scripts could control when certain items are crafteable based on players progress.
- **Ease for Modders and Content creators:** Both will benefit from this language by simplifying recipe creation and customization. Instead of manually editing JSON files, they could write recipes more efficiently, enhancing gameplay and integrating their mods seamlessly.

Part II
SYNTAX AND SEMANTIC DESIGN

In the design of a programming language it's essential to define keywords, tokens and lexemes that form the foundation of the language's syntax and structure. They're fundamental concepts that relate to the process of lexical analysis, that is the first step of compilation or interpretation.

2 KEYWORDS

They are reserved words that have a predefined meaning and are an integral part of the syntax of a programming language. In our case, these keywords, which cannot be redefined or used as identifiers, are crucial for defining the structure of the program that will allow us to craft items and recipes. For our Minecraft programming language, these keywords may have different meanings compared to other languages. Some are unique to our language, while others are shared with many common programming languages to maintain simplicity and ease of use. Using well-established keywords, we ensure that the language remains intuitive for both experienced developers and beginners, allowing them to quickly understand the syntax and begin automating crafting processes in the game.

- **recipe:** This keyword defines a new crafting recipe.
- **input:** Input specifies the list of input items required for crafting.
- **output:** In contrast to the input, this specifies the resulting item after crafting.
- **quantity:** Defines how many of the output item are produced.
- **tool_required** This specifies which crafting tool (such as a crafting table, furnace, etc.) is required.
- **if and else:** Conditional statements shared with another languages for adding flexibility.
- **while, for:** Looping structures shared with another languages as well.
- **func:** Defines a reusable crafting function or recipe block.
- **return:** Returns a specific result from a function.
- **craft:** Command to trigger the actual crafting process.
- **log:** This is used to print output for debugging.

3 TOKENS

Tokens represents a basic building block of the source code, this is the result of the lexical analysis phase as well, where the source code is split into smaller components for further processing. In this order, they're the smallest elements recognized by the language parser; these include identifiers, symbols and operators used in the code. Some essential tokens can be:

- **Comments:** They represent inline comments that start with "`//`" and extend to the end of the line.
- **Whitespace:** These tokens represent spaces, tabs, and newline characters, which are typically ignored during parsing.

- **Keywords:** Reserved words of the language, such as: `recipe`, `input`, `output`, `tool_required`, `quantity`, `func`, `if`, `else`, `while`, `for`, `craft`, `log`, `int`, `float`, `char`, `return`.
- **Numbers:** Numeric literals representing integers or decimals, which can be negative (e.g., 123, -456, 3.14).
- **Strings:** Sequences of characters enclosed in double quotes (e.g., `"Hello, world!"`).
- **Identifiers:** Names for variables, functions, recipes, and items (e.g., `Iron_Sword`, `Cobblestone`, `Stick`) that start with a letter or underscore, followed by letters, digits, or underscores.
- **Symbols:** Characters used to structure the code, such as braces `{}`, brackets `[]`, parentheses `()`, colon `:`, comma `,`, and semicolon `;`.
- **Operators:** One or two-character sequences that define operations (e.g., `=`, `+`, `-`, `*`, `/`, `<`, `>`, `==`, etc.).

4 LEXEMES

This refers to the combination of keywords and tokens (specifically, the concrete representation of a token). Lexemes are the actual character sequences that match a pattern defined by a token. In this way, they represent the concrete content of the code. For example, if we analyze this block of code, we can identify a lexeme within it.

```
    recipe iron_pickaxe {
    input: [ (0,0) 1 iron_ingot, (0,1) 1 iron_ingot, (0,2) 1 iron_ingot,
             (1,1) 1 stick,
             (2,1) 1 stick ];
    output: iron_pickaxe;
    tool_required: crafting_table;
    quantity: 1;
}
```

We take a sequence of characters one by one and match them to their corresponding token representation.

5 GENERATIVE GRAMMAR

We must define a context-free grammar to describe the syntax of our Minecraft programming language, specifying valid structures through production rules. In this way, we can establish a generative grammar for recipes, control structures, crafting commands, and console output.

First, we need to define production rules for the creation of recipes and the description of the crafting process. These rules will ensure that recipes are syntactically valid and allow for clear and concise definitions of crafting logic.

Next, we need to define rules for control structures, including conditionals, loops, and assignments. These structures will enable players to implement dynamic and flexible crafting workflows, adapting to different in-game scenarios.

Keyword	Token
recipe	keyword
Iron_Sword	identifier
{	special_character
input	keyword
:	special_character
[	special_character
2	literal
Iron_Ingot	identifier
,	special_character
1	literal
Stick	identifier
]	special_character
output	keyword
:	special_character
Iron_Sword	identifier
tool_required	keyword
:	special_character
"	special_character
Crafting_Table	identifier
"	special_character
quantity	keyword
:	special_character
1	literal
}	special_character

Figure 1: Lexeme table for a recipe definition. Source: Own elaboration

Recipe production rules	
Non-Terminal symbols	Terminal symbols
<program>	<recipe_list>
<recipe_list>	<recipe> <recipe_list> | <recipe>
<recipe>	"recipe" <identifier> "{" <recipe_body> "}"
<recipe_body>	<input_clause> <output_clause> <tool_clause> <quantity_clause>
<input_clause>	"input:" "[" <item_list> "]"
<output_clause>	"output:" <identifier>
<tool_clause>	"tool_required:" <string>
<quantity_clause>	"quantity:" <number>
<item_list>	<item> | <item> "," <item_list>
<item>	<number> <identifier>
<identifier>	[a-zA-Z_][a-zA-Z0-9_]*
<number>	[0-9]+
<string>	"\"" [a-zA-Z0-9_]* "\""

Figure 2: Recipe production rules for the creation of recipes. Source: Own elaboration

Finally, we must establish a way to define functions, which will provide modularity and reusability to the code. Functions will allow players to encapsulate repetitive tasks and complex logic, enhancing the overall maintainability and scalability of their crafting programs.

Control structures production rules	
Non-Terminal symbols	**Terminal symbols**
<statement_list>	<statement> <statement_list> | <statement>
<statement>	<assingment> | <conditional> | <loop> | <function_call> | <craft_command> | <log_command>
<assingment>	<identifier> "=" <expression>
<conditional>	"if" "(" <expression> ")" "{" <statement_list> "}" ["else" "{" <statement_list> "}"]
<loop>	"while" "(" <expression> ")" "{" <statement_list> "}" | "for" "(" <assingment> ";" <expression> ";" <assingment> ")" "{" <statement_list> "}"
<function_call>	<identifier> "(" [<expression_list>] ")"
<craft_command>	"craft" "recipe" <identifier> ";"
<log_command>	"log" "(" <string> ")" ";"
<expression_list>	<expression> | <expression> "," <expression_list>
<expression>	<term> | <term> <operator> <expression>
<term>	<number> | <identifier>
<operator>	"+" | "-" | "*" | "/" | "==" | "!=" | ">" | "<" | ">=" | "<="

Figure 3: Control structures production rules to define conditionals, loops and assingments . Source: Own elaboration

Functions production rules	
Non-Terminal symbols	**Terminal symbols**
<function_definition>	"func" <identifier> "(" [<param_list>] ")" "{" <statement_list> "}"
<param_list>	<identifier> | <identifier> "," <param_list>

Figure 4: Functions production rules to define functions. Source: Own elaboration

Aditionally the full production rules look like this.

Grammar Production Rules	
Non-Terminal symbols	**Terminal symbols**
<S>	<program>
<program>	<function_definition> | <recipe> | <statement>
<function_definition>	"func" <identifier> "(" [<param_list>] ")" "{" <statement_list> "}"
<recipe>	"recipe" <identifier> "{" <recipe_body> "}"
<recipe_body>	<input_clause> ";" <output_clause> ";" <tool_clause> ";" <quantity_clause> ";"
<input_clause>	"input" ":" "[" <item_list> "]"
<output_clause>	"output" ":" <identifier>
<tool_clause>	"tool_required" ":" <identifier>
<quantity_clause>	"quantity" ":" <number>
<item_list>	<item> { "," <item> }
<item>	"(" <number> "," <number> ")" <number> <identifier>
<statement_list>	{ <statement> }
<statement>	<assignment>| <conditional>| <loop>| <log_command>| <craft_command>
<assingment>	<identifier> "=" <expression> ";"
<conditional>	"if" "(" <expression> ")" "{" <statement_list> "}" ["else" "{" <statement_list> "}"]
<loop>	"while" "(" <expression> ")" "{" <statement_list> "}" | "for" "(" <assignment> <expression> ";" <assignment> ")" "{" <statement_list> "}" <assingment> ")" "{" <statement_list> "}"
<log_command>	"log" "(" <expression> ")" ";"
<craft_command>	"craft" "recipe" <identifier> ";"
<expression>	<term> { <operator> <term> }
<term>	<number> | <string> | <identifier> | "(" <expression> ")"
<operator>	"=" | "+" | "-" | "*" | "/" | "<" | ">" | "=="
<identifier>	[a-zA-Z_][a-zA-Z0-9_]*
<number>	[0-9]+
<string>	"\"" [a-zA-Z0-9_]* "\""

Figure 5: General production rules to define functions. Source: Own elaboration

Part III
LEXICAL ANALYZER

Lexical analysis is one of the first and most critical stages in the implementation of a compiler for our Minecraft programming language. Its role is essential in transforming source code into a tokenized representation that can be processed by subsequent stages of the compiler, such as the parser. The lexer operates by breaking down the source code into tokens, adhering to the token patterns established for our programming language. At the same time, it performs error detection, identifying lexical issues like invalid characters or malformed symbols. This process ensures that the input code is not only structured but also adheres to the foundational rules of the language, paving the way for a smoother syntactic analysis.

6 LEXEME

This block of code defines a set of tokken patterns for our lexical analysis in the contest of our Minecraft programming language. Using the "re" module for Python, here we define regular expressions for recognizing different types of token within a source of code. Through the several tuples in TOKEN_PATTERNS we can specify a token type and its associated regular expression pattern.

We use re.compile for each pattern in order to optimize some processes and operations for the lexical analyzer.

```
import re

TOKEN_PATTERNS = [
    ("COMMENT", re.compile(r"//.*")),
    ("WHITESPACE", re.compile(r"\s+")),
    ("KEYWORD", re.compile(r"\b(recipe|input|output|tool_required|quantity|func|if
        |else|while|for|craft|log|int|float|char|return)\b")),
    ("NUMBER", re.compile(r"(?<!\d)-?\d+(\.\d+)?\b")),  # Ajustado para evitar
        capturar '-' en expresiones sin espacio
    ("STRING", re.compile(r'\"[^\"]*\"')),
    ("IDENTIFIER", re.compile(r"\b[a-zA-Z_][a-zA-Z0-9_]*\b")),
    ("SYMBOL", re.compile(r"[{}\[\]:,;()]")),
    ("OPERATOR", re.compile(r"[=+*/<>!%\-]{1,2}")),
]
```

6.1 Keyword Token

This pattern is designed to match keywords in the language; these are reserved words like recipe, input, if, while and return. In the regular expression we have:

- **\b:** This ensures that there are whole words.
- **|:** This is an alternation of the pattern, it works similar than a OR operator.

6.2 Number Token

This pattern matches numbers, both integers and floating-point values, using \d+ to match digits and optionally matches a decimal point followed by more digits.

- **\b:** This a word boundary that ensures the number is separate from other characters
- **\d+:** This one matches one or more integer digits.

- **( ):** This means that what's inside is an optional group.
- **⁝** This matches a strict decimal point.
- **?:** Makes the entire decimal part optional.

6.3 String Token

This pattern matches strings (Any sequence of characters that are not double quotes) enclosed in double quotes, ensuring the string is correctly parsed.

- **\":** Matches a literal double quote to anounce the start of a string object.
- **ˆ\":** This part matches any sequence of characters that are not a double quote, using the ˆsymbol to match anything except the double quote that comes after.
- **:** This means that any character or number will be allowed at least 0 times or the times that it is required.
- **\":** This one matches the closing double quote for the string.

6.4 Identifier Token

This regular expression matches identifiers used for variable names, function names or other user-defined names. It must begin with a letter or undescore and can contain letters, digits or as said before, underscores.

- **\b:** This word boundary ensures the identifier is not a part of another word.
- **[a-zA-Z_]:** This matches the first character of the identifier, which must be either a letter or an underscore.
- **[a-zA-Z0-9_]*:** Matches the rest of the identifier, which can contain letters, digits, and underscores, using * to define that the pattern can be repeated zero or more times.

6.5 Symbol Token

This pattern matches symbols that are typically used for structuring the language, such as delimiters, brackets and other special characters that separate or organize code elements.

- **[]:** This is a character class that matches any of the characters inside.
- **, [], :, ...:** These are specific symbols that our pattern will match.

6.6 Operator Token

This pattern matches operators such as arithmetic, coparison and logical operators, along with the assignment operators like = and +=.

- **[=+*/<>!...]:** This matches any single operator.
- **{1, 2}:** This quantifier allows for 1 or 2 characters, so the pattern can match single characters like +, -, or *, and can match two character operators like ==, != or ¿=.

6.7 Comment Token

This pattern is made with the purpose of matching single line comments, witch begin with //. Comments are ignored during lexical analysis and don't affect the program's execution.

- **//:** Matches the literal double slash // used to specify single line comments.
- **.*:** Matches any number of characters after the //

6.8 Whitespace Token

This regular expression matches whitespace in the code thinking in the importance of recognize white spaces to correctly separate tokens and maintain the structure of the code.

- **\s+:** This matches one or more whitespace characters, including spaces, tabs, and newlines.

7 LEXER

This block of code is the main analyzer, it implements the lexical analyzer and takes the source code as input and divides it into tokens in order to identify these tokens in the source code. It cares about lexical errors in the source code as well, raising exceptions. It imports the TOKEN_PATTERNS tuple list from Lexeme class and imports the LexicalError class.

```
1
2       from .lexeme import TOKEN_PATTERNS
3       from .lexical_error import LexicalError
4       import re
5
6       class Lexer:
7           def __init__(self, code):
8               self.code = code
9               self.position = 0
10              self.tokens = []
11
12          def tokenize(self):
13              while self.position < len(self.code):
14                  match_found = self._match_next_token()
15                  if not match_found:
16                      self._handle_invalid_character()
17              return self.tokens
18
19          def _match_next_token(self):
20              for token_name, pattern in TOKEN_PATTERNS:
21                  match = pattern.match(self.code, self.position)
22                  if match:
23                      lexeme = match.group()
24                      position = self.position
25                      self.position = match.end()
26
27                      self._validate_token(token_name, lexeme, position)
28
29                      if token_name not in ("WHITESPACE", "COMMENT"):
30                          self.tokens.append((token_name, lexeme, position))
31                      return True
32              return False
33
34          def _validate_token(self, token_name, lexeme, position):
35              if token_name == "STRING" and not lexeme.endswith('"'):
36                  raise LexicalError("Unterminated string literal", position)
37              elif token_name == "NUMBER":
```

```
38                  try:
39                      float(lexeme)
40                  except ValueError:
41                      raise LexicalError("Number literal overflow or invalid format",
                            position)
42
43          def _handle_invalid_character(self):
44              current_char = self.code[self.position]
45              position = self.position
46
47              if current_char.isprintable():
48                  raise LexicalError(f"Invalid character '{current_char}'", position)
49              else:
50                  raise LexicalError("Non-printable character detected", position)
```

7.1 Constructor

__init__(self, code): Initializes the lexer with the source code `code` to be analyzed. It sets the initial position to 0 and creates an empty list to store the tokens found in the code.

7.2 tokenize

tokenize(self): This is the main function that iterates over the source code. For each position, it calls `_match_next_token()` to try to match a valid token. If no valid token is found at the current position, it calls `_handle_invalid_character()`. Finally, it returns the list of tokens collected.

7.3 _match_next_token

_match_next_token(self): Iterates through the defined token patterns in `TOKEN_PATTERNS` and tries to match a token at the current position in the code. If a match is found:

- The lexeme is extracted.
- The current position is updated to the end of the matched lexeme.
- The token is validated using `_validate_token()`.
- If the token is not a `WHITESPACE` or `COMMENT`, it is appended to the tokens list (together with its type and starting position).

If no token matches, the method returns `False`.

7.4 _validate_token

_validate_token(self, token_name, lexeme, position): Validates tokens with specific requirements:

- For a `STRING` token, if the lexeme does not end with a quotation mark (`"`), it raises a lexical error indicating an unterminated string.
- For a `NUMBER` token, it attempts to convert the lexeme to a float. If the conversion fails (due to overflow or invalid format), it raises a lexical error.

7.5 _handle_invalid_character

_handle_invalid_character(self): When no valid token is matched at the current position, this method retrieves the current character and its position. If the character is printable, it raises a lexical error indicating an invalid character; otherwise, it raises an error indicating a non-printable character was detected.

8 LEXICAL ERROR

This code block defines a custom exception class to specify if the program finds a lexical error at some point of the source code.

```
1      class LexicalError(Exception):
2      def __init__(self, message, position):
3          super().__init__(f"Lexical error at position {position}: {message}")
4          self.position = position
```

This class inherits from Python's exception class, allowing it to be used as a custom exception for handling our lexical errors, it uses its constructor to take the error message and the position of that specific error.

The **super().__init__(f'Lexical error at position {position}: {message}** line of code calls the constructor method of the Exception class with a formated string that includes the position and message, ensuring that the exception has detailed information about where the error occurred and what the error was.

And finally it assigns the position argument to an instance variable allowing the error to be accessed later at debugging time.

Part IV
SYNTACTIC ANALYZER

Syntactic analysis is a critical stage in the compilation process of our Minecraft programming language. In this phase, the sequence of tokens generated by the lexer is transformed into a hierarchical structure—typically an Abstract Syntax Tree (AST). The parser examines the tokens to ensure that they conform to the grammatical rules defined for the language, verifying that constructs such as function definitions, recipes, assignments, and control structures are properly formed.

By structuring the token stream into an AST, the parser not only validates the syntax but also organizes the program's components in a way that facilitates further processing by the semantic analyzer and interpreter. Additionally, the parser is responsible for detecting and reporting syntax errors, such as unexpected tokens or missing delimiters, thus preventing invalid code from progressing to later stages. This organized, tree-like representation of the source code is essential for ensuring that the code is both syntactically correct and ready for semantic analysis.

9 PARSER

Syntactic analysis is a crucial stage in the compilation process of our Minecraft programming language. The parser takes the stream of tokens generated by the lexer and organizes them into a hierarchical structure—usually an Abstract Syntax Tree (AST). This structured representation reflects the syntactic constructs of the language, such as function definitions, recipes, statements, and expressions. By verifying that the token sequence conforms to the grammar rules, the parser not only validates the code's syntax but also prepares the program for semantic analysis and further evaluation. In case of any syntactic anomalies, such as missing delimiters or unexpected tokens, the parser raises appropriate errors, thereby ensuring that only well-formed code proceeds to the next compilation phases.

```
1       from .syntax_error import SyntaxError
2
3       class Parser:
4           def __init__(self, tokens):
5               self.tokens = tokens
6               self.position = 0
7
8           def parse(self):
9               nodes = []
10              while self.position < len(self.tokens):
11                  if self._peek_lexeme() == "func":
12                      nodes.append(self._parse_function_definition())
13                  elif self._peek_lexeme() == "recipe":
14                      nodes.append(self._parse_recipe())
15                  else:
16                      nodes.append(self._parse_statement())
17              return nodes
18
19          # -------------------- FUNCTIONS --------------------
20          def _parse_function_definition(self):
21              self._consume("KEYWORD", "func")
22              name = self._consume("IDENTIFIER")
23              self._consume("SYMBOL", "(")
24              params = self._parse_parameter_list()
25              self._consume("SYMBOL", ")")
26              self._consume("SYMBOL", "{")
27              body = self._parse_statement_list()
28              self._consume("SYMBOL", "}")
29              return {
30                  "node_type": "function_definition",
```

```
            "name": name,
            "params": params,
            "body": body
        }

    def _parse_parameter_list(self):
        params = []
        if self._peek_lexeme() != ")":
            params.append(self._consume("IDENTIFIER"))
            while self._peek_lexeme() == ",":
                self._consume("SYMBOL", ",")
                params.append(self._consume("IDENTIFIER"))
        return params

    # ---------------------- RECIPES ----------------------
    def _parse_recipe(self):
        self._consume("KEYWORD", "recipe")
        name = self._consume("IDENTIFIER")
        self._consume("SYMBOL", "{")
        recipe_body = self._parse_recipe_body()
        self._consume("SYMBOL", "}")
        return {
            "node_type": "recipe",
            "name": name,
            "input": recipe_body["input"],
            "output": recipe_body["output"],
            "tool_required": recipe_body["tool_required"],
            "quantity": recipe_body["quantity"]
        }

    def _parse_recipe_body(self):
        input_clause = self._parse_input_clause()
        self._consume("SYMBOL", ";")
        output_clause = self._parse_output_clause()
        self._consume("SYMBOL", ";")
        tool_clause = self._parse_tool_clause()
        self._consume("SYMBOL", ";")
        quantity_clause = self._parse_quantity_clause()
        self._consume("SYMBOL", ";")
        return {
            "input": input_clause,
            "output": output_clause,
            "tool_required": tool_clause,
            "quantity": quantity_clause
        }

    def _parse_input_clause(self):
        self._consume("KEYWORD", "input")
        self._consume("SYMBOL", ":")
        self._consume("SYMBOL", "[")
        items = self._parse_item_list()
        self._consume("SYMBOL", "]")
        return items

    def _parse_output_clause(self):
        self._consume("KEYWORD", "output")
        self._consume("SYMBOL", ":")
        return self._consume("IDENTIFIER")

    def _parse_tool_clause(self):
        self._consume("KEYWORD", "tool_required")
        self._consume("SYMBOL", ":")
        return self._consume("IDENTIFIER")

    def _parse_quantity_clause(self):
```

```
96              self._consume("KEYWORD", "quantity")
97              self._consume("SYMBOL", ":")
98              return self._consume("NUMBER")
99
100         def _parse_item_list(self):
101             items = [self._parse_item()]
102             while self._peek_lexeme() == ",":
103                 self._consume("SYMBOL", ",")
104                 items.append(self._parse_item())
105             return items
106
107         def _parse_item(self):
108             self._consume("SYMBOL", "(")
109             row = self._consume("NUMBER")
110             self._consume("SYMBOL", ",")
111             col = self._consume("NUMBER")
112             self._consume("SYMBOL", ")")
113             quantity = self._consume("NUMBER")
114             material = self._consume("IDENTIFIER")
115             return {
116                 "position": (row, col),
117                 "quantity": quantity,
118                 "material": material
119             }
120
121         # ---------------------- STATEMENTS AND CONTROL STRUCTURES
                ----------------------
122         def _parse_statement_list(self):
123             stmts = []
124             while self.position < len(self.tokens) and self._peek_lexeme() != "}":
125                 stmts.append(self._parse_statement())
126             return stmts
127
128         def _parse_statement(self):
129             token_type, lexeme, _ = self._peek()
130             if token_type == "IDENTIFIER":
131                 if self._lookahead_is_operator("="):
132                     return self._parse_assignment()
133                 else:
134                     raise SyntaxError("Unexpected statement: an unassigned identifier
                            was found.", self._current_position())
135             elif token_type == "KEYWORD":
136                 if lexeme == "if":
137                     return self._parse_conditional()
138                 elif lexeme in ("while", "for"):
139                     return self._parse_loop()
140                 elif lexeme == "log":
141                     return self._parse_log_command()
142                 elif lexeme == "craft":
143                     return self._parse_craft_command()
144                 else:
145                     raise SyntaxError(f"Invalid statement starting with '{lexeme}'",
                            self._current_position())
146             else:
147                 raise SyntaxError("Invalid statement", self._current_position())
148
149         def _parse_assignment(self, require_semicolon=True):
150             identifier = self._consume("IDENTIFIER")
151             self._consume("OPERATOR", "=")
152             expr = self._parse_expression()
153             if require_semicolon:
154                 self._consume("SYMBOL", ";")
155             return {
156                 "node_type": "assignment",
157                 "identifier": identifier,
```

```
            "expression": expr
        }

    def _parse_conditional(self):
        self._consume("KEYWORD", "if")
        self._consume("SYMBOL", "(")
        condition = self._parse_expression()
        self._consume("SYMBOL", ")")
        self._consume("SYMBOL", "{")
        then_branch = self._parse_statement_list()
        self._consume("SYMBOL", "}")
        else_branch = None
        if self._peek_lexeme() == "else":
            self._consume("KEYWORD", "else")
            self._consume("SYMBOL", "{")
            else_branch = self._parse_statement_list()
            self._consume("SYMBOL", "}")
        return {
            "node_type": "conditional",
            "condition": condition,
            "then_branch": then_branch,
            "else_branch": else_branch
        }

    def _parse_loop(self):
        if self._peek_lexeme() == "while":
            self._consume("KEYWORD", "while")
            self._consume("SYMBOL", "(")
            condition = self._parse_expression()
            self._consume("SYMBOL", ")")
            self._consume("SYMBOL", "{")
            body = self._parse_statement_list()
            self._consume("SYMBOL", "}")
            return {
                "node_type": "while_loop",
                "condition": condition,
                "body": body
            }
        elif self._peek_lexeme() == "for":
            self._consume("KEYWORD", "for")
            self._consume("SYMBOL", "(")
            init = self._parse_assignment(require_semicolon=False)
            self._consume("SYMBOL", ";")
            condition = self._parse_expression()
            self._consume("SYMBOL", ";")
            post = self._parse_assignment(require_semicolon=False)
            self._consume("SYMBOL", ")")
            self._consume("SYMBOL", "{")
            body = self._parse_statement_list()
            self._consume("SYMBOL", "}")
            return {
                "node_type": "for_loop",
                "init": init,
                "condition": condition,
                "post": post,
                "body": body
            }

    def _parse_log_command(self):
        self._consume("KEYWORD", "log")
        self._consume("SYMBOL", "(")
        expr = self._parse_expression()
        self._consume("SYMBOL", ")")
        self._consume("SYMBOL", ";")
        return {
```

```
223                 "node_type": "log",
224                 "expression": expr
225             }
226 
227         def _parse_craft_command(self):
228             self._consume("KEYWORD", "craft")
229             self._consume("KEYWORD", "recipe")
230             recipe_name = self._consume("IDENTIFIER")
231             self._consume("SYMBOL", ";")
232             return {
233                 "node_type": "craft_command",
234                 "recipe_name": recipe_name
235             }
236 
237         def _parse_expression(self):
238             # Construye un árbol binario para expresiones
239             left = self._parse_term()
240             while self.position < len(self.tokens) and self._peek_type() == "OPERATOR"
                    :
241                 op = self._consume("OPERATOR")
242                 right = self._parse_term()
243                 left = {
244                     "node_type": "binary_expression",
245                     "operator": op,
246                     "left": left,
247                     "right": right
248                 }
249             return left
250 
251         def _parse_term(self):
252             token_type, lexeme, _ = self._peek()
253             if token_type == "NUMBER":
254                 value_str = self._consume("NUMBER")
255                 # Convertir el valor a número (int o float) según corresponda
256                 if "." in value_str:
257                     try:
258                         value = float(value_str)
259                     except ValueError:
260                         raise SyntaxError("Invalid float number: " + value_str, self.
                                _current_position())
261                 else:
262                     try:
263                         value = int(value_str)
264                     except ValueError:
265                         raise SyntaxError("Invalid integer number: " + value_str, self
                                ._current_position())
266                 return {"node_type": "literal", "value": value}
267             elif token_type == "STRING":
268                 value = self._consume("STRING")
269                 return {"node_type": "literal", "value": value}
270             elif token_type == "IDENTIFIER":
271                 name = self._consume("IDENTIFIER")
272                 return {"node_type": "identifier", "name": name}
273             elif token_type == "SYMBOL" and lexeme == "(":
274                 self._consume("SYMBOL", "(")
275                 expr = self._parse_expression()
276                 self._consume("SYMBOL", ")")
277                 return expr
278             else:
279                 raise SyntaxError("Invalid term in the expression", self.
                        _current_position())
280 
281         # --------------------- AUX FUNCTIONS ---------------------
282         def _consume(self, expected_type, expected_lexeme=None):
283             if self.position >= len(self.tokens):
```

```
284                 raise SyntaxError("Unexpected end of input", self._current_position())
285             token_type, lexeme, position = self.tokens[self.position]
286             if token_type != expected_type or (expected_lexeme is not None and lexeme
                    != expected_lexeme):
287                 expected_info = f"{expected_type} '{expected_lexeme}'" if
                        expected_lexeme else expected_type
288                 raise SyntaxError(f"Expected {expected_info}, found {token_type} '{
                        lexeme}'", position)
289             self.position += 1
290             return lexeme
291
292         def _peek(self):
293             if self.position >= len(self.tokens):
294                 return None, None, self.position
295             return self.tokens[self.position]
296
297         def _peek_type(self):
298             return self._peek()[0]
299
300         def _peek_lexeme(self):
301             return self._peek()[1]
302
303         def _current_position(self):
304             if self.position < len(self.tokens):
305                 return self.tokens[self.position][2]
306             return self.tokens[-1][2] if self.tokens else 0
307
308         def _lookahead_is_operator(self, op):
309             if self.position + 1 < len(self.tokens):
310                 next_token = self.tokens[self.position + 1]
311                 return next_token[0] == "OPERATOR" and next_token[1] == op
312             return False
```

9.1 Function Definition Rule

function_definition → `func` **identifier** `(` [**parameter_list**] `)` { **statement_list** }

This rule defines a function by requiring the keyword `func`, followed by an identifier (the function name), an optional parameter list enclosed in parentheses, and a block of statements enclosed in braces. The parser ensures that all these elements appear in the proper order.

9.2 Recipe Rule

recipe → `recipe` **identifier** { **recipe_body** }

Recipes are defined with the keyword `recipe`, followed by a name and a body enclosed in braces. The recipe body consists of several clauses (input, output, tool_required, quantity) separated by semicolons, which specify the components needed for crafting an item.

9.3 Statement Rule

statement → **assignment** | **conditional** | **loop** | **log_command** | **craft_command**

This rule enumerates the possible types of statements in the language. The parser differentiates between assignments, conditionals, loops, log commands, and craft commands, processing each according to its specific syntax.

9.4 Expression and Term Rules

expression → **term** { **operator term** }

term → **number** | **string** | **identifier** | `(` **expression** `)`

These rules define how expressions are constructed. An expression consists of one or more terms connected by operators. A term can be a literal number, a string, an identifier, or another expression enclosed in parentheses. This recursive definition enables the parsing of complex arithmetic or logical expressions.

9.5 Additional Definitions

The parser relies on several fundamental tokens defined in our lexical analysis:

- **Identifiers:** Names for functions, recipes, variables, and items (e.g., `Iron_Sword`, `wheat`).
- **Numbers:** Numeric literals, which can be negative and support both integer and floating-point formats.
- **Strings:** Sequences of characters enclosed in double quotes.
- **Symbols:** Structural tokens such as braces {}, brackets `[]` and parentheses `()`.
- **Operators:** Tokens representing arithmetic and logical operations (e.g., `+`, `-`, `*`, `/`, `==`).

Together, these rules provide a complete generative grammar for our parser, ensuring that the source code is organized into an AST that accurately reflects the program's syntactic structure, ready for subsequent semantic analysis and interpretation.

10 SYNTAX ERROR

This code block defines a custom exception class to indicate when a syntax error is encountered in the source code.

```
class SyntaxError(Exception):
    def __init__(self, message, position):
        super().__init__(f"Syntax error at position {position}: {message}")
        self.position = position
```

This class inherits from Python's built-in `Exception` class, allowing it to be used as a custom exception for handling syntax errors in our compiler. Its constructor takes two parameters: an error message and the position in the source code where the error occurred.

The line **super().__init__(f"Syntax error at position {position}: {message}")** calls the constructor of the base `Exception` class with a formatted string that includes both the error message and the position, ensuring that detailed information about the syntax error is available during debugging. Finally, the `position` parameter is assigned to an instance variable so that the error position can be accessed later if needed.

Part V
SEMANTIC ANALYZER

Semantic analysis is a crucial stage in the compilation process of our Minecraft programming language. In this phase, the Abstract Syntax Tree (AST) generated by the parser is traversed and examined to ensure that the code adheres to the semantic rules of the language. The semantic analyzer verifies that identifiers are correctly declared and used, that operations are applied to compatible types, and that constructs such as recipes, functions, and assignments make logical sense within the language context.

By checking the AST for semantic correctness, the analyzer can detect errors that the syntactic analysis may have missed, such as undeclared variables, type mismatches, or invalid index values in recipes. Additionally, it can annotate the AST with useful semantic information for later stages of compilation or interpretation. This process is essential for ensuring that the program is not only syntactically correct but also semantically sound and ready for execution.

11 SEMANTIC ANALYZER

Semantic analysis is the stage in the compiler where the Abstract Syntax Tree (AST) produced by the parser is traversed and examined to ensure that the source code adheres to the semantic rules of our Minecraft programming language. This phase verifies that variables and functions are properly declared before use, that operations are applied to compatible types, and that constructs such as recipes meet specific constraints (for example, valid index ranges). By checking the AST for semantic consistency, the semantic analyzer helps prevent errors that cannot be caught during syntactic analysis.

```
from interpreter.semantic_analyzer.semantic_error import SemanticError

class SemanticAnalyzer:
    def __init__(self):
        self.symbol_table = {}

    def analyze(self, abstract_syntax_tree):
        for node in abstract_syntax_tree:
            self.visit(node)
        return abstract_syntax_tree

    def visit(self, node):
        if node is None:
            return None
        node_type = node.get("node_type")
        % If the node is a recipe, use the specialized method.
        if node_type == "recipe":
            return self.visit_recipe(node)
        method_name = "visit_" + node_type if node_type else "generic_visit"
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        if node is None:
            return None
        for key, value in node.items():
            if isinstance(value, list):
                for item in value:
                    if isinstance(item, dict):
                        self.visit(item)
            elif isinstance(value, dict):
                self.visit(value)
        return node

    def visit_assignment(self, node):
```

```
36          identifier = node.get("identifier")
37          self.symbol_table[identifier] = True
38          self.visit(node.get("expression"))
39          return node
40
41      def visit_identifier(self, node):
42          name = node.get("name")
43          if name not in self.symbol_table:
44              raise SemanticError(f"Undefined variable '{name}'")
45          return node
46
47      def visit_function_definition(self, node):
48          self.symbol_table[node.get("name")] = True
49          for param in node.get("params", []):
50              self.symbol_table[param] = True
51          for stmt in node.get("body", []):
52              self.visit(stmt)
53          return node
54
55      def visit_recipe(self, node):
56          % Validate positions if the required tool is crafting_table.
57          if node.get("tool_required", "").lower() == "crafting_table":
58              input_items = node.get("input", [])
59              for item in input_items:
60                  try:
61                      row = int(item["position"][0])
62                      col = int(item["position"][1])
63                  except Exception as e:
64                      raise SemanticError(f"Invalid position format for item: {item}")
65                  if row < 0 or row > 2 or col < 0 or col > 2:
66                      raise SemanticError(f"Invalid position {item['position']} for item
                            '{item.get('material', 'unknown')}'. Indices must be between 0
                            and 2.")
67          return self.generic_visit(node)
```

11.1 Constructor

__init__(self): Initializes the semantic analyzer by creating an empty symbol table. The symbol table is used to track the declarations of identifiers (such as variables and function names) during the analysis.

11.2 Analyze

analyze(self, abstract_syntax_tree): This method iterates through each node in the AST and invokes the `visit()` method for further semantic processing. It returns the AST after it has been fully analyzed. This phase ensures that every part of the code is semantically valid before moving on to evaluation.

11.3 Visit

visit(self, node): This is the dispatcher method that examines the `node_type` of each AST node and calls the corresponding specialized method (e.g., `visit_function_definition`, `visit_recipe`, etc.). If the node does not have a specific method, it falls back to `generic_visit()`. This mechanism allows modular semantic checks based on the node type.

11.4 Generic Visit

generic_visit(self, node): For nodes that do not require special semantic checks, this method recursively traverses all sub-nodes (whether contained in lists or as single dictionaries) by calling `visit()`

on each. This ensures that every part of the AST is visited and validated.

11.5 Visit Assignment

visit_assignment(self, node): This method handles assignment nodes. It records the assigned identifier in the symbol table and then visits the expression on the right-hand side. This ensures that later references to the identifier will be recognized as valid.

11.6 Visit Identifier

visit_identifier(self, node): When an identifier is encountered, this method checks if it exists in the symbol table. If the identifier is not found, it raises a semantic error indicating that the variable is undefined.

11.7 Visit Function Definition

visit_function_definition(self, node): This method processes function definition nodes. It adds the function name and its parameters to the symbol table, then recursively visits all statements within the function body. This ensures that the function is correctly declared and that its internal statements are semantically valid.

11.8 Visit Recipe

visit_recipe(self, node): This method handles recipe nodes. It performs a crucial semantic check: if the recipe requires a crafting table (i.e., the `tool_required` field is `crafting_table`), it validates that every item in the `input` clause has positions with indices between 0 and 2. If any position falls outside this range or is formatted incorrectly, a `SemanticError` is raised. After performing this check, the method continues to traverse any sub-nodes using `generic_visit()`.

In this way, the semantic analyzer ensures that the source code not only follows the syntactic rules but also adheres to the deeper semantic rules of our language, paving the way for successful interpretation or compilation.

12 SEMANTIC ERROR

This code block defines a custom exception class to indicate when a semantic error is encountered during the analysis of the source code.

```
1 class SemanticError(Exception):
2     def __init__(self, message):
3         super().__init__(f"Semantic Error: {message}")
```

This class inherits from Python's built-in `Exception` class, allowing it to be used as a custom exception for handling semantic errors in our compiler. Its constructor takes an error message as a parameter and passes a formatted string containing the message to the parent class's constructor. This ensures that the exception provides clear information about the nature of the semantic error, aiding in debugging and error reporting.

Part VI
INTERPRETATION AND EVALUATION

Interpretation and evaluation represent the final stage in the compilation process of our Minecraft programming language. After the source code has been transformed into an Abstract Syntax Tree (AST) by the parser and validated by the semantic analyzer, the interpreter takes over to execute the program.

In this phase, the interpreter traverses the AST and evaluates expressions, executes function calls, and processes control structures such as conditionals and loops. For instance, when a recipe is interpreted, the interpreter not only checks that the code is syntactically and semantically correct, but also carries out the necessary actions—such as updating the crafting table with the required items based on the recipe's input.

During evaluation, operations such as arithmetic calculations, logical comparisons, and variable assignments are performed, converting the static AST into dynamic, executable behavior. Moreover, any runtime errors or anomalies encountered during this phase are captured and reported with detailed messages, ensuring that issues are clearly communicated for debugging purposes. This organized process of interpretation and evaluation bridges the gap between source code and its interactive execution, enabling our language to respond to user-defined commands and produce tangible results in the application.

13 INTERPRETER

This code block implements the interpreter for our Minecraft programming language. After the parser has generated an Abstract Syntax Tree (AST) from the source code, the interpreter traverses and evaluates the AST. Its purpose is to execute the code by processing each AST node, updating the global environment, evaluating expressions, and handling control structures. In doing so, the interpreter transforms the static representation of the program into dynamic behavior.

```
class Interpreter:
    def __init__(self):
        # Global environment to store variables, functions, recipes, etc.
        self.global_env = {}

    def run(self, abstract_syntax_tree):
        """
        Executes the Abstract Syntax Tree (AST), which is assumed to be a list of AST
            nodes.
        """
        for node in abstract_syntax_tree:
            self.visit(node)

    def visit(self, node):
        """
        Dispatches the node to the appropriate visitor method based on its 'node_type
            '.
        If no specific visitor method is found, it calls generic_visit().
        """
        node_type = node.get("node_type")
        method_name = "visit_" + node_type
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        """
        This method is called if no visitor method is defined for a node type.
```

```
26          It raises an exception indicating that the node type is not supported.
27          """
28          raise Exception("No visitor method defined for node type: " + str(node.get("
                node_type")))
29
30      # ------------------ Visitor Methods for AST Nodes ------------------
31
32      def visit_recipe(self, node):
33          """
34          Processes a recipe node.
35          Expected node structure:
36            - "name": recipe name.
37            - "input": list of items (each item is a dict with "position", "quantity",
                  and "material").
38            - "output": name of the resulting item.
39            - "tool_required": required tool.
40            - "quantity": output quantity.
41          """
42          print(f"Processing recipe: {node['name']}")
43          print("Input materials:")
44          for item in node["input"]:
45              print(f"  Place {item['quantity']} of {item['material']} at position {item
                    ['position']}")
46          print(f"Output: {node['output']}")
47          print(f"Required tool: {node['tool_required']}")
48          print(f"Quantity: {node['quantity']}")
49          return None
50
51      def visit_function_definition(self, node):
52          """
53          Processes a function definition node.
54          Expected node structure:
55            - "name": function name.
56            - "params": list of parameters.
57            - "body": list of AST nodes representing the function body.
58          """
59          # Store the function definition in the global environment.
60          self.global_env[node["name"]] = node
61          print(f"Defined function: {node['name']}")
62          return None
63
64      def visit_assignment(self, node):
65          """
66          Processes an assignment node.
67          Expected node structure:
68            - "identifier": variable name.
69            - "expression": AST node representing the expression.
70          """
71          value = self.visit(node["expression"])
72          self.global_env[node["identifier"]] = value
73          return value
74
75      def visit_conditional(self, node):
76          """
77          Processes a conditional node.
78          Expected node structure:
79            - "condition": AST node for the condition.
80            - "then_branch": list of AST nodes for the 'if' block.
81            - "else_branch": (optional) list of AST nodes for the 'else' block.
82          """
83          condition = self.visit(node["condition"])
84          if condition:
85              for stmt in node["then_branch"]:
86                  self.visit(stmt)
87          elif node.get("else_branch") is not None:
```

```
            for stmt in node["else_branch"]:
                self.visit(stmt)
        return None

    def visit_while_loop(self, node):
        """
        Processes a while loop node.
        Expected node structure:
          - "condition": AST node for the loop condition.
          - "body": list of AST nodes for the loop body.
        """
        while self.visit(node["condition"]):
            for stmt in node["body"]:
                self.visit(stmt)
        return None

    def visit_for_loop(self, node):
        """
        Processes a for loop node.
        Expected node structure:
          - "init": AST node for initialization.
          - "condition": AST node for the loop condition.
          - "post": AST node for the post-loop assignment.
          - "body": list of AST nodes for the loop body.
        """
        self.visit(node["init"])
        while self.visit(node["condition"]):
            for stmt in node["body"]:
                self.visit(stmt)
            self.visit(node["post"])
        return None

    def visit_log(self, node):
        """
        Processes a log command node.
        Expected node structure:
          - "expression": AST node to be evaluated and logged.
        """
        value = self.visit(node["expression"])
        print(f"LOG: {value}")
        return value

    def visit_craft_command(self, node):
        """
        Processes a craft command node.
        Expected node structure:
          - "recipe_name": name of the recipe to craft.
        """
        recipe_name = node["recipe_name"]
        print(f"Craft command invoked for recipe: {recipe_name}")
        return None

    def visit_binary_expression(self, node):
        """
        Processes a binary expression node.
        Expected node structure:
          - "operator": operator (e.g., '+', '-', etc.).
          - "left": left operand (AST node).
          - "right": right operand (AST node).
        """
        left = self.visit(node["left"])
        right = self.visit(node["right"])
        op = node["operator"]

        if op == "+":
```

```
153             try:
154                 return float(left) + float(right)
155             except (ValueError, TypeError):
156                 return str(left) + str(right)
157         elif op == "-":
158             return float(left) - float(right)
159         elif op == "*":
160             return float(left) * float(right)
161         elif op == "/":
162             return float(left) / float(right)
163         elif op == "==":
164             return left == right
165         elif op == "!=":
166             return left != right
167         elif op == "<":
168             return float(left) < float(right)
169         elif op == ">":
170             return float(left) > float(right)
171         elif op == "<=":
172             return float(left) <= float(right)
173         elif op == ">=":
174             return float(left) >= float(right)
175         else:
176             raise Exception("Unsupported operator: " + op)
177 
178     def visit_literal(self, node):
179         """
180         Processes a literal node (number or string).
181         Expected node structure:
182           - "value": the literal value.
183         """
184         value = node["value"]
185         if isinstance(value, str) and value.startswith('"') and value.endswith('"'):
186             return value[1:-1]
187         try:
188             return float(value)
189         except ValueError:
190             return value
191 
192     def visit_identifier(self, node):
193         """
194         Processes an identifier node.
195         Expected node structure:
196           - "name": variable name.
197         """
198         name = node["name"]
199         if name in self.global_env:
200             return self.global_env[name]
201         else:
202             raise Exception("Undefined variable: " + name)
```

13.1 Constructor

__init__(self): Initializes the interpreter by creating an empty global environment to store variables, functions, recipes, and other necessary data. This environment is essential for keeping track of the state during the evaluation of the AST.

13.2 run

run(self, abstract_syntax_tree): This method executes the given Abstract Syntax Tree (AST) by iterating through its nodes and processing each one using the `visit()` method. It serves as the entry point for the interpreter, transforming the AST into executable behavior.

13.3 visit

visit(self, node): Acts as a dispatcher that selects the appropriate visitor method for each AST node based on the node's `node_type`. If no specific visitor method is found, it calls `generic_visit()`, which raises an exception to indicate an unsupported node type.

13.4 Visitor Methods

The interpreter defines specific visitor methods for different AST node types:

- **visit_recipe(self, node):** Processes a recipe node by printing details of the recipe, including input materials, output, required tool, and quantity.
- **visit_function_definition(self, node):** Registers a function in the global environment and prints a confirmation message.
- **visit_assignment(self, node):** Evaluates an assignment by computing the expression on the right-hand side and storing the resulting value in the global environment.
- **visit_conditional(self, node):** Evaluates a conditional statement, executing the 'if' branch if the condition is true, and the 'else' branch if provided.
- **visit_while_loop(self, node):** Processes a while loop by repeatedly evaluating its condition and executing the loop body as long as the condition holds true.
- **visit_for_loop(self, node):** Processes a for loop by executing the initialization, repeatedly evaluating the condition, executing the body, and performing the post-loop assignment.
- **visit_log(self, node):** Evaluates a log command and prints the result.
- **visit_craft_command(self, node):** Processes a craft command by printing the recipe name to be crafted.
- **visit_binary_expression(self, node):** Evaluates a binary expression by processing its left and right operands and applying the specified operator.
- **visit_literal(self, node):** Returns the literal value (number or string) represented by the node.
- **visit_identifier(self, node):** Retrieves the value of an identifier from the global environment. If the identifier is not defined, an exception is raised.

13.5 generic_visit

generic_visit(self, node): This method is called for any node that does not have a dedicated visitor method. It raises an exception indicating that the node type is unsupported, ensuring that every part of the AST is properly handled.

Together, these components allow the interpreter to traverse the AST, evaluate expressions, update the environment, and ultimately execute the program written in our Minecraft programming language.

14 INTERPRETATION PROCESS

This code block integrates the entire compilation pipeline for our Minecraft programming language. It takes the source code as input and processes it through lexical, syntactic, and semantic analysis, finally executing the resulting Abstract Syntax Tree (AST) via the interpreter. The function `run_interpretation_process` orchestrates the following steps:

- **Lexical Analysis:** The source code is passed to the `Lexer`, which tokenizes the input based on predefined token patterns.
- **Syntactic Analysis:** The list of tokens is then processed by the `Parser` to construct the AST. A success message is printed upon completion.
- **Semantic Analysis:** The AST is traversed by the `SemanticAnalyzer` to ensure that the code adheres to the language's semantic rules. A success message is printed upon completion.
- **Interpretation:** Finally, the `Interpreter` executes the AST. If the AST contains a recipe (under the key `"recipe"`), it returns that part; otherwise, it returns the entire AST.

```
from interpreter.lexical_analyzer.lexer import Lexer
from interpreter.syntax_analyzer.parser import Parser
from interpreter.semantic_analyzer.semantic_analyzer import SemanticAnalyzer
from interpreter.evaluator.interpreter import Interpreter

def run_interpretation_process(code):
    # Lexical Analysis
    lexer = Lexer(code)
    tokens = lexer.tokenize()

    # Syntactic Analysis
    parser = Parser(tokens)
    abstract_syntax_tree = parser.parse()
    print("Syntactic analysis completed successfully.")

    # Semantic Analysis
    semantic_analyzer = SemanticAnalyzer()
    semantic_analyzer.analyze(abstract_syntax_tree)
    print("Semantic analysis completed successfully.")

    # Interpretation/Evaluation
    interpreter = Interpreter()
    interpreter.run(abstract_syntax_tree)

    % If the AST is a dictionary containing a recipe, return that part.
    if isinstance(abstract_syntax_tree, dict) and "recipe" in abstract_syntax_tree:
        return abstract_syntax_tree["recipe"]
    else:
        return abstract_syntax_tree

if __name__ == "__main__":
    code = """
        func calculate_area(length, width) {
            result = length * width;
            if (result > 100) {
                log("Large area calculated");
            } else {
                log("Small area calculated");
            }
        }

        recipe bread {
            input: [ (0,0) 1 wheat, (0,1) 1 wheat, (0,2) 1 wheat ];
            output: bread;
            tool_required: crafting_table;
            quantity: 1;
        }

        func greet_user(name) {
            log("Hello, " + name);
        }

```

```
        func countdown(number) {
            while (number > 0) {
                log("Countdown: " + number);
                number = number - 1;
            }
            log("Countdown finished");
        }

        func repeat_task(times) {
            for (i = 1; i <= times; i = i + 1) {
                log("Task repetition #" + i);
            }
        }

        func test_semantic_error() {
            log("Testing semantic error: " + undefined_variable);
        }
    """
    ast = run_interpretation_process(code)
    print("AST returned:", ast)
```

14.1 Lexical Analysis

The source code is first processed by the `Lexer` which tokenizes the input using the predefined token patterns. This stage is essential for converting the raw code into a structured sequence of tokens.

14.2 Syntactic Analysis

The `Parser` then takes the tokens and builds an Abstract Syntax Tree (AST) based on the grammatical rules of the language. This phase ensures that the code structure is correct and well-formed.

14.3 Semantic Analysis

After the AST is built, the `SemanticAnalyzer` traverses it to verify that the code adheres to semantic rules (e.g., correct usage of variables, valid recipes, etc.). This step is crucial for catching logical errors that are not detectable by syntax alone.

14.4 Interpretation and Evaluation

Finally, the `Interpreter` executes the AST, performing the actual computation, updating the global environment, and processing commands (such as crafting a recipe). If the AST contains a recipe (identified by the key `"recipe"`), that part of the AST is returned; otherwise, the entire AST is returned.

Together, these phases transform the source code into executable behavior, ensuring that the code is correct at every stage of the compilation process.

Part VII
CONTROLLERS

The controllers serve as the intermediary layer between the user interface and the core logic of our Minecraft programming language interpreter. In this phase, the controllers coordinate the flow of data from the code editor through the entire compilation pipeline (lexical, syntactic, and semantic analysis) and finally to the evaluation stage.

The primary controller, the `InterpreterController`, is responsible for:

- Retrieving the source code from the code editor.
- Initiating the interpretation process by invoking the `run_interpretation_process` function, which tokenizes, parses, and semantically analyzes the source code.
- Handling any errors during the interpretation process by catching exceptions (lexical, syntax, and semantic errors) and displaying detailed, color-coded messages in the debug panel.
- Updating the crafting table with the interpreted recipe information, ensuring that the final output is visually reflected in the user interface.
- Emitting signals to notify other components when the interpretation process has successfully completed.

By integrating these functionalities, the controllers not only manage the user input and output but also ensure a smooth transition from source code to executable behavior. This organized flow is essential for providing a responsive and informative development environment where users can immediately see the results of their code and receive clear feedback on any errors encountered during compilation.

15 INTERPRETER CONTROLLER

This code block implements the interpreter controller, which coordinates the interpretation process of our Minecraft programming language. The controller retrieves the source code from the code editor, passes it through the full compilation pipeline (lexical, syntactic, and semantic analysis) via the `run_interpretation_process` function, and then updates the crafting table based on the resulting recipe Abstract Syntax Tree (AST). It also handles errors by capturing lexical, syntax, and semantic exceptions, and prints detailed messages to the debug panel using custom formatting.

```
import json
from PyQt5.QtCore import QObject, pyqtSignal
from PyQt5.QtGui import QColor, QFont
from interpreter.run_interpretation_process import run_interpretation_process
from interpreter.lexical_analyzer.lexical_error import LexicalError
from interpreter.syntax_analyzer.syntax_error import SyntaxError
from interpreter.semantic_analyzer.semantic_error import SemanticError

class InterpreterController(QObject):
    interpretationFinished = pyqtSignal(object)

    def __init__(self, code_editor, crafting_table, debug_panel, parent=None):
        super().__init__(parent)
        self.code_editor = code_editor
        self.crafting_table = crafting_table
        self.debug_panel = debug_panel

    def debug_append(self, msg, color="black", bold=False):
        current_font = self.debug_panel.font()
        new_font = QFont(current_font)
```

```
21          new_font.setBold(bold)
22          self.debug_panel.setCurrentFont(new_font)
23          self.debug_panel.setTextColor(QColor(color))
24          self.debug_panel.append(msg)
25
26      def interpret_code(self):
27          if hasattr(self.debug_panel, 'clear'):
28              self.debug_panel.clear()
29          else:
30              print("Debug panel does not support clear()")
31
32          self.debug_append("Starting interpretation process...", color="blue", bold=
                True)
33          code = self.code_editor.toPlainText()
34
35          try:
36              recipe_ast = run_interpretation_process(code)
37          except LexicalError as le:
38              self.debug_append("Lexical Error:\n" + str(le), color="red", bold=True)
39              return
40          except SyntaxError as se:
41              self.debug_append("Syntax Error:\n" + str(se), color="red", bold=True)
42              return
43          except SemanticError as sme:
44              self.debug_append("Semantic Error:\n" + str(sme), color="red", bold=True)
45              return
46          except Exception as e:
47              self.debug_append("Unknown Error:\n" + str(e), color="red", bold=True)
48              return
49
50          if recipe_ast is None:
51              self.debug_append("Interpretation failed due to errors.", color="red",
                    bold=True)
52              return
53
54          if isinstance(recipe_ast, list):
55              if len(recipe_ast) > 0:
56                  recipe_ast = recipe_ast[0]
57              else:
58                  self.debug_append("No recipe found in the code.", color="red", bold=
                        True)
59                  return
60
61          self.crafting_table.update_from_ast(recipe_ast)
62          self.debug_append("Interpretation completed successfully.", color="green",
                bold=True)
63
64          try:
65              ast_details = json.dumps(recipe_ast, indent=4)
66              self.debug_append("", color="black")
67              self.debug_append("AST Details:", color="black", bold=True)
68              self.debug_append(ast_details, color="darkblue")
69          except Exception as e:
70              self.debug_append("Error formatting AST details: " + str(e), color="red",
                    bold=True)
71
72          self.interpretationFinished.emit(recipe_ast)
```

15.1 Constructor

__init__(self, code_editor, crafting_table, debug_panel): Initializes the interpreter controller with references to the code editor, crafting table, and debug panel. It also sets up a signal `interpretationFinished` which is emitted once the interpretation process is completed.

15.2 Debug Append Method

debug_append(self, msg, color="black", bold=False): This helper method formats and appends messages to the debug panel. It sets the text color (using a `QColor`) and adjusts the font (setting it to bold if required) before appending the message. This method is used throughout the controller to display status messages, errors, and details in a visually distinct way.

15.3 Interpret Code Method

interpret_code(self): This is the core method of the controller that performs the following steps:

1. **Clear the Debug Panel:** The debug panel is cleared at the start of the process.
2. **Start Message:** A starting message is appended in blue and bold.
3. **Retrieve Source Code:** The source code is extracted from the code editor.
4. **Interpretation Process:** The source code is processed via the `run_interpretation_process` function. During this phase, if any lexical, syntactic, or semantic errors occur, the corresponding exception is caught and a detailed error message is displayed in red and bold.
5. **AST Processing:** If a recipe AST is returned and is a list, the first recipe is extracted. If no recipe is found, an error message is shown.
6. **Update Crafting Table:** The crafting table widget is updated with the recipe information extracted from the AST.
7. **Success Message:** A success message is appended in green and bold.
8. **Display AST Details:** The AST is formatted as JSON with indentation and displayed in the debug panel (with the title in bold and the details in dark blue).
9. **Signal Emission:** Finally, the `interpretationFinished` signal is emitted with the AST.

15.4 Explanation

- **Error Handling:** The controller catches `LexicalError`, `SyntaxError`, `SemanticError`, and any other exceptions, displaying detailed error messages in red and bold to facilitate debugging.
- **AST Processing:** If the returned AST is a list, the controller extracts the first element, assuming it contains the recipe.
- **User Feedback:** Messages are provided at each stage (start, error, success, and AST details) with different colors to clearly distinguish the status of the interpretation process.
- **Signal Emission:** The `interpretationFinished` signal is emitted at the end, allowing other components of the application to react to the successful interpretation of the source code.

Part VIII
GRAPHICAL USER INTERFACE (GUI)

The Graphical User Interface (GUI) of our Minecraft Crafting Interpreter is designed to provide an interactive and user-friendly environment for writing code, debugging, and visualizing crafting recipes. The GUI integrates several distinct components that work together to offer a comprehensive development experience:

- **Code Editor:** A dedicated area where users can write and modify source code. It features syntax highlighting and a background inspired by Minecraft's aesthetic, ensuring that code is both legible and visually appealing.
- **Debug Panel:** This panel displays real-time feedback and detailed error messages during the interpretation process. It is designed with clear formatting and color-coded messages to help users quickly identify and resolve issues.
- **Crafting Table Widget:** Simulating the familiar 3x3 Minecraft crafting grid, this widget dynamically updates to reflect the recipe specified in the code. It provides a visual representation of the crafting process, showing the placement of materials.
- **Template Panel:** A convenient interface for selecting pre-defined recipe templates. Users can easily insert these templates into the code editor, speeding up the development of new crafting recipes.

The GUI is structured into left and right panels with contrasting background colors, ensuring that the different functional areas (code editing, debugging, and crafting simulation) are clearly separated. This design not only enhances usability but also reinforces the theme of the Minecraft programming language. The seamless integration between the GUI and the underlying interpreter and controllers ensures that any changes in the source code are immediately reflected in the crafting table and debug output, providing an interactive and responsive user experience.

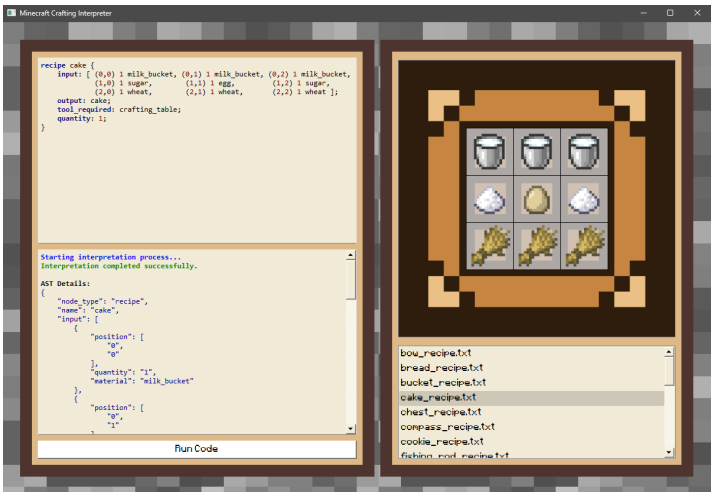

Figure 6: Example of the GUI to make a Cake. Source: Own elaboration

16 CODE EDITOR AND SYNTAX HIGHLIGHTER

This block of code implements a code editor with integrated syntax highlighting for our Minecraft programming language. The code editor is built upon `QPlainTextEdit`, providing a simple and efficient text editing interface. A `QSyntaxHighlighter` is attached to the document to dynamically apply syntax highlighting based on a set of predefined patterns.

```
from PyQt5.QtWidgets import QPlainTextEdit
from PyQt5.QtGui import QSyntaxHighlighter, QTextCharFormat, QFont
from PyQt5.QtCore import Qt, QRegExp

class CodeEditor(QPlainTextEdit):
    def __init__(self, parent=None):
        super(CodeEditor, self).__init__(parent)
        self.setFont(QFont("Consolas", 10))
        self.highlighter = SyntaxHighlighter(self.document())

class SyntaxHighlighter(QSyntaxHighlighter):
    def __init__(self, document):
        super(SyntaxHighlighter, self).__init__(document)
        self.highlightingRules = []

        % Keyword formatting: dark blue and bold
        keywordFormat = QTextCharFormat()
        keywordFormat.setForeground(Qt.darkBlue)
        keywordFormat.setFontWeight(QFont.Bold)
        keywords = [
            "func", "recipe", "input", "output", "tool_required", "quantity",
            "if", "else", "while", "for", "craft", "log", "return", "int", "float", "
                char"
        ]
        for keyword in keywords:
            pattern = QRegExp(r"\b" + keyword + r"\b")
            self.highlightingRules.append((pattern, keywordFormat))

        % Number formatting: dark red
        numberFormat = QTextCharFormat()
        numberFormat.setForeground(Qt.darkRed)
        pattern = QRegExp(r"\b\d+(\.\d+)?\b")
        self.highlightingRules.append((pattern, numberFormat))

        % String formatting: dark green
        stringFormat = QTextCharFormat()
        stringFormat.setForeground(Qt.darkGreen)
        pattern = QRegExp(r'"[^\"]*"')
        self.highlightingRules.append((pattern, stringFormat))

        % Comment formatting: gray
        commentFormat = QTextCharFormat()
        commentFormat.setForeground(Qt.gray)
        pattern = QRegExp(r"//[^\n]*")
        self.highlightingRules.append((pattern, commentFormat))

    def highlightBlock(self, text):
        for pattern, fmt in self.highlightingRules:
            expression = QRegExp(pattern)
            index = expression.indexIn(text)
            while index >= 0:
                length = expression.matchedLength()
                self.setFormat(index, length, fmt)
                index = expression.indexIn(text, index + length)
```

16.1 Constructor

__init__(self, parent=None): Initializes the code editor by setting the font to `Consolas` with size 10. It then instantiates a `SyntaxHighlighter` attached to the document of the code editor. This ensures that as the user types, the text is automatically formatted according to the defined syntax rules.

16.2 SyntaxHighlighter Constructor

__init__(self, document): The syntax highlighter is initialized with the document to be highlighted. It sets up a list of highlighting rules, where each rule is a pair of a regular expression (created with `QRegExp`) and a text format (defined with `QTextCharFormat`).

- **Keywords:** The regular expressions match whole words (using word boundaries), and the format is set to dark blue and bold.
- **Numbers:** A regular expression matches both integers and decimals, applying a dark red color.
- **Strings:** A pattern that matches sequences enclosed in double quotes, colored in dark green.
- **Comments:** A pattern that matches single-line comments (starting with `//`), displayed in gray.

16.3 highlightBlock

highlightBlock(self, text): This method is automatically invoked by `QSyntaxHighlighter` for each block of text in the document. It iterates through each highlighting rule and applies the corresponding text format to any substring in the block that matches the regular expression. This dynamic formatting improves the readability of the code by visually distinguishing different token types.

17 CRAFTING TABLE WIDGET

This block of code implements the `CraftingTableWidget`, a custom widget that simulates a 3x3 Minecraft crafting grid using the QGraphicsView framework. It displays a background image representing the crafting table and overlays a grid of cells where recipe items will be positioned. The widget also includes a method to update the grid based on a recipe AST (Abstract Syntax Tree).

```
import os
from PyQt5.QtWidgets import QGraphicsView, QGraphicsScene, QGraphicsRectItem,
    QGraphicsPixmapItem
from PyQt5.QtGui import QPen, QBrush, QColor, QPixmap
from PyQt5.QtCore import QRectF, Qt

class CraftingTableWidget(QGraphicsView):
    def __init__(self, rows=3, cols=3, cell_size=80, margin=20, parent=None):
        super().__init__(parent)
        self.rows = rows
        self.cols = cols
        self.cell_size = cell_size
        self.margin = margin
        self.grid_width = self.cols * self.cell_size
        self.grid_height = self.rows * self.cell_size
        self.bg_width = (self.grid_width + 4 * self.margin) + 100
        self.bg_height = (self.grid_height + 4 * self.margin) + 100

        self.scene = QGraphicsScene(self)
        self.setScene(self.scene)
        self.scene.setSceneRect(0, 0, self.bg_width, self.bg_height)
        self.setFixedSize(self.bg_width + 50, self.bg_height + 50)
        self._load_background()
        self._draw_grid()
```

```
24          self.items = {}
25
26      def _load_background(self):
27          base_dir = os.path.dirname(os.path.abspath(__file__))
28          bg_path = os.path.join(base_dir, "..", "resources", "images", "
                crafting_table_bg.png")
29          bg_path = bg_path.replace("\\", "/")
30          pixmap = QPixmap(bg_path)
31          if pixmap.isNull():
32              print(f"Error: Could not load background image from {bg_path}")
33              return
34          scaled_pixmap = pixmap.scaled(self.bg_width, self.bg_height, Qt.
                IgnoreAspectRatio, Qt.SmoothTransformation)
35          self.scene.setBackgroundBrush(QBrush(scaled_pixmap))
36
37      def _draw_grid(self):
38          pen = QPen(QColor("black"))
39          cell_color = QColor("lightgray")
40          cell_color.setAlpha(196)  % Setting transparency to 70%
41          brush = QBrush(cell_color)
42          offset_x = (self.bg_width - self.grid_width) / 2
43          offset_y = (self.bg_height - self.grid_height) / 2
44          for row in range(self.rows):
45              for col in range(self.cols):
46                  x = offset_x + col * self.cell_size
47                  y = offset_y + row * self.cell_size
48                  rect = QRectF(x, y, self.cell_size, self.cell_size)
49                  cell = QGraphicsRectItem(rect)
50                  cell.setPen(pen)
51                  cell.setBrush(brush)
52                  self.scene.addItem(cell)
53
54      def update_from_ast(self, recipe_ast):
55          % Remove any previous items from the scene.
56          for item in self.items.values():
57              self.scene.removeItem(item)
58          self.items.clear()
59
60          offset_x = (self.bg_width - self.grid_width) / 2
61          offset_y = (self.bg_height - self.grid_height) / 2
62
63          for item in recipe_ast.get("input", []):
64              position = item["position"]
65              quantity = item["quantity"]
66              material = item["material"]
67              try:
68                  row, col = int(position[0]), int(position[1])
69              except Exception as e:
70                  print(f"Invalid position format for item: {item}")
71                  continue
72
73              base_dir = os.path.dirname(os.path.abspath(__file__))
74              image_path = os.path.join(base_dir, "..", "resources", "images", f"{
                    material}.png")
75              image_path = image_path.replace("\\", "/")
76              pixmap = QPixmap(image_path)
77              if pixmap.isNull():
78                  print(f"Image not found for material: {material}")
79                  continue
80
81              pixmap = pixmap.scaled(self.cell_size - 10, self.cell_size - 10, Qt.
                    KeepAspectRatio, Qt.SmoothTransformation)
82              pixmap_item = QGraphicsPixmapItem(pixmap)
83              x = offset_x + col * self.cell_size + (self.cell_size - pixmap.width()) /
                    2
```

```
84              y = offset_y + row * self.cell_size + (self.cell_size - pixmap.height()) /
                    2
85              pixmap_item.setPos(x, y)
86              self.scene.addItem(pixmap_item)
87              self.items[(row, col)] = pixmap_item
```

17.1 Constructor

__init__(self, rows, cols, cell_size, margin, parent): Initializes the crafting table widget with a default grid of 3 rows by 3 columns. It sets up parameters such as cell size and margin, calculates the overall dimensions of the grid and the background, creates a QGraphicsScene, and calls helper methods to load the background and draw the grid. It also initializes an empty dictionary to store the item images.

17.2 _load_background

_load_background(self): Loads the background image for the crafting table from the specified resources path, scales it to the dimensions of the background, and sets it as the background brush of the scene. If the image cannot be loaded, an error message is printed.

17.3 _draw_grid

_draw_grid(self): Draws a 3x3 grid on the crafting table. It calculates the horizontal and vertical offsets to center the grid within the background. Each cell is drawn as a rectangle with a black border and a light gray fill with 70% opacity.

17.4 update_from_ast

update_from_ast(self, recipe_ast): Updates the crafting table based on the recipe AST. It first clears any existing items, then iterates over the list of input items in the AST. For each item, it calculates the centered position within the corresponding cell and loads the image of the material from the resources folder. The item image is then scaled appropriately and added to the scene, with its position stored in a dictionary for future reference.

18 DEBUG PANEL

This code block implements the DebugPanel, a specialized widget designed for displaying debugging messages. It extends the `QTextEdit` class, ensuring that the panel is read-only and uses a fixed-width font for clarity. The DebugPanel is used by the interpreter controller to output status updates, error messages, and other debugging information in a user-friendly manner.

```
1  from PyQt5.QtWidgets import QTextEdit
2  from PyQt5.QtGui import QFont
3
4  class DebugPanel(QTextEdit):
5      def __init__(self, parent=None):
6          super(DebugPanel, self).__init__(parent)
7          self.setReadOnly(True)
8          self.setFont(QFont("Consolas", 10))
9          self.setStyleSheet("background-color: #f0f0f0; color: black;")
10
11     def append_message(self, message):
12         self.append(message)
```

18.1 Constructor

__init__(self, parent=None): Initializes the DebugPanel by invoking the constructor of the `QTextEdit` class. It sets the widget to read-only mode, ensuring that users cannot modify the debugging output. The font is set to `Consolas` with a size of 10, which is ideal for displaying code and logs, and a style sheet is applied to give the panel a light gray background with black text.

18.2 append_message

append_message(self, message): This method appends a new debugging message to the panel by calling the `append()` method inherited from `QTextEdit`. It provides a simple interface to display logs, errors, and status messages, which helps in tracking the program's execution and debugging.

19 TEMPLATE PANEL

This code block implements the `TemplatePanel`, a widget that displays a list of pre-defined recipe templates. The panel extends `QListWidget` and loads template files from a specified directory. When a template is selected (clicked), its content is loaded into the code editor, allowing the user to quickly insert or modify existing recipes.

```
import os
from PyQt5.QtWidgets import QListWidget

class TemplatePanel(QListWidget):
    def __init__(self, templates_dir, code_editor, parent=None):
        super(TemplatePanel, self).__init__(parent)
        base_dir = os.path.dirname(os.path.abspath(__file__))
        self.templates_dir = os.path.join(base_dir, "..", templates_dir)
        self.code_editor = code_editor
        self.load_templates()
        self.itemClicked.connect(self.load_template)

    def load_templates(self):
        self.clear()
        try:
            for filename in os.listdir(self.templates_dir):
                if filename.endswith('.txt'):
                    self.addItem(filename)
        except Exception as e:
            print("Error loading templates:", e)

    def load_template(self, item):
        filepath = os.path.join(self.templates_dir, item.text())
        try:
            with open(filepath, "r", encoding="utf-8") as f:
                code = f.read()
            self.code_editor.setPlainText(code)
        except Exception as e:
            print("Error loading template:", e)
```

19.1 Constructor

__init__(self, templates_dir, code_editor, parent): Initializes the TemplatePanel by:

- Determining the absolute path of the templates directory by joining the base directory with the provided `templates_dir` path.
- Storing a reference to the code editor so that selected templates can be loaded into it.

- Loading the available templates from the directory by calling `load_templates()`.
- Connecting the `itemClicked` signal to the `load_template` method to enable dynamic loading of a template when a user clicks on an item in the list.

19.2 load_templates

load_templates(self): This method clears the current list of templates and then iterates through the files in the templates directory. It adds to the list those files that end with the `.txt` extension. If an error occurs during this process, it prints an error message.

19.3 load_template

load_template(self, item): When a user clicks on a template in the list, this method is called. It constructs the full file path from the template name, reads the file content (assuming UTF-8 encoding), and loads the content into the code editor by setting its plain text. If any error occurs during file reading, an error message is printed.

Part IX
MAIN WINDOW

This block of code implements the main graphical user interface (GUI) of our Minecraft Crafting Interpreter. The MainWindow class, which inherits from `QMainWindow`, sets up the primary window of the application. It arranges the major components of the interface, including the code editor, the debug panel, the crafting table, and the template panel, into left and right panels. The GUI is styled with colors that reflect a Minecraft-inspired aesthetic and utilizes a custom font if available.

```
import sys
import os
from PyQt5.QtWidgets import (
    QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout, QPushButton
)
from PyQt5.QtGui import QPixmap, QPalette, QBrush, QFontDatabase, QFont
from PyQt5.QtCore import Qt
from gui.code_editor import CodeEditor
from gui.debug_panel import DebugPanel
from gui.crafting_table import CraftingTableWidget
from gui.template_panel import TemplatePanel
from controller.interpreter_controller import InterpreterController

class MainWindow(QMainWindow):
    def __init__(self):
        super(MainWindow, self).__init__()
        self.setWindowTitle("Minecraft Crafting Interpreter")
        self.setGeometry(100, 100, 1200, 800)
        self.init_ui()

    def init_ui(self):
        base_dir = os.path.dirname(os.path.abspath(__file__))

        % Set the background image for the main window.
        cobblestone_path = os.path.join(base_dir, "resources/images/
            cobblestone_background.png").replace("\\", "/")
        cobblestone_pixmap = QPixmap(cobblestone_path)
        if not cobblestone_pixmap.isNull():
            palette = QPalette()
            palette.setBrush(QPalette.Window, QBrush(cobblestone_pixmap))
            self.setPalette(palette)
            self.setAutoFillBackground(True)
        else:
            print(f"Background image not found: {cobblestone_path}")

        % Define color schemes for different panels.
        dark_brown = "#4E342E"
        light_brown = "#DEB887"
        birch_color = "#F0EAD6"

        % Create the central widget with a transparent background.
        central_widget = QWidget()
        central_widget.setStyleSheet("background: transparent;")
        self.setCentralWidget(central_widget)

        main_layout = QHBoxLayout(central_widget)
        main_layout.setContentsMargins(30, 30, 30, 30)
        main_layout.setSpacing(10)

        % Left Panel: Contains the Code Editor, Debug Panel, and Run Button.
        left_panel = QWidget()
        left_panel.setStyleSheet(f"background-color: {dark_brown};")
        left_panel.setMaximumWidth(800)
```

```
53          left_layout = QVBoxLayout(left_panel)
54          left_layout.setContentsMargins(20, 20, 20, 20)
55          left_layout.setSpacing(10)
56
57          % Left Content: Has a light brown background to contrast with the outer panel.
58          left_content = QWidget()
59          left_content.setStyleSheet(f"background-color: {light_brown};")
60          left_content_layout = QVBoxLayout(left_content)
61          left_content_layout.setContentsMargins(10, 10, 10, 10)
62          left_content_layout.setSpacing(10)
63
64          self.code_editor = CodeEditor()
65          self.code_editor.setStyleSheet(f"background-color: {birch_color};")
66          self.debug_panel = DebugPanel()
67          self.debug_panel.setStyleSheet(f"background-color: {birch_color};")
68          self.run_button = QPushButton("Run Code")
69          self.run_button.setStyleSheet("background-color: white;")
70          self.run_button.clicked.connect(self.run_code)
71
72          left_content_layout.addWidget(self.code_editor)
73          left_content_layout.addWidget(self.debug_panel)
74          left_content_layout.addWidget(self.run_button)
75
76          left_layout.addWidget(left_content)
77
78          % Right Panel: Contains the Crafting Table and Template Panel.
79          right_panel = QWidget()
80          right_panel.setStyleSheet(f"background-color: {dark_brown};")
81          right_panel.setMaximumWidth(600)
82          right_layout = QVBoxLayout(right_panel)
83          right_layout.setContentsMargins(20, 20, 20, 20)
84          right_layout.setSpacing(10)
85
86          right_content = QWidget()
87          right_content.setStyleSheet(f"background-color: {light_brown};")
88          right_content_layout = QVBoxLayout(right_content)
89          right_content_layout.setContentsMargins(10, 10, 10, 10)
90          right_content_layout.setSpacing(10)
91          right_content_layout.addStretch()
92          self.crafting_table = CraftingTableWidget()
93          right_content_layout.addWidget(self.crafting_table, alignment=Qt.AlignCenter)
94          right_content_layout.addStretch()
95          self.template_panel = TemplatePanel("templates", self.code_editor)
96          self.template_panel.setStyleSheet(f"background-color: {birch_color};")
97          right_content_layout.addWidget(self.template_panel, 0)
98          right_layout.addWidget(right_content)
99
100         main_layout.addWidget(left_panel, 2)
101         main_layout.addWidget(right_panel, 1)
102
103         self.interpreter_controller = InterpreterController(
104             self.code_editor, self.crafting_table, self.debug_panel
105         )
106
107     def run_code(self):
108         self.interpreter_controller.interpret_code()
109
110 if __name__ == "__main__":
111     app = QApplication(sys.argv)
112
113     base_dir = os.path.dirname(os.path.abspath(__file__))
114     font_path = os.path.join(base_dir, "resources/fonts/Minecraftia.ttf").replace("\\"
        , "/")
115     if os.path.exists(font_path):
116         font_id = QFontDatabase.addApplicationFont(font_path)
```

```
117         if font_id != -1:
118             families = QFontDatabase.applicationFontFamilies(font_id)
119             if families:
120                 minecraft_font = QFont(families[0], 10)
121                 app.setFont(minecraft_font)
122             else:
123                 print("No font families found after loading Minecraftia.ttf")
124         else:
125             print("Failed to load Minecraftia.ttf")
126     else:
127         print(f"Font file not found: {font_path}")
128 
129     window = MainWindow()
130     window.show()
131     sys.exit(app.exec_())
```

19.4 Constructor

__init__(self): Initializes the MainWindow by setting its title, geometry, and then calling the `init_ui()` method. This method is responsible for building the entire GUI by creating the central widget and arranging the left and right panels.

19.5 init_ui

init_ui(self): Sets up the GUI components:

- Loads the cobblestone background image and sets it as the window background.
- Defines color schemes for the panels: dark brown for outer panels, light brown for content containers, and a birch-like color for specific widget backgrounds.
- Creates a central widget with a transparent background.
- Constructs the left panel, which contains the code editor, debug panel, and a "Run Code" button, all arranged vertically.
- Constructs the right panel, which contains the crafting table (centered) and the template panel for recipe templates.
- Initializes the `InterpreterController` with references to the code editor, crafting table, and debug panel.

19.6 run_code

run_code(self): This method is connected to the "Run Code" button. When triggered, it calls the `interpret_code()` method of the `InterpreterController`, initiating the interpretation process of the source code in the code editor.

This comprehensive setup not only organizes the user interface into functional areas but also integrates the interpreter pipeline seamlessly with the GUI components, allowing for real-time code execution and visual feedback in our Minecraft Crafting Interpreter.

Part X
CONCLUSIONS

The development of our Minecraft programming language has been an enlightening journey through the fundamental stages of compiler design. In this project, we have implemented a complete pipeline that transforms source code into executable behavior, with each phase playing a critical role in ensuring the robustness and correctness of the language.

Lexical Analysis

Lexical analysis serves as the first barrier against malformed code. By using regular expressions and well-defined token patterns, our lexer efficiently breaks down the raw source code into meaningful tokens. The use of production rules and regex patterns not only ensures that each element of the language is recognized correctly, but also paves the way for precise error detection at the very outset. This early filtering of invalid characters and symbols is essential, as it establishes the foundation upon which the remaining phases build. Over the decades, the refinement of lexical analyzers has contributed significantly to the reliability of programming languages, and our approach reflects this evolution by strictly adhering to the foundational rules.

Syntactic Analysis

Syntactic analysis, or parsing, is the stage where the tokenized input is organized into a hierarchical structure, typically an Abstract Syntax Tree (AST). Our parser verifies that constructs such as function definitions, recipes, assignments, and control structures conform to the grammatical rules of our language. The use of generative grammars in Extended Backus-Naur Form (EBNF) provides a clear and precise definition of the language syntax, ensuring that every valid program is correctly represented in the AST. This phase not only validates the structural integrity of the source code but also creates a well-organized representation that is essential for subsequent semantic analysis. Historically, the evolution of parsing techniques has been central to the development of reliable compilers, and our implementation stands as a modern example of these time-tested methods.

Semantic Analysis

Once the AST is constructed, semantic analysis ensures that the program makes logical sense. This stage verifies that variables are declared before use, that operations are performed on compatible types, and that specific rules—such as valid index ranges in recipes—are upheld. Our semantic analyzer traverses the AST, consulting a symbol table to confirm the proper use of identifiers and to enforce semantic constraints. The incorporation of custom semantic error handling further enhances the robustness of our language by providing clear, actionable feedback when semantic rules are violated. This rigorous approach to semantic checking has been a cornerstone in the evolution of programming languages, ensuring that even syntactically correct code does not produce unintended or harmful behavior.

Interpretation and Compilation

The final phase of our project is the interpretation and evaluation of the AST. The interpreter traverses the AST, executing functions, evaluating expressions, and updating the global environment. This phase bridges the gap between static code and dynamic execution. By implementing a modular visitor

pattern, our interpreter is capable of handling diverse constructs such as conditionals, loops, and crafting commands. This dynamic execution model is a testament to the power of modern language design, which builds upon decades of research in both interpretation and compilation techniques.

Impact of Production Rules, Regular Expressions, and Generative Grammars

At the heart of our language are the production rules defined in our generative grammar and the regular expressions used in lexical analysis. These formal specifications ensure that every element of the language is unambiguously defined and that any valid source code can be systematically analyzed and transformed. The rigorous application of these concepts has not only made our language robust but also highlights the historical significance of formal language theory in the evolution of programming languages. From the early days of compiler design to modern interpreters and just-in-time compilers, these principles have consistently enabled developers to build powerful, reliable, and maintainable systems.

Historical and Practical Significance

The techniques employed in our project are the result of decades of research and practical experience in compiler design. Lexical analysis, parsing, and semantic analysis have evolved from simple ad-hoc methods into sophisticated processes that underpin modern programming languages. Our Minecraft programming language serves as both an educational tool and a practical application, demonstrating how these fundamental principles can be applied to create a language that is both expressive and robust. The integration of these stages—each with its own well-defined responsibilities—ensures that our language not only catches errors early in the compilation process but also provides a clear and structured path from source code to execution.

In summary, the combination of well-designed lexical analysis, syntactic structure, semantic validation, and dynamic interpretation forms the backbone of our programming language. This multi-layered approach not only guarantees that the source code is transformed accurately into an executable form but also reflects the historical progression of compiler design. As we continue to develop and refine our language, these foundational concepts remain at the core, illustrating how advanced programming paradigms have been built upon rigorous theoretical principles.