Grammar Rules and Syntactic Structure of KIK Language

KIK is a procedural programming language , with syntax for control structures (using colons after conditions), data types, constants, and input/output operations. Programs begin with a main function int kik() { ... }, supporting declarations, statements, and additional functions. Imports (e.g., import "io.kik";) enable custom I/O functionality. The grammar is defined in Extended Backus-Naur Form (EBNF), where terminals are quoted (e.g., "int"), non-terminals are in angle brackets (e.g., <identifier>), { ... }* denotes zero or more repetitions, { ... }+ denotes one or more, | separates alternatives, and [...] indicates optional elements.

Here , we go with all grammars for **KIK** language . . . .

1. Program Structure

A KIK program consists of optional imports, a mandatory main function, and optional additional functions or declarations.

```
<program> ::= { <import> }* <main-function> { <function> | <declaration> }*
<import> ::= "import" <string-literal> ";"
<main-function> ::= "int" "kik" "(" ")" "{" <statement>* "return" <expression> ";" "}"
<function> ::= <type> <identifier> "(" [ <parameter-list> ] ")" "{" <statement>* "}"
<parameter-list> ::= <parameter> { "," <parameter> }*
<parameter> ::= <type> <identifier>
```

- **Notes**: The main function returns an integer (e.g., return 0;). Additional functions follow similar syntax.

2. Data Types

KIK supports primitive types (int, float, char, bool) and strings (via str or char arrays). Booleans use true or false.

```
<type> ::= "int" | "float" | "char" | "bool" | "str" | "void"
<array-type> ::= "char" <identifier> "[" <integer-literal> "]"
<boolean-literal> ::= "true" | "false"
```

- **Examples**:
    - int age = 25;
    - float height = 5.9;
    - char grade = 'A';
    - bool isStudent = true;
    - str firstName;
    - char name[100];

3. Declarations

Variables can be declared with or without initialization. Multiple variables in a single declaration initialize only the last variable unless explicitly assigned. Constants use the constant keyword.

```
<declaration> ::= [ "constant" ] <type> <var-list> ";"
<var-list> ::= <var-init> { "," <var-init> }*
<var-init> ::= <identifier> [ "=" <expression> ]
```

- **Multiple Variables**:
    - Syntax: <type> var1, var2, var3 = value; (only var3 is initialized; var1, var2 are uninitialized).
    - Explicit initialization: a = b = c = 10;.
- **Constants**:
    - Syntax: constant [type] constantName = value;
    - Examples: constant price_apple = 1.50; (type inferred), constant float price_banana = 0.75;.

4. Expressions

Expressions include literals, variables, operators, function calls, string concatenation, and array access. Operator precedence mirrors major programming language (e.g., *, / before +, -).

```
<expression> ::= <assignment-expr> | <logical-expr> | <comparison-expr
> | <arithmetic-expr> | <unary-expr> | <primary-expr>
<primary-expr> ::= <identifier> | <literal> | "(" <expression> ")" | <function
-call> | <string-concat> | <array-access>
<literal> ::= <integer-literal> | <float-literal> | <char-literal> | <string-literal
> | <boolean-literal>
<integer-literal> ::= [ "-" ] <digit>+
<float-literal> ::= [ "-" ] <digit>* "." <digit>+
<char-literal> ::= "'" <character> "'"
<string-literal> ::= '"' { <character> }* '"'

<arithmetic-expr> ::= <expression> ("+" | "-" | "*" | "/" | "%") <expression
>
<comparison-expr> ::= <expression> ("==" | "!=" | ">" | "<" | ">=" | "<=")
<expression>
<logical-expr> ::= <expression> ("&&" | "||") <expression> | "!" <expressio
n>
<assignment-expr> ::= <identifier> ("=" | "+=" | "-=" | "*=" | "/=" | "%=") <
expression>

<function-call> ::= <identifier> "(" [ <arg-list> ] ")"
<arg-list> ::= <expression> { "," <expression> }*

<string-concat> ::= <expression> "+" <expression>
<array-access> ::= <identifier> "[" <expression> "]"
<array-assign> ::= <identifier> "[" <expression> "]" "=" <expression>
```

- **Operators**:
    - Arithmetic: +, -, *, /, %.
    - Assignment: =, +=, -=, *=, /=, %=.
    - Comparison: ==, !=, >, <, >=, <= (return bool).

    - Logical: && (AND), || (OR), ! (NOT).
- **String Operations**:
    - Declaration: str name;
    - Concatenation: str result = string1 + string2;
    - Access/Modify: char c = str[0];, str[0] = 'X';.

5. Statements

Statements include declarations, assignments, I/O operations, control structures, and control flow keywords.

```
<statement> ::= <declaration> | <expression> ";" | <io-statement> | <if-statement> | <switch-statement> | <loop-statement> | "return" <expression> ";" | "break;" | "continue;"
```

- **Input/Output**:
    - Uses major programming language -style cout << expr << endl; and cin >> var;, or custom io.kik functions: input() (returns string), output(message).

```
<io-statement> ::= "cout" "<<" <expression> { "<<" <expression> }* [ "<<" "endl" ] ";"
             | "cin" ">>" <identifier> { ">>" <identifier> }* ";"
             | "output" "(" <expression> ")" ";"
             | <identifier> "=" "input" "(" ")" ";"
```

6. Control Structures

Control structures use a colon (:) after conditions or expressions, distinguishing KIK from major programming language.

- **If-Else**:

```
<if-statement> ::= "if" <expression> ":" "{" <statement>* "}"
             { "else if" <expression> ":" "{" <statement>* "}" }*
```

```
            [ "else" ":" "{" <statement>* "}" ]
```

- Example: if score >= 90: { cout << "Grade: A"; } else if score >= 80: { ... } else: { ... }

- **Switch-Case**:

```
<switch-statement> ::= "switch" <expression> ":" "{"
             { "case" <literal> ":" <statement>* "break;" }*
             [ "default" ":" <statement>* ] "}"
```

  - Example: switch day: { case 1: dayName = "Monday"; break; ... default: ... }

- **Loops**:
  - **While**:

```
<while-loop> ::= "while" <expression> ":" "{" <statement>* "}"
```

    - Example: while count <= 5: { cout << count; count++; }
  - **Do-While**:

```
<do-while-loop> ::= "do" "{" <statement>* "}" "while" <expression
> ":" "{" "}"
```

    - Example: do { cout << count; count++; } while count <= 5: { } (empty block after colon).
  - **For**:

```
<for-loop> ::= "for" <initialization> ";" <expression> ";" ":" <increm
ent> ":" "{" <statement>* "}"
<initialization> ::= [ <type> ] <assignment-expr>
<increment> ::= <expression>
```

    - Example: for int count = 1; count <= 5;: count++ : { cout << count; }
  - **Nested Loops**: For loops can be nested, e.g., for int i = 1; i <= 5; i++ : { for int j = 1; j <= 5; j++ : { ... } }

- **Break and Continue**:
    - break; (exits loop or switch).
    - continue; (skips to next iteration).

7. Identifiers and Literals

```
<identifier> ::= <letter> { <letter> | <digit> | "_" }*
<digit> ::= "0".."9"
<letter> ::= "a".."z" | "A".."Z"
<character> ::= any printable character except "'" or '"'
```

- **Semantics**:
    - KIK is statically typed. Uninitialized variables have undefined values.
    - Division by zero or invalid operations cause runtime errors.
    - Strings are mutable via index access.

8. Plan for Editor Development

To develop a KIK code editor (e.g., as a VS Code extension or standalone Electron app), we are working on it .

We have reviewed all requirements for the editor and are currently in the planning phase. However, full-fledged development work has yet to begin.

factorial.kik - KIK Program to Calculate Factorial of a Number

```


import "io.kik";

```

```
int kik() {

int factorial = 1;

output("Enter a positive integer: ");
number = input();

## Convert string input to integer (assuming input is valid)
int num = 0;
for int i = 0; i < number.length(); i++: {
    num = num * 10 + (number[i] - '0');
}



if num < 0: {
    cout << "Error: Please enter a non-negative number!" << endl;
    return 1;
}

## Calculate factorial using a loop
for int i = 1; i <= num; i++: {
    factorial *= i;
}



cout << "Factorial of " << num << " is " << factorial << endl;

return 0;
}
```

9. Contributions

All of above done by Aman Anand (CS22B054)