Documentation: BNF Grammar, Syntax, and Features

---

1. Introduction

This document provides a comprehensive overview of the BNF (Backus-Naur Form) grammar used in the interpreter described in the provided code. The document outlines the language's syntax, features, and key trade-offs, as well as any limitations that may exist within the language.

2. BNF Grammar

The BNF grammar defines the structure of the language, specifying how different constructs can be combined to form valid expressions and statements. Below is the BNF grammar used in the interpreter:

```
expr        : KEYWORD:VAR IDENTIFIER EQ expr
            : comp-expr ((AND|OR) comp-expr)*

comp-expr   : arith-expr ((EE|LT|GT|LTE|GTE) arith-expr)*
            : NOT? comp-expr

arith-expr  : term ((PLUS|MINUS) term)*

term        : factor ((MUL|DIV|MOD) factor)*

call        : atom (LPAREN (expr (COMMA expr)*)? RPAREN)?

factor      : (PLUS|MINUS) factor
            : power

power       : call (POW factor)*

atom        : INT | IDENTIFIER | bool
            : (LPAREN expr RPAREN)
            : if-expr | func-def
            : NOT expr

if-expr     : KEYWORD:IF expr KEYWORD:THEN expr
            (KEYWORD:ELIF expr KEYWORD:THEN expr)*
            (KEYWORD:ELSE expr)?

func-def    : KEYWORD:FUN IDENTIFIER?
            LPAREN (IDENTIFIER (COMMA IDENTIFIER)*)? RPAREN
            ARROW expr
```

3. Detailed Syntax Description

3.1 Expressions (expr)

- **Variable Assignment:**

expr : KEYWORD:VAR IDENTIFIER EQ expr

  - **Description:** This rule defines how a variable is declared and assigned a value. The keyword VAR is followed by an identifier (the variable name), the assignment operator (=), and an expression.
  - **Example:** VAR a = 5

- **Boolean Logic:**

expr : comp-expr ((AND|OR) comp-expr)*

  - **Description:** This rule defines logical expressions using AND and OR. Multiple comparison expressions (comp-expr) can be combined using these logical operators.
  - **Example:** a && b

3.2 Comparison Expressions (comp-expr)

- **Comparison Operations:**

comp-expr : arith-expr ((EE|LT|GT|LTE|GTE) arith-expr)*

  - **Description:** This rule defines comparison operations between arithmetic expressions. Supported comparison operators include == (EE), < (LT), > (GT), <= (LTE), and >= (GTE).
  - **Example:** a == b

- **Negation:**

comp-expr : NOT? comp-expr

  - **Description:** This rule allows optional negation (NOT) of a comparison expression.
  - **Example:** !a

3.3 Arithmetic Expressions (arith-expr)

- **Addition and Subtraction:**

arith-expr : term ((PLUS|MINUS) term)*

  - **Description:** This rule defines how terms are added or subtracted. Multiple terms can be combined using the + (PLUS) and - (MINUS) operators.
  - **Example:** a + b - c

3.4 Terms (term)

- **Multiplication, Division, and Modulus:**

term : factor ((MUL|DIV|MOD) factor)*

  - **Description:** This rule defines how factors are multiplied, divided, or moduloed. The operators supported are * (MUL), / (DIV), and % (MOD).
  - **Example:** a * b / c % d

3.5 Factors (factor)

- **Unary Operations:**

factor : (PLUS|MINUS) factor

  - **Description:** This rule allows for unary operations (positive or negative signs) to be applied to factors.
  - **Example:** -5

3.6 Power (power)

- **Exponentiation:**

power : call (POW factor)*

  - **Description:** This rule allows for exponentiation using the ^ (POW) operator. Multiple exponentiations can be chained.
  - **Example:** 3 ^ 2

3.7 Function Calls (call)

- **Function Invocation:**

call : atom (LPAREN (expr (COMMA expr)*)? RPAREN)?

  - **Description:** This rule defines how functions are called with arguments. The function name (atom) is followed by parentheses enclosing the arguments.
  - **Example:** func(a, b)

3.8 Atoms (atom)

- **Basic Units:**

atom : INT | IDENTIFIER | bool

| LPAREN expr RPAREN

| if-expr | func-def

| NOT expr

  - **Description:** This rule covers the basic units of the language, including integers, identifiers, booleans, expressions enclosed in parentheses, if expressions, function definitions, and negated expressions.
  - **Example:** 5, a, (a + b)

3.9 If Expressions (if-expr)

- **Conditional Statements:**

if-expr : KEYWORD:IF expr KEYWORD:THEN expr

(KEYWORD:ELIF expr KEYWORD:THEN expr)*

(KEYWORD:ELSE expr)?

  - **Description:** This rule defines conditional statements with IF, ELIF, and ELSE branches. Each condition is followed by a THEN clause indicating the expression to execute if the condition is true.
  - **Example:** IF a > b THEN a ELSE b

3.10 Function Definitions (func-def)

- **Defining Functions:**

func-def : KEYWORD:FUN IDENTIFIER?

LPAREN (IDENTIFIER (COMMA IDENTIFIER)*)? RPAREN

ARROW expr

  - **Description:** This rule defines functions using the FUN keyword, optionally followed by a function name and arguments. The function body follows the -> (ARROW) operator.
  - **Example:** FUN add(a, b) -> a + b

3.11 Booleans (bool)

- **Boolean Values:**

bool : TRUE | FALSE

  - **Description:** This rule defines the boolean values TRUE and FALSE.
  - **Example:** True

---

4. Trade-offs and Limitations

1. **Trade-offs:**
   - **Simplicity vs. Flexibility:** The language syntax is designed to be simple and easy to understand, which limits the flexibility and complexity of operations. For example, the language does not support more complex data types like arrays or objects.
   - **Operator Precedence:** Operator precedence is manually controlled within the grammar. This simplicity ensures that users have a straightforward mental model, but it might lead to some counterintuitive results if users expect precedence rules similar to other programming languages.
   - **Implicit Type Handling:** The language has basic type handling, where booleans are treated as integers (e.g., True as 1 and False as 0). While this simplifies implementation, it can lead to subtle bugs or confusion, especially for beginners.
2. **Limitations:**
   - **Limited Data Types:** The language supports only integers and booleans. There is no support for strings, floats, or more complex data structures.
   - **No Error Recovery:** If an error occurs during parsing or execution, the interpreter halts and reports the error. There is no mechanism for error recovery or handling within the language.
   - **No Scoping or Blocks:** The language does not support scoping beyond the global scope, which can lead to issues with variable shadowing or unintended side effects.
   - **Limited Control Flow:** Apart from if statements, the language lacks advanced control flow structures like loops, switches, or pattern matching.

---

5. Conclusion

The language defined by this BNF grammar provides a simple and straightforward syntax suitable for basic arithmetic and boolean operations, function definitions, and conditional statements. While it offers ease of use and clarity, it has certain trade-offs and limitations in terms of flexibility, data types, and control flow structures. Despite these limitations, the language is a useful tool for educational purposes or for implementing simple scripting tasks.