UNIVERSIDADE VILA VELHA

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Diogo Francis Belshoff

RELATÓRIO TÉCNICO

Linguagem de programação Kotlin com paradigma de programação Procedural

VILA VELHA

2024

Diogo Francis Belshoff

RELATÓRIO TÉCNICO

Linguagem de programação Kotlin com paradigma de programação Procedural

Trabalho apresentado no curso de Graduação em Ciência da Computação da Universidade Vila Velha.

Orientador: Wagner de Andrade Perin

VILA VELHA

2024

Sumário

1. INTRODUÇÃO

Este relatório tem como objetivo explorar o uso do paradigma procedural na linguagem de programação Kotlin, que é amplamente conhecida por suportar múltiplos paradigmas de programação, como orientado a objetos, funcional e procedural. O paradigma procedural é focado em uma execução sequencial de instruções, utilizando funções e procedimentos para organizar e modularizar o código. Neste trabalho, exemplificaremos o uso deste paradigma em Kotlin por meio de diferentes exemplos de funções e discutiremos suas aplicações e características.

2. O PARADIGMA PROCEDURAL

O paradigma procedural ou programação procedural é uma abordagem que organiza o código em procedimentos ou funções, executando instruções de forma sequencial. A seguir, destacamos as principais características deste paradigma:

- **Sequência de Instruções**: O programa é uma sequência de operações executadas em uma ordem específica.
- **Funções e Procedimentos**: O código é estruturado em blocos reutilizáveis, promovendo a modularidade.
- **Variáveis Globais e Locais**: Os dados são organizados em variáveis, que podem ser globais ou locais às funções.
- **Modularidade**: A decomposição em funções menores facilita o desenvolvimento e a manutenção.
- **Estruturas de Controle**: Laços (for, while) e condicionais (if, else) são amplamente utilizados para o controle do fluxo de execução.

3. ANÁLISE DA LINGUAGEM KOTLIN

3.1. CARACTERÍSTICAS DA LINGUAGEM:

3.1.1. Sintaxe e Semântica Básicas

Kotlin possui uma sintaxe moderna e concisa, desenhada para ser legível e eficiente. Herda muitas das estruturas da linguagem Java, mas com melhorias significativas:

- Inferência de tipo: Embora o tipo das variáveis possa ser declarado explicitamente, Kotlin oferece inferência automática, reduzindo a necessidade de declarações repetitivas.
- Interoperabilidade com Java: Kotlin é 100% compatível com Java, permitindo que código Java seja usado diretamente em projetos Kotlin e vice-versa.
- Null Safety: Kotlin possui um sistema de tipos que elimina a possibilidade de referências nulas (null), com tipos explícitos nullable e operadores que garantem a segurança ao lidar com possíveis valores nulos.

3.1.2. Tipagem

- Tipagem estática e forte: Kotlin possui uma tipagem estática, o que significa que os tipos das variáveis são conhecidos e verificados em tempo de compilação. Além disso, a tipagem é forte, ou seja, há poucas ou nenhuma conversão implícita entre tipos diferentes, prevenindo erros de tipo comuns em linguagens de tipagem fraca.
- Nullable Types: Uma característica importante de Kotlin é a distinção entre tipos "nullable" (String?) e não-nullables (String), o que força o programador a lidar explicitamente com a presença ou ausência de valores nulos.

3.1.3. Gerenciamento de Memória

Kotlin, quando executado na JVM, utiliza o coletor de lixo (Garbage Collection) da própria máquina virtual Java, que gerencia automaticamente a alocação e desalocação de memória. Quando compilado para código nativo (usando Kotlin Native), ele usa gerenciamento de memória automático baseado no ARC (Automatic Reference Counting). Dessa forma, o

desenvolvedor não precisa se preocupar diretamente com alocação e liberação de memória, mas sim com otimizações em cenários específicos.

3.1.4. Outros Aspectos Relevantes

- Data Classes: Kotlin facilita a criação de classes que representam apenas dados, com geração automática de métodos como equals(), hashCode(), toString(), e copy().
- Extension Functions: Permite adicionar funcionalidades a classes existentes sem modificá-las, uma funcionalidade poderosa para aumentar a legibilidade e organização do código.
- Coroutines: Kotlin oferece suporte nativo a corrotinas, o que simplifica a programação assíncrona e concorrente, tornando mais fácil escrever código não bloqueante.

3.2. POPULARIDADE E APLICAÇÕES

3.2.1. Popularidade Atual

Kotlin tem se destacado como uma das linguagens de programação mais populares, especialmente após ser adotada como a linguagem oficial para o desenvolvimento Android pelo Google, em 2017. O crescimento da linguagem também é impulsionado por sua interoperabilidade com Java, o que facilita a migração de projetos existentes.

De acordo com pesquisas recentes (Stack Overflow e JetBrains), Kotlin está entre as linguagens mais amadas por desenvolvedores, especialmente devido à sua modernidade e simplicidade em relação a Java, mantendo a robustez da JVM.

3.3. ÁREAS E APLICAÇÕES

- Desenvolvimento Android: Kotlin se tornou a principal linguagem para desenvolvimento de aplicativos Android, substituindo Java como a escolha preferida por muitos desenvolvedores.

- Back-end: Com frameworks como Ktor e Spring Boot, Kotlin também é utilizado para o desenvolvimento de APIs e aplicações web, aproveitando a interoperabilidade com o ecossistema Java.
- Multiplataforma: Kotlin Multiplatform permite a criação de código que pode ser reutilizado em diferentes plataformas, como Android, iOS, e Web, facilitando o desenvolvimento de soluções móveis e front-end.
- Aplicações nativas: Através do Kotlin Native, a linguagem também pode ser usada para compilar aplicações que não dependem da JVM, sendo utilizadas em desenvolvimento para iOS, sistemas embarcados, e ambientes onde a JVM não é viável.

3.4. PARADIGMAS INCORPORADOS

3.4.1. Suporte a Paradigmas de Programação

Kotlin suporta vários paradigmas de programação, o que a torna uma linguagem flexível para diferentes tipos de projetos:

- Orientação a Objetos (OO): Como Java, Kotlin é totalmente orientada a objetos, permitindo a criação de classes, herança, interfaces e encapsulamento. Sua abordagem moderna oferece melhor gerenciamento de classes de dados e métodos inline.
- Programação Funcional: Kotlin suporta o paradigma funcional, oferecendo funções de alta ordem, lambdas e funções puras, além de permitir o uso de imutabilidade. A programação funcional em Kotlin é fortemente integrada com a orientação a objetos, permitindo uma abordagem mista.
- Programação Imperativa: Kotlin, como Java, também é uma linguagem imperativa, onde os desenvolvedores podem escrever instruções sequenciais e manipular estados diretamente.

3.5. IMPLEMENTAÇÃO DOS PARADIGMAS

- Programação Funcional em Kotlin: Funções podem ser tratadas como valores, passadas como argumentos ou retornadas de outras funções. Suporte a map, filter, reduce e outras operações de coleção fazem parte do arsenal funcional da linguagem.
- Programação Orientada a Objetos: Kotlin mantém os princípios fundamentais da orientação a objetos, como classes, herança e polimorfismo, mas com aprimoramentos, como a remoção de códigos verbosos que existem no Java.
- Programação Imperativa: O uso de loops, variáveis mutáveis e instruções condicionais é comum em Kotlin, especialmente quando se trata de interoperabilidade com Java ou em cenários onde a programação funcional não é a escolha ideal.

Kotlin é uma linguagem versátil, moderna e eficiente, que oferece uma mistura de paradigmas e características que a tornam ideal para diversos tipos de aplicações, desde o desenvolvimento Android até sistemas back-end robustos. Sua combinação de segurança de tipos, sintaxe expressiva e interoperabilidade com Java facilita a adoção por desenvolvedores que buscam produtividade sem sacrificar a performance.

4. EXECUÇÃO DO CÓDIGO

Kotlin é uma linguagem compilada, o que significa que o código-fonte é convertido em bytecode, que é posteriormente executado na JVM (Java Virtual Machine). No entanto, Kotlin também possui a capacidade de ser compilada para JavaScript ou nativo, dependendo da plataforma. Isso a torna uma linguagem híbrida, pois pode ser executada tanto em ambientes de máquina virtual quanto nativos.

4.1. OTIMIZAÇÕES DO COMPILADOR

O compilador Kotlin utiliza otimizações padrão da JVM, como:

- Inlined functions: Funções inline podem ser otimizadas, eliminando a sobrecarga de chamadas de função.
- Escaping analysis: Otimizações que evitam a criação de objetos desnecessários na heap.
- Smart casts: O compilador detecta automaticamente os tipos apropriados, tornando o código mais eficiente.

Essas otimizações melhoram a performance do código sem comprometer a legibilidade.

5. EXEMPLOS

5.1. EXEMPLO 1: SOMA DE NÚMEROS EM UMA LISTA

TrabalhoLP > Ex1.kt

```kotlin
fun sumList(numbers: List<Int>): Int {
    var sum = 0
    for (number in numbers) {
        sum += number
    }
    return sum
}

fun main() {
    val numbers = listOf(1, 2, 3, 4, 5)
    val result = sumList(numbers)
    println("A soma dos números é: $result")
}
```

Este exemplo demonstra uma função simples que percorre uma lista de números e acumula a soma. O fluxo de execução é direto e sequencial, típico do paradigma procedural.

5.2. EXEMPLO 2: ENCONTRAR O MAIOR NÚMERO

TrabalhoLP > Ex2.kt

```kotlin
fun findMax(numbers: List<Int>): Int {
    var max = numbers[0]
    for (number in numbers) {
        if (number > max) {
            max = number
        }
    }
    return max
}

fun main() {
    val numbers = listOf(3, 9, 2, 8, 6)
    val maxNumber = findMax(numbers)
    println("O maior número é: $maxNumber")
}
```

Aqui, a função percorre uma lista de números para encontrar o maior valor. Novamente, o fluxo segue uma sequência clara de etapas.

5.3. EXEMPLO 3: VERIFICAÇÃO DE NÚMERO PRIMO

TrabalhoLP > Ex3.kt

```kotlin
fun isPrime(number: Int): Boolean {
    if (number < 2) return false
    for (i in 2 until number) {
        if (number % i == 0) {
            return false
        }
    }
    return true
}

fun main() {
    val num = 17
    if (isPrime(num)) {
        println("$num é um número primo")
    } else {
        println("$num não é um número primo")
    }
}

```

Este exemplo ilustra a verificação de um número primo, com instruções executadas sequencialmente até encontrar um divisor.

5.4. EXEMPLO 4: CÁLCULO DO FATORIAL

TrabalhoLP > Ex4.kt

```kotlin
fun factorial(n: Int): Int {
    var result = 1
    for (i in 2..n) {
        result *= i
    }
    return result
}

fun main() {
    val num = 5
    val fact = factorial(num)
    println("O fatorial de $num é: $fact")
}

```

O cálculo do fatorial é um clássico do paradigma procedural, utilizando um loop para multiplicar os números em sequência.

5.5. EXEMPLO 5: VERIFICAÇÃO DE PALÍNDROMO

TrabalhoLP > Ex5.kt

```kotlin
fun isPalindrome(word: String): Boolean {
    val cleanedWord = word.replace("\\s".toRegex(), "").toLowerCase()
    val reversedWord = cleanedWord.reversed()
    return cleanedWord == reversedWord
}

fun main() {
    val word = "arara"
    if (isPalindrome(word)) {
        println("$word é um palíndromo")
    } else {
        println("$word não é um palíndromo")
    }
}

```

A função verifica se uma palavra é um palíndromo, removendo espaços e comparando-a com sua versão invertida.

5.6. EXEMPLO 6: CALCULADORA

TrabalhoLP > Ex6.kt

```kotlin
// Função principal, ponto de entrada do programa
fun main() {
    // Exibe as instruções para o usuário
    println("Bem-vindo à Calculadora em Kotlin!")
    println("Escolha uma operação:")
    println("1. Adição (+)")
    println("2. Subtração (-)")
    println("3. Multiplicação (*)")
    println("4. Divisão (/)")

    // Lê a escolha do usuário
    print("Digite o número da operação: ")
    val operacao = readLine()?.toIntOrNull() ?: -1  // Lê e converte para Int, ou retorna -1 se for inválido

    // Lê os dois números do usuário
    print("Digite o primeiro número: ")
    val numero1 = readLine()?.toDoubleOrNull() ?: 0.0  // Lê e converte para Double
    print("Digite o segundo número: ")
    val numero2 = readLine()?.toDoubleOrNull() ?: 0.0  // Lê e converte para Double

    // Variável para armazenar o resultado da operação
    var resultado: Double = 0.0

    // Estrutura condicional procedural para decidir qual operação executar
    when (operacao) {
        1 -> resultado = adicao(numero1, numero2)  // Chama a função de adição
        2 -> resultado = subtracao(numero1, numero2)  // Chama a função de subtração
        3 -> resultado = multiplicacao(numero1, numero2)  // Chama a função de multiplicação
        4 -> resultado = divisao(numero1, numero2)  // Chama a função de divisão
        else -> println("Operação inválida!")  // Caso o usuário escolha uma operação inválida
    }

    // Exibe o resultado se a operação for válida
    if (operacao in 1..4) {
        println("O resultado da operação é: $resultado")
    }
}
```

5.6.1. Funcionamento Do Programa

1. **Função principal (main)**:

Este é o ponto de entrada do programa, seguindo o estilo procedural. Todas as operações do programa começam a partir daqui, como a leitura dos dados do usuário, o controle de fluxo (condicionais when) e a chamada para funções auxiliares.

2. **Entrada do usuário**:

- O programa lê a operação desejada e os números do usuário através da função readLine(). Essa é a maneira básica de lidar com a entrada em Kotlin.
- O operador e os números são convertidos para seus tipos adequados (Int para a operação e Double para os números).

3. **Estrutura de controle (when)**:

O programa usa uma estrutura condicional procedural para determinar qual operação será executada com base na escolha do usuário.

- Funções de operações matemáticas:
- Funções como adicao(), subtracao(), multiplicacao() e divisao() são funções que seguem o paradigma procedural clássico: cada uma realiza uma tarefa específica, recebe parâmetros e retorna um resultado.
- Cada função está isolada e apenas executa sua operação, tornando o código mais modular e fácil de entender.

4. **Divisão por zero**:

A função divisao() tem um controle básico para evitar a divisão por zero. Se o divisor for zero, o programa exibe uma mensagem de erro e retorna 0.0.

5. **Principais aspectos do paradigma procedural aplicados:**

- Procedimentos (funções): O código é organizado em procedimentos que recebem entradas e retornam saídas, sem interações com objetos ou estado interno, típico do paradigma procedural.
- Controle de fluxo: Usa estruturas condicionais (when) para controlar o fluxo do programa.
- Simplicidade: Cada função realiza uma única tarefa bem definida, sem criar dependências complexas entre as partes do código.

5.6.2. Saida

```
PS C:\Users\Belshoff\projects\trabalholp> code Ex6.kt
PS C:\Users\Belshoff\projects\trabalholp> kotlinc Ex6.kt -include-runtime -d Ex6.jar
PS C:\Users\Belshoff\projects\trabalholp> java -jar Ex6.jar
Bem-vindo à Calculadora em Kotlin!
Escolha uma operação:
1. Adição (+)
2. Subtração (-)
3. Multiplicação (*)
4. Divisão (/)
Digite o número da operação: 1
Digite o primeiro número: 2
Digite o segundo número: 2
O resultado da operação é: 4.0
PS C:\Users\Belshoff\projects\trabalholp> java -jar Ex6.jar
Bem-vindo à Calculadora em Kotlin!
Escolha uma operação:
1. Adição (+)
2. Subtração (-)
3. Multiplicação (*)
4. Divisão (/)
Digite o número da operação: 3
Digite o primeiro número: 3
Digite o segundo número: 2
O resultado da operação é: 6.0
PS C:\Users\Belshoff\projects\trabalholp>
```

6. PARSE TREE EM KOTLIN

O código fornecido constrói uma árvore sintática de uma expressão matemática, decompondo-a em nós que representam números e operações. Essa árvore é essencial para entender a estrutura de uma expressão e calcular seu valor.

6.1. CÓDIGO: PARSE TREE

TrabalhoLP > Parsetree.kt

```kotlin
// Classe base para os nós da árvore sintática
sealed class Node
data class NumberNode(val value: Int) : Node()
data class OperationNode(val operator: Char, val left: Node, val right: Node) : Node()

// Função responsável por analisar a expressão e montar a árvore sintática
fun parseExpression(expression: String): Node {
    val tokens = expression.filter { !it.isWhitespace() }.toList() // Remove os espaços em branco e converte para uma lista de caractere
    return parseAdditionSubtraction(tokens.toMutableList())
}

// Função que trata a análise de adição e subtração, com base na precedência de operadores
fun parseAdditionSubtraction(tokens: MutableList<Char>): Node {
    var node = parseMultiplicationDivision(tokens)

    while (tokens.isNotEmpty() && (tokens.first() == '+' || tokens.first() == '-')) {
        val operator = tokens.removeAt(0)
        val right = parseMultiplicationDivision(tokens)
        node = OperationNode(operator, node, right)
    }

    return node
}

// Função para tratar multiplicação e divisão, considerando a precedência desses operadores
fun parseMultiplicationDivision(tokens: MutableList<Char>): Node {
    var node = parseParenthesesOrNumber(tokens)

    while (tokens.isNotEmpty() && (tokens.first() == '*' || tokens.first() == '/')) {
        val operator = tokens.removeAt(0)
        val right = parseParenthesesOrNumber(tokens)
        node = OperationNode(operator, node, right)
    }

    return node
}
```

```
    return node
}

// Função que lida com parênteses ou números
fun parseParenthesesOrNumber(tokens: MutableList<Char>): Node {
    if (tokens.first() == '(') {
        tokens.removeAt(0) // Remove '('
        val node = parseAdditionSubtraction(tokens)
        tokens.removeAt(0) // Remove ')'
        return node
    }

    return parseNumber(tokens)
}

// Função responsável por tratar números na expressão
fun parseNumber(tokens: MutableList<Char>): Node {
    val number = StringBuilder()

    while (tokens.isNotEmpty() && tokens.first().isDigit()) {
        number.append(tokens.removeAt(0))
    }

    return NumberNode(number.toString().toInt())
}

// Função para imprimir a árvore sintática de forma hierárquica, com indentação para facilitar a visualização
fun printSyntaxTree(node: Node, level: Int = 0) {
    val indent = " ".repeat(level * 2) // Cria uma indentação proporcional ao nível da árvore
```

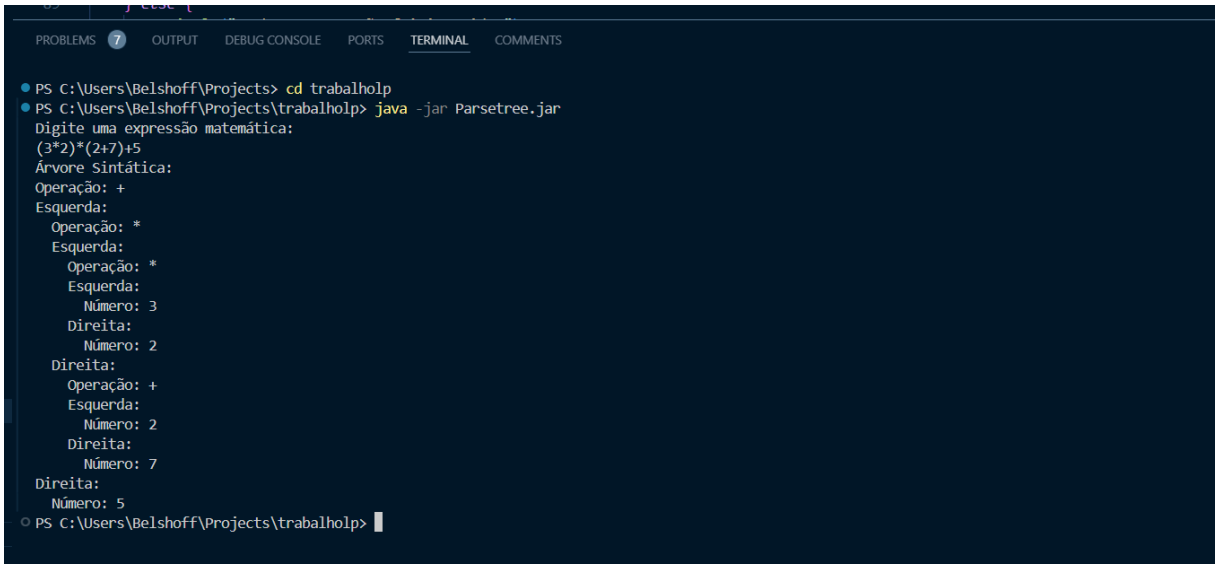
```
// Função para imprimir a árvore sintática de forma hierárquica, com indentação para facilitar a visualização
fun printSyntaxTree(node: Node, level: Int = 0) {
    val indent = " ".repeat(level * 2) // Cria uma indentação proporcional ao nível da árvore

    when (node) {
        is NumberNode -> println("${indent}Número: ${node.value}")
        is OperationNode -> {
            println("${indent}Operação: ${node.operator}")
            println("${indent}Esquerda:")
            printSyntaxTree(node.left, level + 1)
            println("${indent}Direita:")
            printSyntaxTree(node.right, level + 1)
        }
    }
}

// Função principal que captura a expressão inserida pelo usuário e exibe a árvore sintática
fun main() {
    println("Digite uma expressão matemática:")
    val expression = readLine() ?: ""  // Captura a expressão fornecida pelo usuário
    if (expression.isNotBlank()) {
        try {
            val syntaxTree = parseExpression(expression)
            println("Árvore Sintática:")
            printSyntaxTree(syntaxTree)
        } catch (e: Exception) {
            println("Erro ao analisar a expressão: ${e.message}")
        }
    } else {
        println("Nenhuma expressão foi inserida.")
    }
}
```

6.2. SAIDA

Abaixo podemos ver o código em funcionamento, construindo a árvore sintática da expressão (3*2)*(2+7)+5. Este código nos permite ainda informar a expressão que quisermos e ele irá construir a árvore sintática respeitando as precedências de operadores e parênteses.



A função parseExpression analisa uma expressão fornecida pelo usuário e cria uma árvore de operações, onde cada nó pode ser um número (NumberNode) ou uma operação (OperationNode). As operações são processadas de acordo com a precedência de operadores, garantindo que multiplicações e divisões sejam avaliadas antes de adições e subtrações.

6.2.1. Explicação do Funcionamento

- Tokens: A expressão é dividida em tokens, removendo espaços em branco.
- Análise: Cada operação é analisada de acordo com sua precedência. Funções como parseAdditionSubtraction e parseMultiplicationDivision garantem que a árvore seja construída corretamente.
- Árvore: A árvore é construída hierarquicamente, com cada operação sendo um nó, e seus operandos (números ou sub-expressões) formando os nós filhos.

Este código demonstra como expressões podem ser estruturadas e avaliadas de maneira eficiente, sendo uma abordagem comum em compiladores e interpretadores.

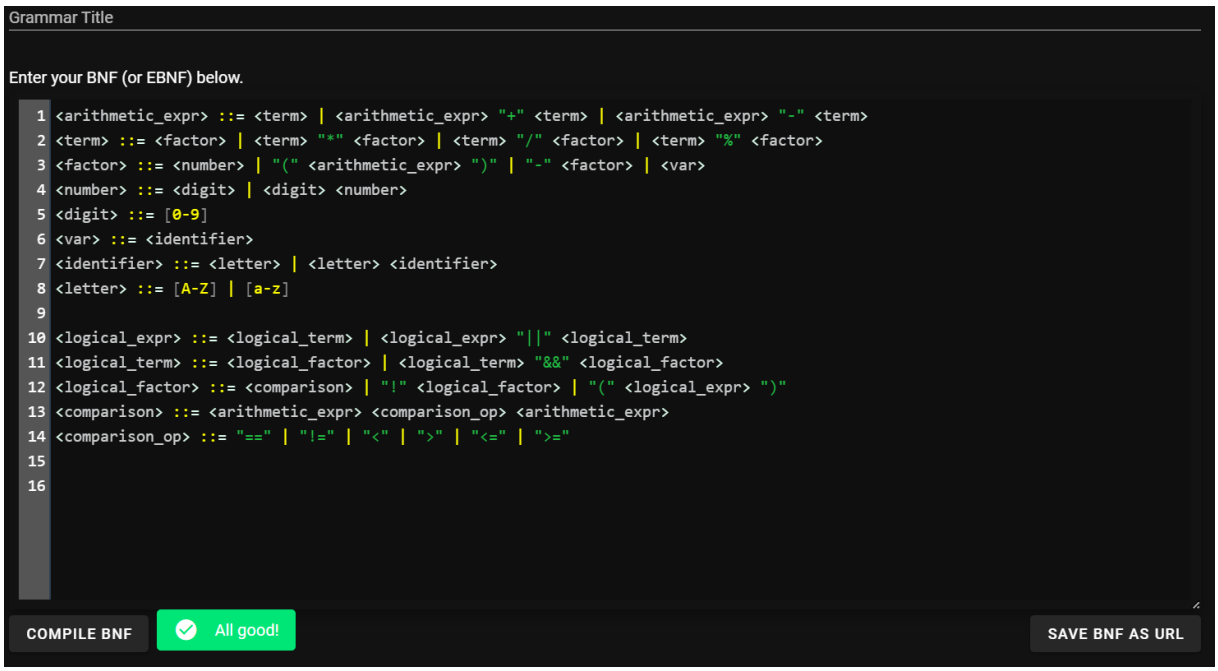
7. GRAMÁTICA BNF PARA EXPRESSÕES MATEMÁTICAS E LÓGICAS EM KOTLIN

A seguir, a gramática BNF (Backus-Naur Form) que descreve as expressões matemáticas e lógicas em Kotlin, incluindo a ordem de precedência dos operadores aritméticos e lógicos.

7.1. EXPRESSÕES MATEMATICAS

- Soma e subtração (+, -) têm a menor precedência entre os operadores aritméticos.
- Multiplicação, divisão e módulo (*, /, %) têm precedência superior à soma e subtração.
- Parênteses alteram explicitamente a ordem de avaliação.

7.2. EXPRESSOES LÓGICAS



- Negação lógica (!) tem a maior precedência nas expressões lógicas.
- E lógico (&&) tem precedência maior que OU lógico (||).
- Operadores de comparação (==, !=, <, >, <=, >=) têm precedência entre as expressões aritméticas e os operadores lógicos.

7.3. ORDEM DE PRECEDÊNCIA DOS OPERADORES

Em Kotlin, a ordem de precedência dos operadores segue o padrão de muitas linguagens modernas:

- Unários: +, -, !
- Multiplicação, Divisão, Módulo: *, /, %
- Adição e Subtração: +, -
- Comparações: ==, !=, <, >, <=, >=
- E Lógico: &&
- OU Lógico: ||

Os parênteses podem ser usados para alterar a precedência padrão dos operadores, garantindo que partes específicas da expressão sejam avaliadas primeiro.

8. CONCLUSÃO

O paradigma procedural em Kotlin oferece uma abordagem simples e direta para resolução de problemas que envolvem a execução sequencial de instruções. Embora Kotlin seja amplamente usada com orientação a objetos e programação funcional, o paradigma procedural é útil para tarefas mais simples e quando se busca clareza na execução.

Kotlin, sendo uma linguagem compilada com suporte a múltiplos paradigmas, combina eficiência e segurança, com otimizações no compilador que garantem um desempenho competitivo. O uso de árvores sintáticas, como no exemplo do parse tree, demonstra a versatilidade de Kotlin na construção de ferramentas complexas, como analisadores de expressões.