CENTRO UNIVERSITÁRIO ADVENTISTA DE SÃO PAULO

Carolaine Almeida Silva Castro - 007241
Felipe Kadri de Oliveira - 061344
Bruna de Paula Cordeiro - 058967
Nicolas Ferreira Perejon Amorim - 057701
Matheus Camilo Velanzuela Meira - 011628
Fábio Zanin Conegundes de Araújo - 062188

SÃO PAULO –SP

2024

Documentação do Analisador Léxico para Bugscript

1. Introdução

Este documento descreve o funcionamento do analisador léxico da linguagem **Bugscript**, que tem como objetivo identificar e classificar tokens a partir de sua sintaxe. Os tokens são agrupados em categorias como palavras-chave, operadores e símbolos.

2. Estrutura do Código

O código está organizado em duas seções principais: **KW** (palavras-chave) e **simbols** (símbolos).

2.1 Palavras-Chave (KW)

2.1.1 Variáveis (var)

Define os tipos de variáveis disponíveis na linguagem:

- ***cryInt*** : Tipo inteiro (*int*);
- ***cryString*** : Tipo string (*str*);
- ***cryBool*** : Tipo booleano (*bool*);
- ***[]*** : Tipo lista (*list*);

2.1.2 Controle de Fluxo (flow)

Inclui comandos de controle de fluxo:

- ***goAway*** : Estrutura de repetição (*while*)
- ***bugCheck*** : Estrutura condicional (*if*)
- ***endCheck*** : Fim de if condicional
- ***endBugCheck*** : Alternativa na estrutura condicional (*else*)
- ***flyAway*** : Comando para interromper um loop (*break*)

2.1.3 Operadores Aritméticos (aritimical_operator)

Define operadores aritméticos:

- ***moreBug*** : Operador de adição (*ADD*)
- ***minusBug*** : Operador de subtração (*MINUS*)

2.1.4 Valores Lógicos (logical)

Valores lógicos na linguagem:

- ***true*** : Verdadeiro
- ***false*** : Falso

2.1.5 Entrada e Saída (IO)

Comandos para entrada e saída de dados:

- ***inBug*** : Comando de entrada (*INPUT*)
- ***outBug*** : Comando de saída (*OUTPUT*)

2.2 Símbolos (simbols)

Esta seção define os símbolos utilizados na linguagem:

- *STRING* : "
- *COMMENT* : //
- *OPEN_BRACKET* : [
- *CLOSE_BRACKET* :]
- *LESS_THAN* : <
- *MORE_THAN* : >
- *ATRIBUTION* : =
- *START_BLOCK* : {
- *END_BLOCK* : }
- *END_STATEMENT* : ;
- *OPEN_PARENTESIS* : (
- *CLOSE_PARENTESIS* :)

3. Conclusão

Este analisador léxico é uma ferramenta fundamental para a interpretação da linguagem **Bugscript**, permitindo que tokens sejam reconhecidos e categorizados para processamento posterior. A estrutura de palavras-chave e símbolos define claramente a sintaxe e a semântica do código.

Documentação do Analisador Semântico (Semantic Analyzer)

1. Visão Geral

O Analisador Semântico do BugScript, implementado na classe SemanticAnalyzer, é responsável por verificar a validade semântica do código, garantindo que o programa faça sentido logicamente. Ele trabalha sobre a Árvore Sintática Abstrata (AST) gerada pelo Analisador Sintático, verificando tipos de variáveis, declarações, atribuições e operações.

Classe: ***SemanticAnalyzer***

```
def __init__(self)
```

Inicializa o analisador semântico, criando uma instância da SymbolTable que gerencia as declarações de variáveis, atribuições e tipos.

1.1 Método Principal

```
def analyze(self, ast)
```

Realiza a análise semântica da AST. Para cada nó da árvore, chama o método visit.
Método de Visita

```
def visit(self, node)
```

Determina o tipo de declaração ou expressão que o nó representa e direciona para o método de tratamento adequado.

Métodos de Tratamento

Declaração de Variáveis

```
def handle_var_declaration(self, node)
```

Verifica e declara variáveis no escopo atual. Se a variável possuir um valor inicial, este valor é validado e atribuído.

var_name: Nome da variável.
var_type: Tipo da variável (int, float, string, bool, etc.).
initializer: Expressão que define o valor inicial da variável (se houver).

Atribuição de Variáveis

def handle_assignment(self, node)

Verifica a atribuição de valor a uma variável já declarada e compatibiliza o tipo do valor com o tipo da variável.

Estrutura Condicional: *(If-Else)*

def handle_if(self, node)

Valida a condição do if, garantindo que ela resulte em um valor booleano. Em seguida, analisa os blocos de código then_branch e else_branch.

Estrutura de Repetição (While)

def handle_while(self, node)

Valida a condição do while (espera-se um booleano) e verifica o corpo do loop.
Declaração de Saída (Output)

def handle_output(self, node)

Avalia a expressão de saída para garantir que ela seja válida.

Avaliação de Expressões

def evaluate_expression(self, expr)

Avalia expressões e retorna o tipo do valor resultante. Suporta:

LITERALS: Valores literais como números, booleanos e strings.
VARIABLES: Verifica o valor da variável.
BINARY: Avalia operações binárias, como adição e comparação, e garante compatibilidade de tipos.
CALL: Verifica e executa chamadas de funções pré-definidas (adição, subtração, etc.).
Funções

```
def call_function(self, function_name, arguments)
```

Avalia e valida as chamadas de funções. As operações aritméticas (como soma e multiplicação) são tratadas aqui.

Integração com a Tabela de Símbolos

O SemanticAnalyzer utiliza a classe SymbolTable para armazenar informações sobre as variáveis (nome, tipo e status de inicialização). Também verifica:

Declarações de variáveis (com o método declare).

Atribuições (com o método assign). Compatibilidade de tipos em operações (como aritmética e lógica). Tratamento de Erros Semânticos Os erros semânticos são lançados por meio de exceções SemanticError. Quando uma violação de regra é detectada (como uma atribuição de tipo incorreto), o analisador interrompe o processo com uma mensagem descritiva.

Conclusão

O Analisador Semântico do BugScript é uma etapa crítica que garante a integridade lógica do código, assegurando que tipos de dados sejam compatíveis, variáveis sejam corretamente declaradas e utilizadas, e que todas as expressões e operações sejam válidas.