

# **Processamento de Linguagens e Compiladores**

Ano letivo 2024/2025

Daniel Andrade

A100057

José Silva

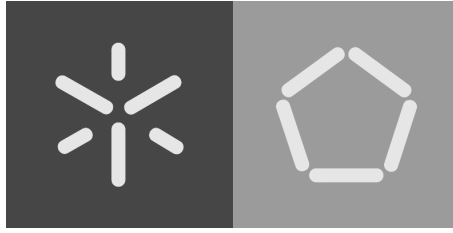
100105

Pedro Malainho

100050

Universidade do Minho  
Escola de Engenharia  
Departamento de Informática

Novembro 2024



# Processamento de Linguagens e Compiladores

Ano letivo 2024/2025

Relatório do Trabalho Prático 2  
Licenciatura em Ciências da Computação

Coordenador Prof. Pedro Rangel Henriques

Novembro 2024

# Resumo

Este relatório descreve o desenvolvimento de uma linguagem de programação e do seu respectivo compilador, criado no âmbito do segundo trabalho prático da unidade curricular de Processamento de Linguagens e Compiladores (PLC). O projeto utilizou os módulos `lex` e `ply` do Python para implementar uma gramática capaz de traduzir a linguagem para código Assembly.

Ao longo deste relatório, explicamos de forma clara as decisões tomadas, as produções da gramática implementadas e o processo de desenvolvimento do compilador.

# Índice

<b>Resumo .....</b>	<b>3</b>
<b>1. Introdução .....</b>	<b>1</b>
<b>2. Enunciado .....</b>	<b>2</b>
<b>3. Linguagem Go .....</b>	<b>3</b>
3.1. Declarações e Atribuições .....	3
3.2. Operações aritméticas, lógicas e condicionais .....	4
3.3. Estruturas de controlo de execução .....	5
3.4. Instruções de Repetição .....	7
3.5. Input e Output .....	8
3.6. Funções .....	9
3.7. Comentários .....	10
<b>4. Código .....</b>	<b>11</b>
4.1. Analisador Léxico .....	11
4.2. Analisador Sintático (Parser) .....	12
4.3. Analisador Semântico .....	12
4.4. Gerador de Código .....	13
<b>5. Detalhes Técnicos .....</b>	<b>14</b>
5.1. Gramática .....	14
5.2. Tabelas de Símbolos .....	14
5.3. Regras de Precedência .....	14
5.4. Tipos de Dados .....	15
5.5. Estruturas de Controlo .....	16
<b>6. Testes .....</b>	<b>17</b>
<b>7. Conclusão .....</b>	<b>32</b>
<b>8. Anexos .....</b>	<b>33</b>
8.1. Compiladores 101 .....	33
8.2. Lexer .....	34
8.3. Parser .....	37
8.4. Semantic Analyzer .....	49
8.5. Code Generator .....	66
8.6. Symbol Table .....	85

# 1. Introdução

O desenvolvimento de linguagens de programação e suas ferramentas associadas, como compiladores e interpretadores, exige o domínio de conceitos essenciais, como análise léxica e análise sintática. Esses processos, conduzidos pelo lexer e pelo parser, desempenham um papel fundamental ao transformar o código-fonte em estruturas organizadas e compreensíveis para execução ou análise.

O lexer é responsável por dividir o código em unidades mínimas chamadas tokens, enquanto o parser organiza esses tokens de acordo com as regras da gramática da linguagem, construindo representações hierárquicas, como árvores sintáticas. Juntas, essas etapas são indispensáveis para validar e transformar o código-fonte.

A compilação, que engloba tanto a análise léxica quanto a sintática, pode ser vista como um processo em múltiplas etapas, no qual o código-fonte é progressivamente transformado até ser convertido em código de máquina. Um diagrama representando esse fluxo pode ser encontrado no **Anexo 1**

Este relatório tem como objetivo apresentar o funcionamento e analisar a implementação de um lexer e um parser, destacando sua importância e aplicação no contexto de uma linguagem específica.

## Estrutura do Relatório

Além desta introdução, o relatório está organizado da seguinte forma: o **Capítulo 2** apresenta o enunciado do projeto; o **Capítulo 3** descreve o design da linguagem e a sua sintaxe; o **Capítulo 4** aborda a implementação do Analisador Léxico e Sintático; o capítulo **Capítulo 5** apresenta os detalhes técnicos do projeto; o **Capítulo 6** apresenta vários testes realizados pelo grupo e por fim, o **Capítulo 7** apresenta as conclusões do projeto.

## Ferramentas Utilizadas

- [Visual Studio Code \(VScode\)](#)
- [PyCharm](#)
- [Python 3.12](#)
- [PLY \(Phyton Lex-Yacc\)](#)
- [Typst](#)
- [EWVM](#)

## 2. Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto. Apenas deve ter em consideração que essa linguagem terá de permitir:

- declarar variáveis atômicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas;
- efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis;
- ler do standard input e escrever no standard output;
- efetuar instruções condicionais para controlo do fluxo de execução;
- efetuar instruções de repetição (cíclicas) para controlo do fluxo de execução, permitindo o seu aninhamento. Note que deve implementar pelo menos o ciclo **while-do**, **repeat-until** ou **for-do**.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes.

- declarar e manusear variáveis estruturadas do tipo array (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado do tipo inteiro.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explícito, o valor da variável após a declaração é 0 (zero). Desenvolva, então, um compilador para essa linguagem com base na **GIC** criada acima e com recurso aos modelos Yacc/ Lex do PLY/ Python. O compilador deve gerar **pseudo-código**, Assembly da Máquina Virtual **VM**.

Muito Importante:

Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código Assembly gerado bem como o programa a correr na máquina virtual VM.

## 3. Linguagem Go

A sintaxe é especificada usando Extended Backus-Naur Form

### 3.1. Declarações e Atribuições

Sintaxe:

```
var_declaration = VAR ID type EQUAL expression SEMICOLON
                | VAR ID type SEMICOLON
                | VAR ID type LBRACKET RBRACKET EQUAL array_initializer SEMICOLON
                | VAR ID type LBRACKET NUMBER_LITERAL RBRACKET SEMICOLON
                | VAR ID type LBRACKET NUMBER_LITERAL RBRACKET
                  EQUAL array_initializer SEMICOLON ;

short_var_declaration = ID WALRUS expression SEMICOLON ;
```

```
var variableName type = value ;
var variableName type ;
var variableName type[] = {value1, value2, value3, ...} ;
var variableName type[lenght] ;
var variableName type[lenght] = {value1, value2, value3, ...} ;

variableName := value ;
variableName := type[lenght]{value1, value2, value3, ...} ;
variableName := type[] {value1, value2, value3, ...} ;
variableName := type[lenght] ;
```

Exemplos:

```
var word1 string = "Hello" ;
var isStudent bool ;
var dayTimes string[] = {"Morning", "Evening", "Night"} ;
var intArr int[10] ;
var floatArr float[5] = {1.5, 2.5, 3.5, 4.5, 5.5} ;
var mArr float[2][2] = {{1, 2}, {2, 3}}

word2 := "World" ;
intArr2 := int[9]{100, 300, 200, 400, 600, 500, 700, 900, 800} ;
floatArr2 := float[]{5.5, 0.25, -103.342} ;
mArr2 := float[][]{{1, 2}, {3, 4}}
stringArr1 := string[5] ;
```

### 3.2. Operações aritméticas, lógicas e condicionais

```
sum = x + y           // Addition
sub = x - y           // Subtraction
mult = x * y          // Multiplication
div = x / y           // Division
mod = x % y           // Modulus
inc = x++             // Increment
dec = x--             // Decrement

x += 3                // Plus Equals
x -= 3                // Minus Equals
x *= 3
x /= 3
x %= 3

x == y               // Equal
x != y               // Not Equal
x > y                // Greater than
x < y                // Less than
x >= 3               // Greater than or Equal to
x <= 3               // Less than or Equal to

x < 5 && x < 10       // Logical And
x < 5 || x < 4         // Logical Or
!(x < 5 && x < 10)     // Logical Not
```



### 3.3. Estruturas de controlo de execução

Sintaxe:

```
if_statement = IF expression block else_clause ;
```

```
else_clause = ELSE if_statement  
             | ELSE block  
             | empty ;
```

```
switch_statement = SWITCH expression LBRACE switch_cases default_clause RBRACE ;
```

```
switch_cases = switch_cases switch_case  
              | switch_case ;
```

```
switch_case = CASE expression COLON statements ;
```

```
default_clause = DEFAULT COLON statements  
                | empty ;
```

```
if condition1 {  
    // code to be executed if condition1 is true  
}  
  
if condition1 {  
    // code to be executed if condition1 is true  
} else {  
    // code to be executed if condition1 is false  
}  
  
if condition1 {  
    // code to be executed if condition1 is true  
} else if condition2 {  
    // code to be executed if condition1 is false and condition2 is true  
} else {  
    // code to be executed if condition1 and condition2 are both false  
}  
  
switch a {  
    case condition1: expression1 ;  
    case condition2: expression2 ;  
    case condition3: expression3 ;  
}
```

Exemplos:

```
if 20 > 18 {  
    print("20 is greater than 18") ;  
}
```

```
time := 20 ;  
if (time < 18) {  
    print("Good day.") ;  
} else {  
    print("Good evening.") ;  
}
```

```
a := 14 ;  
b := 14 ;  
if a < b {  
    print("a is less than b.") ;  
} else if a > b {  
    print("a is more than b.") ;  
} else {  
    print("a and b are equal.") ;  
}
```

```
var day string ;  
var dayNumber int = 2 ;  
switch day {  
    case 1:  
        day = "Monday";  
        break;  
    case 2:  
        day = "Tuesday";  
        break;  
    case 3:  
        day = "Wednesday";  
        break;  
    default :  
        day = "Not a valid day" ;  
}
```

### 3.4. Instruções de Repetição

Sintaxe:

```
for_statement = FOR for_init SEMICOLON expression SEMICOLON for_post block  
               | FOR expression block  
               | FOR block ;
```

```
for_init = ID WALRUS expression  
           | VAR ID type EQUAL expression ;
```

```
for_post = assignment  
          | expression ;
```

```
for init ; cond; pos {  
    // code to be executed while condition is true  
}  
  
for cond {  
    // code to be executed while condition is true  
}  
  
for {  
    // code to be executed while condition is true  
}
```

Exemplos:

```
a := 1 ;  
for a > 10 {  
    fmt.Println("Today is day {a}") ;  
    a = a + 1 ;  
}  
  
sum := 0 ;  
for i := 20; i > 0; i-- {  
    sum = sum + i ;  
}  
  
for {  
    print("Infinite loop") ;  
}
```

### 3.5. Input e Output

Sintaxe:

O grupo considera o I/O como uma função. Desta forma "read" e "print" seguem a sintaxe de uma função.

```
var variableName string ;  
variableName = read() ; // read() returns a string  
  
print("STRING") ;  
print(variableName) ;  
print("STRING ", variableName, " STRING ", variableName, ...) ;
```

Exemplos:

```
arr1 := int[2][2]{{1, 2}, {3, 4}} ;  
  
print(arr1[0][0]) ; // 1  
  
var student1 string ;  
student1 = read() ;  
  
var age int ;  
age = toInt(read()) ;  
  
print(student1, " is a student who is ", age, " years old") ;
```

### 3.6. Funções

Sintaxe:

```
function_declaration = FUNC ID LPAREN parameters_opt RPAREN type block ;

parameters_opt = parameters
                | empty ;

parameters = parameters COMMA parameter
            | parameter ;

parameter = ID type ;

block = LBRACE block_contents RBRACE
       | LBRACE RBRACE ;

block_contents = statements ;

statements = statements statement
            | statement ;
```

```
func FunctionName() type {
    // code to be executed
}

func FunctionName(variableName type) type {
    // code to be executed
}
```

Exemplos:

```
func main() void {
    age := 20 ;
    if age >= 18 && age <= 60 {
        print("ELIGIBLE FOR THE PROGRAM") ;
    } else {
        print("NOT ELIGIBLE") ;
    }
}

func Abs(num int) int {
    if num < 0 {
        return -num;
    }
    return num;
}
```

### 3.7. Comentários

Comentários servem como documentação do programa. Existem duas formas:

Comentários de linha: começam com a sequência de caracteres `'//'` e terminam no final da linha.

```
// The code below will print Hello World
```

Comentários gerais: começam com a sequência de caracteres `'/*'` e terminam na primeira sequência subsequente de caracteres `'*/'`.

```
/* The code below will print Hello World  
to the screen, and it is amazing */
```

## 4. Código

O nosso trabalho está estruturado em quatro ficheiros distintos, um dedicado ao **Analisador Léxico**, outro ao **Analisador Sintático**, o terceiro dedicado ao **Analisador Semântico** e por último o **Gerador de Código**.

### 4.1. Analisador Léxico

O código do ficheiro **Lexer** está disponível no **Anexo 2**. Este ficheiro define os tokens e literals necessários, além de incluir a especificação das palavras reservadas, que são tratadas de forma distinta e não reconhecidas como identificadores. Para a identificação dessas palavras reservadas, foi utilizado um tuplo que as lista explicitamente.

```
reserved = [  
    # Keywords  
    'BREAK', 'CASE', 'CONTINUE', 'DEFAULT', 'ELSE', 'FOR',  
    'FUNC', 'IF', 'RETURN', 'SWITCH', 'VAR', 'WHILE',  
    # Types  
    'INT', 'FLOAT', 'STRING', 'BOOL', 'VOID',  
]  
  
@TOKEN(r'[a-zA-Z_][a-zA-Z_0-9]*')  
def t_ID(t):  
    t.type = t.value.upper() if  
        (t.value.islower() and t.value.upper() in reserved)  
    else "ID"  
    return t
```

Conforme a definição de Identificador (Id), as variáveis da linguagem devem obrigatoriamente começar com uma letra, podendo ser seguidas por outros caracteres comuns.

```
@TOKEN(r'//.*\n')  
def t_COMMENT_SINGLE(t):  
    t.lexer.lineno += 1  
  
@TOKEN(r'/\*(\[^\*]\|\\*\[^\*\/])\*\/')  
def t_COMMENT_MULTI(t):  
    t.lexer.lineno += t.value.count('\n')
```

Outro exemplo de uma regra definida foi para os comentários utilizando keyword pass. Esta regra encapsula qualquer texto que siga `//` para comentários de linha única, bem como qualquer texto delimitado por `/* ... */` para comentários de múltiplas linhas.

## 4. 2. Analisador Sintático (Parser)

O código do ficheiro **Parser** foi desenvolvido para interpretar e validar a estrutura do código-fonte com base na gramática definida. Ele utiliza as definições de tokens fornecidas pelo lexer para construir a árvore sintática e assegurar que o código esteja em conformidade com as regras definidas. O conteúdo completo do **Parser** código pode ser consultado em **Anexo 3**.

Principais Funcionalidades:

- Definição de Regras de Gramática: Inclui regras para programas, declarações globais, declarações de funções, declarações de variáveis, expressões, estruturas de controlo e operações de atribuição.
- Gestão de Precedência: Define a precedência e a associatividade dos operadores para resolver ambiguidades na análise de expressões.
- Tratamento de Erros Sintáticos: Identifica e reporta erros de sintaxe.

## 4. 3. Analisador Semântico

O analisador semântico verifica a correção semântica do código-fonte, garantindo que operações e declarações façam sentido dentro do contexto da linguagem. Envolve a verificação de tipos, scope de variáveis e funções, e a consistência nas operações. O código do ficheiro **Semantic Analyzer** encontra-se em **Anexo 4**.

Principais Funcionalidades:

- Tabela de Símbolos: Gerencia informações sobre variáveis e funções, incluindo seus tipos e scopes.
- Verificação de Tipos: Garante que as operações sejam realizadas entre tipos compatíveis, prevenindo erros como a tentativa de somar uma string com um inteiro.
- Verificação de Scope: Assegura que variáveis e funções são usadas dentro de seus scopes válidos, prevenindo referências a símbolos não declarados.
- Análise de Estruturas de Controle: Verifica condições em estruturas como if, for loop, while e switch, garantindo que sejam booleanas e que os casos de switch sejam consistentes.



#### 4. 4. Gerador de Código

O gerador de código encontra-se em **Anexo 5** e é responsável por converter a árvore sintática abstrata (AST) validada em um conjunto de instruções que podem ser executadas numa VM. Este processo utiliza técnicas de geração de código como alocação de registradores, gerenciamento de memória e implementação de estruturas de controle.

Principais Funcionalidades:

- Geração de Instruções: Cada nó da AST é transformado em instruções equivalentes, seja em linguagem de máquina ou em uma representação intermediária.
- Gerenciamento de Memória: O gerador cuida da alocação de endereços para variáveis globais e locais, organizando corretamente os frames de pilha para chamadas de funções.
- Implementação de Estruturas de Controle: Traduz estruturas como if, for e switch em saltos condicionais e incondicionais no código gerado.
- Controle de Funções: A tradução de funções inclui o retorno de valores, a passagem de parâmetros e a liberação da pilha após a execução.

## 5. Detalhes Técnicos

### 5.1. Gramática

A gramática definida para o parser utiliza a notação BNF (Backus-Naur Form) para especificar as regras de produção da linguagem. Inclui definições para programas, declarações globais, funções, variáveis, expressões e estruturas de controle.

### 5.2. Tabelas de Símbolos

A tabela de símbolos é uma estrutura de dados que armazena informações sobre identificadores (variáveis e funções) utilizados no código-fonte. É organizada em scopes hierárquicos, permitindo a verificação de declarações e acessos corretos.

```
class SymbolTable:
    def __init__(self):
        self.scopes: List[Dict[str, Symbol]] = [{}]

    def enter_scope(self):
        self.scopes.append({})

    def exit_scope(self):
        if len(self.scopes) > 1:
            self.scopes.pop()

    def define(self, name: str, symbol: Symbol) -> None:
        if name in self.scopes[-1]:
            raise SemanticError(f"Symbol '{name}' already defined in current scope")
        self.scopes[-1][name] = symbol

    def lookup(self, name: str) -> Optional[Symbol]:
        for scope in reversed(self.scopes):
            if name in scope:
                return scope[name]
        return None
```

### 5.3. Regras de Precedência

As regras de precedência definem a ordem na qual os operadores são avaliados nas expressões. Isto é crucial para resolver ambiguidades e garantir que expressões complexas sejam interpretadas corretamente.

```
precedence = (
    ("left", "CONDITIONALOR"),
    ("left", "CONDITIONALAND"),
    ("left", "EQUALITY", "NOTEQUAL"),
    ("left", "LESS", "LESSOREQUAL", "GREATER", "GREATEROREQUAL"),
    ("left", "PLUS", "MINUS"),
    ("left", "TIMES", "DIVIDE", "MODULO"),
    ("right", "NOT"),
    ("right", "UMINUS"),
)
```

#### Exemplo

Considere a expressão:

```
expr1 = (a + b \* c == d && e || f ).
```

Passo 1: \* b \\* c é avaliado primeiro devido à maior precedência de \\*.

Passo 2: \* a + (b \\* c) é avaliado em seguida.

Passo 3: \* (a + b \\* c) == d é avaliado.

Passo 4: \* ((a + b \\* c) == d) && e é avaliado.

Passo 5: \* (((a + b \\* c) == d) && e) || f é o resultado final.

## 5.4. Tipos de Dados

A linguagem suporta os seguintes tipos de dados básicos:

**Inteiro** (int): Representa números inteiros.

**Flutuante** (float): Representa números de ponto flutuante.

**Booleano** (bool): Representa valores lógicos **true** ou **false**.

**String** (string): Representa palavras.

**Vazio** (void): Indica a ausência de valor, utilizado principalmente em funções que não retornam nada.

**Arrays**: Estruturas de dados que armazenam múltiplos valores do mesmo tipo. (Podem ser multidimensionais)

## 5.5. Estruturas de Controlo

A linguagem implementa várias estruturas de controlo que permitem a criação de fluxos de execução condicional e repetitivo.

Condicional **if-else**: Executa blocos de código baseados em condições booleanas.

Repetição **for**: Executa blocos de código repetidamente com inicialização, condição e atualização ou só com **condição** (equivalente a um **while**).

Estrutura **switch-case**: Seleciona entre múltiplos caminhos de execução baseados no valor de uma expressão.

## 6. Testes

INPUT VARS		Go
1	<code>var a int = 42;</code>	
2	<code>var pi float = 3.14159;</code>	
3	<code>var s string = "Hello, World!";</code>	
4	<code>var b bool = true;</code>	
5	<code>var c int;</code>	
6		
7	<code>shortDeclaredVar := 75 + 25 * 2 + 50 / 2;</code>	
8	<code>boolVar := false;</code>	
9		
10	<code>var test float = toFloat(3);</code>	
11		
12	<code>var arr1 int[5] = {1, 2, 3, 4, 5};</code>	
13	<code>arr2 := int[]{1, 2, 3, 4, 5};</code>	

OUTPUT		EWVM
1	<code>// Global variable declarations</code>	
2	<code>// Global variable a at address 0</code>	
3	<code>pushi 42</code>	
4	<code>storeg 0</code>	
5	<code>// Global variable pi at address 1</code>	
6	<code>pushf 3.14159</code>	
7	<code>storeg 1</code>	
8	<code>// Global variable s at address 2</code>	
9	<code>pushs "Hello, World!"</code>	
10	<code>storeg 2</code>	
11	<code>// Global variable b at address 3</code>	
12	<code>pushi 1</code>	
13	<code>storeg 3</code>	
14	<code>// Global variable c at address 4</code>	
15	<code>pushi 0</code>	
16	<code>storeg 4</code>	
17	<code>// Global variable shortDeclaredVar at address 5</code>	
18	<code>pushi 75</code>	
19	<code>pushi 25</code>	
20	<code>pushi 2</code>	
21	<code>mul</code>	
22	<code>add</code>	
23	<code>pushi 50</code>	
24	<code>pushi 2</code>	
25	<code>div</code>	
26	<code>add</code>	

```

27 storeg 5
28 // Global variable boolVar at address 6
29 pushi 0
30 storeg 6
31 // Global variable test at address 7
32 pushi 3
33 itof
34 storeg 7
35 // Global variable arr1 at address 8
36 alloc 5 // Allocate array of total size 5
37 dup 1
38 pushi 0
39 pushi 1
40 storen
41 dup 1
42 pushi 1
43 pushi 2
44 storen
45 dup 1
46 pushi 2
47 pushi 3
48 storen
49 dup 1
50 pushi 3
51 pushi 4
52 storen
53 dup 1
54 pushi 4
55 pushi 5
56 storen
57 storeg 8
58 // Global variable arr2 at address 9
59 pushi 1
60 storeg 9
61 start // Program start
62 stop // Program end

```

#### INPUT SWICTH CASE

 Go

```

1 func main() void {
2     var dayNumber int;
3     var day string;
4     var flag string = " IS ";
5     print("INSERT A NUMBER [1-7]: ");
6     dayNumber = toInt(read());
7

```

```

8  switch dayNumber {
9      case 1:
10         day = "MONDAY";
11         break;
12     case 2:
13         day = "TUESDAY";
14         break;
15     case 3:
16         day = "WEDNESDAY";
17         break;
18     case 4:
19         day = "THURSDAY";
20         break;
21     case 5:
22         day = "FRIDAY";
23         break;
24     case 6:
25         day = "SATURDAY";
26         break;
27     case 7:
28         day = "SUNDAY";
29         break;
30     default :
31         day = "NOT A VALID DAY" ;
32 }
33
34 print("DAY NUMBER ", dayNumber, flag, day);
35 }

```

#### OUTPUT

EWVM

```

1  // Global variable declarations
2  start  // Program start
3  pusha main  // Call main function
4  call
5  stop  // Program end
6
7  main:  // Function main declaration
8  pushn 3 // Reserve space for 3 local variables
9  pushi 0
10 storel 0
11 pushs ""
12 storel 1
13 pushs " IS "
14 storel 2
15 pushs "INSERT A NUMBER [1-7]: "

```

16	writes
17	writeln
18	read
19	atoi
20	storel 0
21	pushl 0
22	
23	dup 1
24	pushi 1
25	equal
26	jz case2
27	pushs "MONDAY"
28	storel 1
29	jump endswitch1
30	
31	case2:
32	dup 1
33	pushi 2
34	equal
35	jz case3
36	pushs "TUESDAY"
37	storel 1
38	jump endswitch1
39	
40	case3:
41	dup 1
42	pushi 3
43	equal
44	jz case4
45	pushs "WEDNESDAY"
46	storel 1
47	jump endswitch1
48	
49	case4:
50	dup 1
51	pushi 4
52	equal
53	jz case5
54	pushs "THURSDAY"
55	storel 1
56	jump endswitch1
57	
58	case5:
59	dup 1
60	pushi 5
61	equal



```

62  jz case6
63  pushs "FRIDAY"
64  storel 1
65  jump endswitch1
66
67  case6:
68  dup 1
69  pushi 6
70  equal
71  jz case7
72  pushs "SATURDAY"
73  storel 1
74  jump endswitch1
75
76  case7:
77  dup 1
78  pushi 7
79  equal
80  jz default8
81  pushs "SUNDAY"
82  storel 1
83  jump endswitch1
84
85  default8:
86  pushs "NOT A VALID DAY"
87  storel 1
88  endswitch1:
89  pop 1
90  pushs "DAY NUMBER "
91  writes
92  pushl 0
93  writei
94  pushl 2
95  writes
96  pushl 1
97  writes
98  writeln
99  return // Return from void function

```

#### INPUT FOR LOOPS

 Go

```

1  i := 0;
2  func main() void {
3
4      // for loop (init, cond, pos)
5      for var j int = 1; j <= 10; j++ {

```

```

6         if j % 2 == 0 {
7             print(j, " IS EVEN") ;
8         } else {
9             print(j, " IS ODD") ;
10        }
11    }
12
13    // for loop (cond) equal to while
14    var arr int[5] = {1, 2, 3, 4, 5};
15    i = 4;
16    for i >= 0 {
17        print(arr[i]);
18        i = i - 1;
19    }
20
21
22    // for loop (forever)
23    i = 0;
24    for {
25        if i == 5 {
26            return ;
27        }
28        print("INFINITE LOOP");
29        i = i + 1 ;
30    }
31
32 }

```

#### OUTPUT

EWVM

```

1    // i := 0 ;
2    pushi 0
3    storeg 0
4
5    start
6    pusha main
7    call
8    stop
9
10   main:
11   // for var j int = 1 ; j <= 10 ; j++
12   pushn 1
13   pushi 1
14   storel 0
15   forl:
16   pushl 0

```

```

17     pushi 10
18     infeq
19     jz endfor3
20
21     // if j % 2 == 0
22     pushl 0
23     pushi 2
24     mod
25     pushi 0
26     equal
27     jz else4
28
29     // print(j, " IS EVEN ") ;
30     pushl 0
31     writei
32     pushs " IS EVEN"
33     writes
34     writeln
35     jump endif5
36
37     // else
38     else4:
39
40     // print(j, "IS ODD") ;
41     pushl 0
42     writei
43     pushs " IS ODD"
44     writes
45     writeln
46     endif5:
47
48     continue2:
49     pushl 0
50     pushi 1
51     add
52     storel 0
53     jump for1
54     endfor3:
55
56     // var arr int[5] = {1, 2, 3, 4, 5} ;
57     alloc 5
58     dup 1
59     pushi 0
60     pushi 1
61     storen
62     dup 1

```

63	pushi 1
64	pushi 2
65	storen
66	dup 1
67	pushi 2
68	pushi 3
69	storen
70	dup 1
71	pushi 3
72	pushi 4
73	storen
74	dup 1
75	pushi 4
76	pushi 5
77	storen
78	
79	// i = 4 ;
80	storel 1
81	pushi 4
82	storeg 0
83	
84	// for i >= 0
85	for6:
86	pushg 0
87	pushi 0
88	supeq
89	jz endfor8
90	
91	// print(arr[i]) ;
92	pushl 1
93	pushg 0
94	loadn
95	writeln
96	writeln
97	
98	// i = i - 1 ;
99	pushg 0
100	pushi 1
101	sub
102	storeg 0
103	continue7:
104	jump for6
105	endfor8:
106	
107	// i = 0 ;
108	pushi 0

```

109 storeg 0
110
111 // for
112 for9:
113
114 // if i == 5
115 pushg 0
116 pushi 5
117 equal
118 jz else12
119
120 // return ;
121 return
122
123 jump endif13
124 else12:
125 endif13:
126
127 // print("INFINITE LOOP") ;
128 pushs "INFINITE LOOP"
129 writes
130 writeln
131
132 // i = i + 1 ;
133 pushg 0
134 pushi 1
135 add
136 storeg 0
137 continue10:
138 jump for9
139 endfor11:
140 return

```

#### INPUT IF-STATEMENTS



```

1 func main() void {
2
3     balance := 1500.00;
4     withdrawal := 500.00;
5     pinCode := 1234;
6     enteredPin := toInt(read());
7     isAccountActive := true;
8     minimumBalance := 100.00;
9
10    if enteredPin != pinCode {
11        print("INCORRECT PIN. ACCESS DENIED.");

```

```

12     } else if !isActive {
13         print("ACCOUNT IS INACTIVE. PLEASE CONTACT CUSTOMER SERVICE.");
14     } else if withdrawal > balance {
15         print("INSUFFICIENT FUNDS.");
16     } else if balance-withdrawal < minimumBalance {
17         print("TRANSACTION DENIED. YOUR ACCOUNT MUST MAINTAIN A MINIMUM BALANCE OF
18             ", minimumBalance, " EUROS");
19     } else {
20         balance = balance - withdrawal;
21         print("WITHDRAWAL SUCCESSFUL! YOUR NEW BALANCE IS ", balance, " EUROS");
22     }
23     age := 20 ;
24     if age >= 18 && age <= 60 {
25         print("ELIGIBLE FOR THE PROGRAM") ;
26     } else {
27         print("NOT ELIGIBLE") ;
28     }
29 }

```

#### OUTPUT

EWVM

```

1     start
2     pusha main
3     call
4     stop
5
6     main:
7         // balance := 1500.00 ;
8         pushn 7
9         pushf 1500.0
10        storel 0
11
12        // withdrawal := 500.00 ;
13        pushf 500.0
14        storel 1
15
16        // pincode := 1234 ;
17        pushi 1234
18        storel 2
19
20        // enteredPin := toInt(read()) ;
21        read
22        atoi
23        storel 3
24

```

```

25 // isAccountActive := true ;
26 pushl 1
27 storel 4
28
29 // minimumBalance := 100.00 ;
30 pushf 100.0
31 storel 5
32
33 // if enteredPin != pinCode
34 pushl 3
35 pushl 2
36 equal
37 not
38 jz else1
39
40 // print("INCORRECT PIN. ACCESS DENIED.")
41 pushes "INCORRECT PIN. ACCESS DENIED."
42 writes
43 writeln
44 jump endif2
45 else1:
46
47 // else if !isAccountActive
48 pushl 4
49 not
50 jz else3
51
52 // print("ACCOUNT IS INACTIVE. PLEASE CONTACT CUSTOMER SERVICE.") ;
53 pushes "ACCOUNT IS INACTIVE. PLEASE CONTACT CUSTOMER SERVICE."
54 writes
55 writeln
56 jump endif4
57 else3:
58
59 // else if withdrawal > balance
60 pushl 1
61 pushl 0
62 sup
63 jz else5
64
65 // print("INSUFFICIENT FUNDS") ;
66 pushes "INSUFFICIENT FUNDS."
67 writes
68 writeln
69 jump endif6
70 else5:

```

```

71
72 // else if balance-withdrawal < minumunBalance
73 pushl 0
74 pushl 1
75 sub
76 pushl 5
77 inf
78 jz else7
79
80 // print("TRANSACTION DENIED. YOUR ACCOUNT MUST MAINTAIN A MINIMUM BALANCE OF ",
    minimumBalance, " EUROS") ;
81 pushes "TRANSACTION DENIED. YOUR ACCOUNT MUST MAINTAIN A MINIMUM BALANCE OF "
82 writes
83 pushl 5
84 writei
85 pushes " EUROS"
86 writes
87 writeln
88 jump endif8
89
90 // else
91 else7:
92
93 // balance = balance - withdrawal ;
94 pushl 0
95 pushl 1
96 sub
97 storel 0
98
99 // print("WITHDRAWAL SUCCESSFUL! YOUR NEW BALANCE IS ", balance, " EUROS") ;
100 pushes "WITHDRAWAL SUCCESSFUL! YOUR NEW BALANCE IS "
101 writes
102 pushl 0
103 writei
104 pushes " EUROS"
105 writes
106 writeln
107 endif8:
108 endif6:
109 endif4:
110 endif2:
111
112 // age := 20 ;
113 pushi 20
114 storel 6
115

```



```

116 // if age >= 18 && age <= 60
117 pushl 6
118 pushl 18
119 supeq
120 pushl 6
121 pushl 60
122 infeq
123 and
124 jz else9
125
126 // print("ELIGIBLE FOR THE PROGRAM") ;
127 pushes "ELIGIBLE FOR THE PROGRAM"
128 writes
129 writeln
130 jump endif10
131
132 //else
133 else9:
134
135 // print("NOT ELIGIBLE") ;
136 pushes "NOT ELIGIBLE"
137 writes
138 writeln
139 endif10:
140 return

```

#### INPUT FUNCTIONS

 Go

```

1 func Abs(num int) int {
2     if num < 0 {
3         return -num;
4     }
5     return num;
6 }
7
8 func main() void {
9     number := -42;
10    result := Abs(number);
11    print("THE ABSOLUTE VALUE OF ", number, " IS ", result); }

```

#### OUTPUT

 EWVM

```

1
2 start
3 pusha main
4 call
5 stop

```

```

6
7  Abs:
8    // if num < 0
9    pushfp
10   load -1
11   pushi 0
12   inf
13   jz else1
14
15   // return - num ;
16   pushi 0
17   pushfp
18   load -1
19   sub
20   return
21   jump endif2
22
23   return num ;
24   else1:
25   endif2:
26   pushfp
27   load -1
28   return
29
30 main:
31   // number := -42 ;
32   pushn 2
33   pushi 0
34   pushi 42
35   sub
36   storel 0
37
38   // result := Abs(number) ;
39   pushl 0
40   pusha Abs
41   call
42   storel 1
43
44   // print("THE ABSOLUTE VALUE OF ", number, " IS ", result);
45   pushes "THE ABSOLUTE VALUE OF "
46   writes
47   pushl 0
48   writei
49   pushes " IS "
50   writes
51   pushl 1

```

```
52 writei
53 writeln
54 return
```

## 7. Conclusão

Este trabalho focou-se na criação de uma linguagem de programação bem como na implementação de um compilador correspondente. Para isso, desenvolvemos a Gramática Independente de Contexto (GIC), juntamente com um Analisador Léxico e um Analisador Sintático.

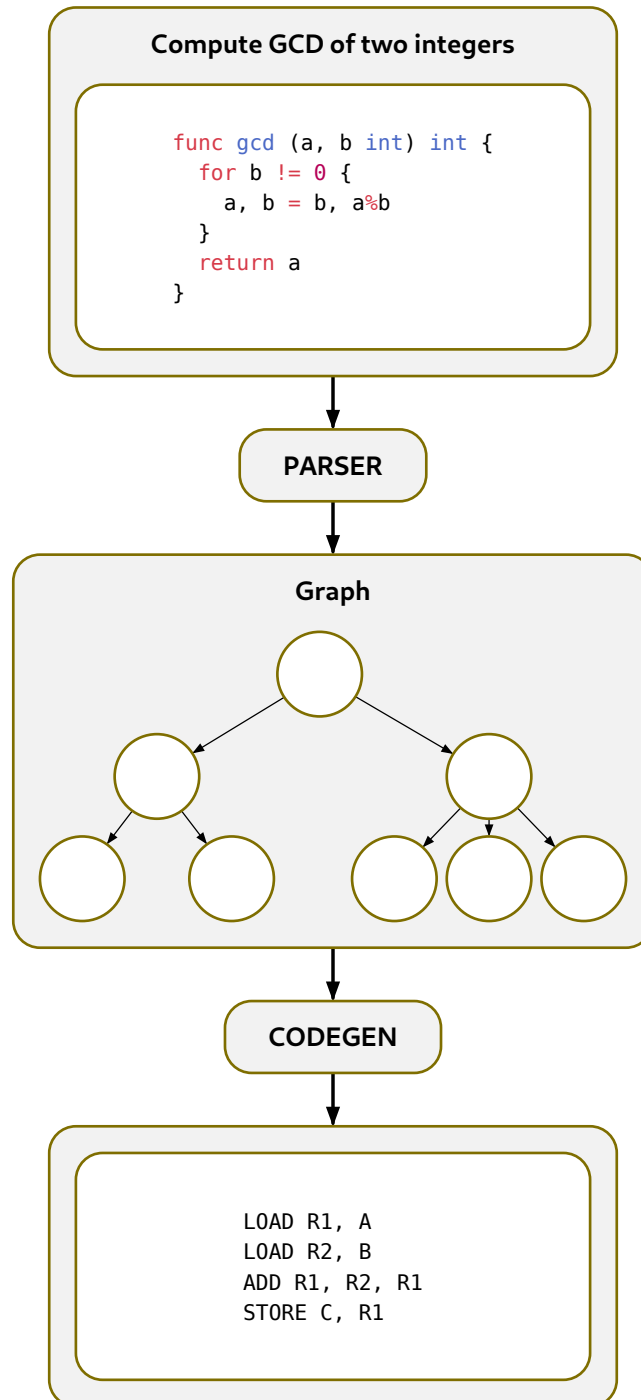
Neste relatório, detalhamos as decisões de design tomadas ao longo do projeto e explicamos as abordagens adotadas na sua implementação.

O resultado final é uma linguagem de programação robusta e elegante, concebida para promover boas práticas de desenvolvimento, com ênfase na Programação Estruturada.

Para o futuro, o grupo tem a intenção de permitir a implementação de arrays multidimensionais e dinâmicos, adicionar suporte para imports de outros ficheiros e melhorar as mensagens de erro, tornando-as mais descritivas e úteis.

## 8. Anexos

### 8.1. Compiladores 101



## 8.2. Lexer

LEXER		Python
1	<code>import ply.lex as lex</code>	
2	<code>from ply.lex import TOKEN</code>	
3		
4	<code># Lista de palavras reservadas (reserved words) usadas na linguagem</code>	
5	<code>reserved = [</code>	
6	<code>    # Palavras-chave</code>	
7	<code>    "BREAK",</code>	
8	<code>    "CASE",</code>	
9	<code>    "CONTINUE",</code>	
10	<code>    "DEFAULT",</code>	
11	<code>    "ELSE",</code>	
12	<code>    "FOR",</code>	
13	<code>    "FUNC",</code>	
14	<code>    "IF",</code>	
15	<code>    "RETURN",</code>	
16	<code>    "SWITCH",</code>	
17	<code>    "VAR",</code>	
18	<code>    # Tipos</code>	
19	<code>    "INT",</code>	
20	<code>    "FLOAT",</code>	
21	<code>    "STRING",</code>	
22	<code>    "BOOL",</code>	
23	<code>    "VOID",</code>	
24	<code>]</code>	
25		
26	<code># Lista de nomes de tokens que inclui as palavras reservadas e outros símbolos</code>	
27	<code>tokens = reserved + [</code>	
28	<code>    # Operadores</code>	
29	<code>    "PLUS",</code>	
30	<code>    "MINUS",</code>	
31	<code>    "TIMES",</code>	
32	<code>    "DIVIDE",</code>	
33	<code>    "MODULO",</code>	
34	<code>    "EQUALITY",</code>	
35	<code>    "NOTEQUAL",</code>	
36	<code>    "GREATER",</code>	
37	<code>    "LESS",</code>	
38	<code>    "GREATEROREQUAL",</code>	
39	<code>    "LESSOREQUAL",</code>	
40	<code>    "CONDITIONALAND",</code>	
41	<code>    "CONDITIONALOR",</code>	
42	<code>    "NOT",</code>	
43	<code>    "EQUAL",</code>	
44	<code>    "WALRUS",    # Operador de atribuição curta :=</code>	

```

45     "INCREMENT", # ++
46     "DECREMENT", # --
47     # Delimitadores
48     "LPAREN",
49     "RPAREN",
50     "LBRACE",
51     "RBRACE",
52     "LBRACKET", # [
53     "RBRACKET", # ]
54     "COMMA",
55     "SEMICOLON",
56     "COLON",
57     # Literais
58     "NUMBER_LITERAL",
59     "STRING_LITERAL",
60     "BOOLEAN_LITERAL",
61     # Identificador
62     "ID",
63 ]
64
65 # Expressões regulares para tokens simples
66 t_PLUS      = r"\+"
67 t_MINUS     = r"-"
68 t_TIMES     = r"\*"
69 t_DIVIDE    = r"/"
70 t_MODULO    = r"%"
71 t_EQUALITY  = r"=="
72 t_NOTEQUAL  = r"!="
73 t_GREATER   = r">"
74 t_LESS      = r"<"
75 t_GREATEROREQUAL = r">="
76 t_LESSEOREQUAL  = r"<="
77 t_CONDITIONALAND = r"&&"
78 t_CONDITIONALOR  = r"||"
79 t_NOT            = r"!"
80 t_EQUAL          = r"="
81 t_WALRUS         = r":="
82 t_INCREMENT      = r"\++"
83 t_DECREMENT      = r"--"
84 t_LPAREN          = r"("
85 t_RPAREN          = r")"
86 t_LBRACE          = r"{"
87 t_RBRACE          = r"}"
88 t_LBRACKET        = r"["
89 t_RBRACKET        = r"]"
90 t_COMMA           = r","

```

```

91 t_SEMICOLON      = r";"
92 t_COLON          = r":"
93
94 # Ignorar espaços e tabulações
95 t_ignore = " \t"
96
97 # Comentário de linha simples (//...), conta a nova linha para atualizar o número
  de linha
98 @TOKEN(r'//.*\n')
99 def t_COMMENT_SINGLE(t):
100     t.lexer.lineno += 1
101
102 # Comentário de múltiplas linhas (/* ... */), conta todas as quebras de linha
  internas
103 @TOKEN(r'/\*(\[^\*]\|\*[^\*])*\*/')
104 def t_COMMENT_MULTI(t):
105     t.lexer.lineno += t.value.count('\n')
106
107
108 # Definição de literais booleanos (True/False)
109 @TOKEN(r"[Tt]rue|[Ff]alse")
110 def t_BOOLEAN_LITERAL(t):
111     t.type = "BOOLEAN_LITERAL"
112     t.value = True if t.value.lower() == "true" else False
113     return t
114
115
116 # Definição de literais numéricos (inteiros ou floats)
117 @TOKEN(r"\d+(\.\d+)?")
118 def t_NUMBER_LITERAL(t):
119     if "." in t.value:
120         t.value = float(t.value)
121     else:
122         t.value = int(t.value)
123     return t
124
125
126 # Definição de literais de string (entre aspas duplas)
127 @TOKEN(r"\"([^\\"n]|(\\\.))*?\"")
128 def t_STRING_LITERAL(t):
129     # Remove as aspas iniciais e finais
130     t.value = t.value[1:-1]
131     return t
132
133
134 # Definição de identificadores (ID) e verificação se são palavras reservadas
135 @TOKEN(r"[a-zA-Z_][a-zA-Z_0-9]*")

```




```

136 def t_ID(t):
137     # Só aceitamos palavras reservadas se estiverem em letras minúsculas
138     # e constarem na lista de reserved (em maiúsculas).
139     t.type = t.value.upper() if (t.value.islower() and t.value.upper() in
        reserved) else "ID"
140     return t
141
142
143 # Controlo de quebras de linha
144 @TOKEN(r"\n+")
145 def t_newline(t):
146     t.lexer.lineno += len(t.value)
147
148
149 # Regra de erro: em caso de carácter ilegal, salta 1 posição e imprime mensagem
150 def t_error(t):
151     print(f"Illegal character '{t.value[0]}' at line {t.lineno}")
152     t.lexer.skip(1)
153
154
155 # Construção do lexer (analisador léxico)
156 lexer = lex.lex(debug=False)
157
158 # Teste do lexer com um exemplo
159 if __name__ == "__main__":
160     test_data = """
161         func main() int {
162             var arr int[10];
163             for i := 0; i < 10; i++ {
164                 if arr[i] == 5 {
165                     break;
166                 }
167                 continue;
168             }
169             return 0;
170         }
171     """
172     lexer.input(test_data)
173     for token in lexer:
174         print(f"{token.type}, '{token.value}', {token.lineno}")

```

### 8.3. Parser

PARSER

 Python

```

1 import ply.yacc as yacc
2 from cparser.lexer import tokens

```

```

3
4  from c_ast.ast_nodes import (
5      Program,
6      FunctionDeclaration,
7      VariableDeclaration,
8      ArrayType,
9      ArrayInitializer,
10     ContinueStatement,
11     BreakStatement,
12     ExpressionStatement,
13     IfStatement,
14     ForStatement,
15     SwitchStatement,
16     SwitchCase,
17     ReturnStatement,
18     BinaryOp,
19     UnaryOp,
20     Variable,
21     FunctionCall,
22     ArrayAccess,
23     Assignment,
24     ArrayAssignment,
25     Literal,
26 )
27
28 # ----- Precedência de operadores e outras configurações do parser -----
29 # A tupla 'precedence' informa ao PLY como desempatar conflitos de parsing,
30 # definindo a precedência e associatividade (left, right) de cada token.
31 precedence = (
32     ("left", "CONDITIONALOR"),
33     ("left", "CONDITIONALAND"),
34     ("left", "EQUALITY", "NOTEQUAL"),
35     ("left", "LESS", "LESSOREQUAL", "GREATER", "GREATEROREQUAL"),
36     ("left", "PLUS", "MINUS"),
37     ("left", "TIMES", "DIVIDE", "MODULO"),
38     ("right", "NOT"),
39     ("right", "UMINUS"),
40 )
41
42 # ----- Regras principais de análise sintática (gramática) -----
43
44 # Programa completo: Conjunto de declarações globais
45 def p_program(p):
46     "program : global_declarations"
47     # Cria um nó de AST Program contendo as declarações e o número de linha
48     p[0] = Program(declarations=p[1], lineno=p.lineno(1))

```

```

49
50 # Múltiplas declarações globais (concatenadas em lista)
51 def p_global_declarations_multiple(p):
52     "global_declarations : global_declarations global_declaration"
53     p[0] = p[1] + [p[2]]
54
55 # Única declaração global
56 def p_global_declarations_single(p):
57     "global_declarations : global_declaration"
58     p[0] = [p[1]]
59
60 # Declaração global de função
61 def p_global_declaration_func(p):
62     "global_declaration : function_declaration"
63     p[0] = p[1]
64
65 # Declaração global de variável (normal ou curta)
66 def p_global_declaration_var(p):
67     """global_declaration : var_declaration
68                               | short_var_declaration"""
69     p[0] = p[1]
70
71 # ----- Declarações de Função -----
72
73 def p_function_declaration(p):
74     """
75     function_declaration : FUNC ID LPAREN parameters_opt RPAREN type block
76     """
77     # Cria um nó FunctionDeclaration com o nome, parâmetros, tipo de retorno,
78     # corpo (bloco) e a linha correspondente.
79     p[0] = FunctionDeclaration(
80         name=p[2],
81         params=p[4],
82         return_type=p[6],
83         body=p[7],
84         lineno=p.lineno(1),
85     )
86
87 # Parâmetros opcionais (caso vazio)
88 def p_parameters_opt(p):
89     "parameters_opt : parameters"
90     p[0] = p[1]
91
92 # Se não houver parâmetros, retorna lista vazia
93 def p_parameters_empty(p):
94     "parameters : empty"

```

```

95     p[0] = []
96
97     # Múltiplos parâmetros (separados por vírgulas)
98     def p_parameters_multiple(p):
99         "parameters : parameters COMMA parameter"
100         p[0] = p[1] + [p[3]]
101
102     # Um único parâmetro
103     def p_parameters_single(p):
104         "parameters : parameter"
105         p[0] = [p[1]]
106
107     # Definição de um parâmetro (nome + tipo)
108     def p_parameter(p):
109         "parameter : ID type"
110         p[0] = {"name": p[1], "type": p[2]}
111
112     # Bloco de código: { ... }
113     def p_block(p):
114         "block : LBRACE block_contents RBRACE"
115         p[0] = p[2]
116
117     # Bloco vazio: { }
118     def p_block_empty(p):
119         "block : LBRACE RBRACE"
120         p[0] = []
121
122     def p_block_contents(p):
123         "block_contents : statements"
124         p[0] = p[1]
125
126     # ----- Declarações / Instruções -----
127
128     # Múltiplas instruções
129     def p_statements_multiple(p):
130         "statements : statements statement"
131         p[0] = p[1] + [p[2]]
132
133     # Única instrução
134     def p_statements_single(p):
135         "statements : statement"
136         p[0] = [p[1]]
137
138     # Tipos de instruções possíveis
139     def p_statement(p):
140         """statement : var_declaration

```

```

141 | short_var_declaration
142 | expression_statement
143 | if_statement
144 | for_statement
145 | switch_statement
146 | return_statement
147 | assignment"""
148     p[0] = p[1]
149
150 # Instrução de continue
151 def p_statement_continue(p):
152     "statement : CONTINUE SEMICOLON"
153     p[0] = ContinueStatement(lineno=p.lineno(1))
154
155 # Instrução de break
156 def p_statement_break(p):
157     "statement : BREAK SEMICOLON"
158     p[0] = BreakStatement(lineno=p.lineno(1))
159
160 # Instrução de expressão (terminada com ponto-e-vírgula)
161 def p_expression_statement(p):
162     "expression_statement : expression SEMICOLON"
163     p[0] = ExpressionStatement(expression=p[1], lineno=p.lineno(2))
164
165 # ----- If Statements -----
166
167 def p_if_statement(p):
168     "if_statement : IF expression block else_clause"
169     p[0] = IfStatement(condition=p[2], then_branch=p[3], else_branch=p[4],
170                       lineno=p.lineno(1))
171
172 def p_else_clause_if(p):
173     "else_clause : ELSE if_statement"
174     p[0] = p[2]
175
176 def p_else_clause_block(p):
177     "else_clause : ELSE block"
178     p[0] = p[2]
179
180 def p_else_clause_empty(p):
181     "else_clause : empty"
182     p[0] = None
183
184 # ----- For Statements -----
185 def p_for_statement(p):

```

```

186     """
187     for_statement : FOR for_init SEMICOLON expression SEMICOLON for_post block
188     """
189     # for ( init ; condition ; post ) { block }
190     p[0] = ForStatement(init=p[2], condition=p[4], post=p[6], body=p[7],
191         lineno=p.lineno(1))
192
193     # for ( expression ) { block } => estilo "for" simplificado
194     def p_for_statement_expression(p):
195         "for_statement : FOR expression block"
196         p[0] = ForStatement(init=None, condition=p[2], post=None, body=p[3],
197             lineno=p.lineno(1))
198
199     # for ( block ) => estilo "for sem condição" (equivalente a loop infinito)
200     def p_for_while_statement(p):
201         "for_statement : FOR block"
202         p[0] = ForStatement(init=None, condition=None, post=None, body=p[2],
203             lineno=p.lineno(1))
204
205     # Inicialização curta dentro do "for" (i := 0)
206     def p_for_init_short_var(p):
207         "for_init : ID WALRUS expression"
208         p[0] = VariableDeclaration(name=p[1], var_type=None, initializer=p[3],
209             lineno=p.lineno(2))
210
211     # Declaração de variável no "for" (var i int = 0)
212     def p_for_init_var_declaration(p):
213         "for_init : VAR ID type EQUAL expression"
214         p[0] = VariableDeclaration(name=p[2], var_type=p[3], initializer=p[5],
215             lineno=p.lineno(1))
216
217     # Post (ou step) no "for" pode ser assignment
218     def p_for_post_assignment(p):
219         "for_post : assignment"
220         p[0] = p[1]
221
222     # Ou pode ser uma expressão
223     def p_for_post_expression(p):
224         "for_post : expression"
225         p[0] = p[1]
226
227     # ----- Switch Statements -----
228     def p_switch_statement(p):
229         """
230         switch_statement : SWITCH expression LBRACE switch_cases default_clause RBRACE
231         """

```

```

228     p[0] = SwitchStatement(expression=p[2], cases=p[4], default=p[5],
                             lineno=p.lineno(1))
229
230 def p_switch_cases_multiple(p):
231     "switch_cases : switch_cases switch_case"
232     p[0] = p[1] + [p[2]]
233
234 def p_switch_cases_single(p):
235     "switch_cases : switch_case"
236     p[0] = [p[1]]
237
238 def p_switch_case(p):
239     "switch_case : CASE expression COLON statements"
240     p[0] = SwitchCase(value=p[2], body=p[4], lineno=p.lineno(1))
241
242 def p_default_clause_with_statements(p):
243     "default_clause : DEFAULT COLON statements"
244     p[0] = p[3]
245
246 def p_default_clause_empty(p):
247     "default_clause : empty"
248     p[0] = None
249
250 # ----- Instrução de retorno -----
251
252 def p_return_statement(p):
253     "return_statement : RETURN expression_opt SEMICOLON"
254     p[0] = ReturnStatement(value=p[2], lineno=p.lineno(1))
255
256 def p_expression_opt(p):
257     """expression_opt : expression
258                       | empty"""
259     p[0] = p[1]
260
261 # ----- Declarações de Variáveis -----
262
263 def p_var_declaration_array_init(p):
264     "var_declaration : VAR ID array_type EQUAL array_initializer SEMICOLON"
265     p[0] = VariableDeclaration(name=p[2], var_type=p[3], initializer=p[5],
                                lineno=p.lineno(1))
266
267 def p_var_declaration_array_noinit(p):
268     "var_declaration : VAR ID array_type SEMICOLON"
269     p[0] = VariableDeclaration(name=p[2], var_type=p[3], initializer=None,
                                lineno=p.lineno(1))
270
271 def p_var_declaration_init(p):

```

```

272     "var_declaration : VAR ID type EQUAL expression SEMICOLON"
273     p[0] = VariableDeclaration(name=p[2], var_type=p[3], initializer=p[5],
274                                lineno=p.lineno(1))
275
276 def p_var_declaration_noinit(p):
277     "var_declaration : VAR ID type SEMICOLON"
278     p[0] = VariableDeclaration(name=p[2], var_type=p[3], initializer=None,
279                                lineno=p.lineno(1))
280
281 def p_short_var_declaration(p):
282     "short_var_declaration : ID WALRUS expression SEMICOLON"
283     p[0] = VariableDeclaration(name=p[1], var_type=None, initializer=p[3],
284                                lineno=p.lineno(2))
285
286 # ----- Tipos de Array -----
287
288 def p_array_type_first_dimension(p):
289     "array_type : type LBRACKET NUMBER_LITERAL RBRACKET"
290     # Exemplo: int[10]
291     p[0] = ArrayType(base_type=p[1], dimensions=[p[3]], lineno=p.lineno(1))
292
293 def p_array_type_more_dimensions(p):
294     "array_type : array_type LBRACKET NUMBER_LITERAL RBRACKET"
295     # Exemplo: int[10][5]
296     p[1].dimensions.append(p[3])
297     p[0] = p[1]
298
299 def p_array_type_first_empty(p):
300     "array_type : type LBRACKET RBRACKET"
301     # Exemplo: int[]
302     # None usado como dimensão desconhecida
303     p[0] = ArrayType(base_type=p[1], dimensions=[None], lineno=p.lineno(1))
304
305 def p_array_type_more_empty(p):
306     "array_type : array_type LBRACKET RBRACKET"
307     p[1].dimensions.append(None)
308     p[0] = p[1]
309
310 # ----- Inicializadores de Array -----
311
312 def p_array_initializer_empty(p):
313     "array_initializer : LBRACE RBRACE"
314     # Exemplo: {}
315     p[0] = ArrayInitializer(elements=[], dimensions=[0], lineno=p.lineno(1))
316
317 def p_array_initializer_flat(p):
318     "array_initializer : LBRACE expression_list RBRACE"

```



```

316     # Exemplo: {1, 2, 3}
317     p[0] = ArrayInitializer(elements=p[2], dimensions=[len(p[2])],
318                             lineno=p.lineno(1))
318
319 def p_array_initializer_nested(p):
320     "array_initializer : LBRACE nested_initializer_list RBRACE"
321     # Exemplo: {{1,2},{3,4}}
322     elements = p[2]
323     if not elements:
324         # Caso de {} aninhado
325         p[0] = ArrayInitializer(elements=[], dimensions=[0], lineno=p.lineno(1))
326         return
327     dims = [len(elements)]
328     if isinstance(elements[0], ArrayInitializer):
329         # Se os elementos forem também ArrayInitializer, concatenar dimensões
330         dims.extend(elements[0].dimensions)
331     p[0] = ArrayInitializer(elements=elements, dimensions=dims,
332                             lineno=p.lineno(1))
332
333 def p_nested_initializer_list_single(p):
334     "nested_initializer_list : array_initializer"
335     p[0] = [p[1]]
336
337 def p_nested_initializer_list_multiple(p):
338     "nested_initializer_list : nested_initializer_list COMMA array_initializer"
339     p[0] = p[1] + [p[3]]
340
341 # ----- Acesso a Array -----
342
343 def p_array_access_first(p):
344     "array_access : ID LBRACKET expression RBRACKET"
345     # Exemplo: arr[2]
346     p[0] = ArrayAccess(array=Variable(name=p[1], lineno=p.lineno(1)),
347                         indices=[p[3]], lineno=p.lineno(1))
347
348 def p_array_access_next(p):
349     "array_access : array_access LBRACKET expression RBRACKET"
350     # Exemplo: arr[2][3]
351     p[1].indices.append(p[3])
352     p[0] = p[1]
353
354 # ----- Atribuições -----
355
356 def p_simple_assignment(p):
357     "assignment : ID EQUAL expression SEMICOLON"
358     # Exemplo: x = 10;

```

```

359     p[0] = Assignment(target=Variable(name=p[1], lineno=p.lineno(1)), value=p[3],
360                       lineno=p.lineno(2))
361
362 def p_array_element_assignment(p):
363     "assignment : array_access EQUAL expression SEMICOLON"
364     # Exemplo: arr[2] = 5;
365     p[0] = ArrayAssignment(array=p[1].array, indices=p[1].indices, value=p[3],
366                           lineno=p.lineno(2))
367
368 def p_array_assignment(p):
369     "assignment : ID EQUAL array_initializer SEMICOLON"
370     # Exemplo: arr = {1,2,3};
371     p[0] = Assignment(target=Variable(name=p[1], lineno=p.lineno(1)), value=p[3],
372                       lineno=p.lineno(2))
373
374 # ----- Expressões -----
375
376 def p_expression_binop(p):
377     """expression : expression PLUS expression
378                   | expression MINUS expression
379                   | expression TIMES expression
380                   | expression DIVIDE expression
381                   | expression MODULO expression
382                   | expression EQUALITY expression
383                   | expression NOTEQUAL expression
384                   | expression GREATER expression
385                   | expression LESS expression
386                   | expression GREATEROREQUAL expression
387                   | expression LESSOREQUAL expression
388                   | expression CONDITIONALAND expression
389                   | expression CONDITIONALOR expression"""
390     # Operações binárias (Ex.: +, -, *, /, ==, !=, etc.)
391     p[0] = BinaryOp(operator=p[2], left=p[1], right=p[3], lineno=p.lineno(2))
392
393 def p_expression_unaryop(p):
394     """expression : NOT expression
395                   | MINUS expression %prec UMINUS"""
396     # Operadores unários (Ex.: !expr, -expr)
397     p[0] = UnaryOp(operator=p[1], operand=p[2], lineno=p.lineno(1))
398
399 def p_expression_group(p):
400     "expression : LPAREN expression RPAREN"
401     # Expressão entre parênteses
402     p[0] = p[2]
403
404 def p_expression_literal(p):
405     "expression : literal"

```

```

403     # Literal (número, string, bool)
404     p[0] = p[1]
405
406 def p_expression_id(p):
407     "expression : ID"
408     # Variável
409     p[0] = Variable(name=p[1], lineno=p.lineno(1))
410
411 def p_expression_array_access(p):
412     "expression : array_access"
413     # Acesso a array como expressão
414     p[0] = p[1]
415
416 def p_expression_increment(p):
417     "expression : ID INCREMENT"
418     # i++
419     p[0] = UnaryOp(operator="++", operand=Variable(name=p[1], lineno=p.lineno(1)),
420                  lineno=p.lineno(2))
421
422 def p_expression_decrement(p):
423     "expression : ID DECREMENT"
424     # i--
425     p[0] = UnaryOp(operator="--", operand=Variable(name=p[1], lineno=p.lineno(1)),
426                  lineno=p.lineno(2))
427
428 def p_expression_function_call(p):
429     "expression : function_call"
430     # Chamada de função como expressão
431     p[0] = p[1]
432
433 def p_function_call(p):
434     "function_call : ID LPAREN arguments_opt RPAREN"
435     p[0] = FunctionCall(name=p[1], arguments=p[3], lineno=p.lineno(1))
436
437 def p_arguments_opt(p):
438     "arguments_opt : arguments"
439     p[0] = p[1]
440
441 def p_arguments_empty(p):
442     "arguments : empty"
443     p[0] = []
444
445 def p_arguments_multiple(p):
446     "arguments : arguments COMMA expression"
447     p[0] = p[1] + [p[3]]
448
449 def p_arguments_single(p):

```

```

448     "arguments : expression"
449     p[0] = [p[1]]
450
451 def p_expression_list_multiple(p):
452     "expression_list : expression_list COMMA expression"
453     p[0] = p[1] + [p[3]]
454
455 def p_expression_list_single(p):
456     "expression_list : expression"
457     p[0] = [p[1]]
458
459 # ----- Literais -----
460
461 def p_literal_number(p):
462     """literal : NUMBER_LITERAL"""
463     # Determina se é int ou float
464     if isinstance(p[1], int):
465         p[0] = Literal(value=p[1], type="int", lineno=p.lineno(1))
466     else:
467         p[0] = Literal(value=p[1], type="float", lineno=p.lineno(1))
468
469 def p_literal_string(p):
470     """literal : STRING_LITERAL"""
471     p[0] = Literal(value=p[1], type="string", lineno=p.lineno(1))
472
473 def p_literal_boolean(p):
474     """literal : BOOLEAN_LITERAL"""
475     p[0] = Literal(value=p[1], type="bool", lineno=p.lineno(1))
476
477 # ----- Tipos Básicos -----
478 def p_basic_type(p):
479     """type : INT
480             | FLOAT
481             | BOOL
482             | STRING
483             | VOID"""
484     # Tipo base (int, float, bool, string, void)
485     p[0] = p[1]
486
487 # Produção vazia (usada para representar opcionalidade)
488 def p_empty(p):
489     "empty :"
490     p[0] = None
491
492 # Função de tratamento de erros de parsing
493 def p_error(p):

```

```

494     """
495     Função simples de tratamento de erro do parser.
496     """
497     if not p:
498         print("Syntax Error: Unexpected end of input")
499     else:
500         print(f"Syntax Error at line {p.lineno}: Unexpected token {p.value}")
501     parser.errok()
502
503 # Construção do parser
504 parser = yacc.yacc()

```

## 8.4. Semantic Analyzer

SEMANTIC ANALYZER		Python
1	<code>from typing import List, Optional, Any</code>	
2	<code>from semantic.symbol_table import SymbolTable</code>	
3	<code>from c_ast.ast_nodes import (</code>	
4	<code>ArrayAccess,</code>	
5	<code>ArrayAssignment,</code>	
6	<code>ArrayType,</code>	
7	<code>Symbol,</code>	
8	<code>Variable,</code>	
9	<code>ArrayInitializer,</code>	
10	<code>Literal,</code>	
11	<code>Program,</code>	
12	<code>FunctionDeclaration,</code>	
13	<code>VariableDeclaration,</code>	
14	<code>Assignment,</code>	
15	<code>IfStatement,</code>	
16	<code>ForStatement,</code>	
17	<code>SwitchStatement,</code>	
18	<code>ReturnStatement,</code>	
19	<code>BreakStatement,</code>	
20	<code>ContinueStatement,</code>	
21	<code>ExpressionStatement,</code>	
22	<code>BinaryOp,</code>	
23	<code>UnaryOp,</code>	
24	<code>FunctionCall,</code>	
25	<code>)</code>	
26		
27	<code>class SemanticError(Exception):</code>	
28	<code>"""</code>	
29	Exceção para erros semânticos.	
30	Permite associar uma mensagem e, opcionalmente, o nó da AST onde ocorreu o erro.	

```

31     """
32
33     def __init__(self, message, node=None):
34         self.node = node
35         message = f"SemanticError: {message}"
36         # Se o nó tiver número de linha (lineno), acrescentar essa informação à
           mensagem
37         if node is not None and hasattr(node, "lineno") and node.lineno is not
           None:
38             message += f" at line {node.lineno}"
39         super().__init__(message)
40
41
42     class SemanticAnalyzer:
43         """
44         Analisador semântico que verifica:
45         - Compatibilidade de tipos (e anota 'inferred_type' nos nós),
46         - Definição e uso correto de símbolos (variáveis, funções),
47         - Uso de instruções return compatíveis com o tipo de retorno da função,
48         - Uso adequado de break/continue dentro de estruturas de repetição (loops).
49         """
50
51         def __init__(self):
52             # Tabela de símbolos (SymbolTable) que mantém registo de variáveis e
               funções
53             self.symbol_table = SymbolTable()
54
55             # Nome da função atualmente em análise (ou None se estiver fora de
               qualquer função)
56             self.current_function: Optional[str] = None
57
58             # Indica se estamos dentro de um loop (para gerir break/continue)
59             self.in_loop = False
60
61             # Conjunto de tipos básicos aceites
62             self.basic_types = {"int", "float", "bool", "string", "void"}
63
64         def analyze(self, ast) -> Any:
65             """
66             Executa a análise semântica sobre a AST fornecida.
67             1. Adiciona as funções built-in à tabela de símbolos.
68             2. Percorre a AST com a função visit.
69             3. Devolve a AST anotada (p.ex., com tipos inferidos).
70             """
71             if ast is None:
72                 return
73

```

```

74         # Insere funções built-in na tabela de símbolos
75         self.add_builtins()
76
77         # Visita recursivamente a AST
78         self.visit(ast)
79         return ast
80
81     def add_builtins(self):
82         """
83         Adiciona funções built-in (print, read, toInt, toFloat, toStr, len) como
84         símbolos na tabela,
85         cada uma marcada como is_function=True.
86         """
87         builtin_functions = [
88             Symbol(name="print", type="void", is_function=True, params=[],
89                   return_type="void"),
90             Symbol(name="read", type="string", is_function=True, params=[],
91                   return_type="string"),
92             Symbol(name="toInt", type="int", is_function=True,
93                   params=[{"name": "value", "type": "any"}], return_type="int"),
94             Symbol(name="toFloat", type="float", is_function=True,
95                   params=[{"name": "value", "type": "any"}], return_type="float"),
96             Symbol(name="toStr", type="string", is_function=True,
97                   params=[{"name": "value", "type": "any"}], return_type="string"),
98             Symbol(name="len", type="int", is_function=True,
99                   params=[{"name": "value", "type": "any"}], return_type="int"),
100         ]
101
102         for func in builtin_functions:
103             self.symbol_table.define(func.name, func)
104
105     def visit(self, node) -> Optional[str]:
106         """
107         Método genérico de despacho de visita.
108         Dado um nó da AST, encontra o método 'visit_<nome da classe do nó>' e
109         invoca-o.
110         """
111         method_name = f"visit_{node.__class__.__name__}"
112         visitor = getattr(self, method_name, self.generic_visit)
113         return visitor(node)
114
115     def generic_visit(self, node):
116         """
117         Se não houver um método 'visit_X' específico para o tipo de nó, geramos um
118         erro.
119         """
120         raise SemanticError(f"No visit method for node type:
121                             {type(node).__name__}")

```

```

112
113     # -----
114     # MÉTODOS AUXILIARES
115     # -----
116
117     def _visit_block(self, statements):
118         """
119         Auxiliar para visitar um bloco de instruções.
120         O bloco pode ser uma lista de instruções ou uma única instrução.
121         """
122         if isinstance(statements, list):
123             for stmt in statements:
124                 self.visit(stmt)
125         else:
126             self.visit(statements)
127
128     def type_compatible(self, source_type: Any, target_type: Any) -> bool:
129         """
130         Verifica se 'source_type' pode ser atribuído a 'target_type'.
131         Trata:
132         - Tipos de array,
133         - Tipos básicos,
134         - Parâmetro 'any' usado em funções built-in.
135         """
136         # Se o source_type for uma função (Symbol) obtida pelo nome, usar o seu
137         # return_type
138         func_symbol = self.symbol_table.lookup(str(source_type))
139         if func_symbol and func_symbol.is_function:
140             source_type = func_symbol.return_type
141
142         # Caso de arrays
143         if isinstance(target_type, ArrayType):
144             # É necessário que ambos sejam ArrayType com dimensões compatíveis e
145             # mesmo tipo base
146             if not isinstance(source_type, ArrayType):
147                 return False
148             if len(source_type.dimensions) != len(target_type.dimensions):
149                 return False
150             # Verificar cada dimensão (quando não for None)
151             for src_dim, tgt_dim in zip(source_type.dimensions,
152                                         target_type.dimensions):
153                 if tgt_dim is not None and src_dim > tgt_dim:
154                     return False
155             return self.type_compatible(source_type.base_type,
156                                         target_type.base_type)

```



```

154         # 'any' é usado em alguns parâmetros de funções built-in (sem restrição de
        tipo)
155         if target_type == "any":
156             return True
157
158         # Caso geral: verificar igualdade para tipos básicos
159         return source_type == target_type
160
161     # -----
162     # MÉTODOS DE VISITA
163     # -----
164
165     def visit_Program(self, node: Program) -> None:
166         """
167         Visita cada declaração no nó Program (funções, variáveis globais, etc.).
168         """
169         for decl in node.declarations:
170             self.visit(decl)
171
172     def visit_FunctionDeclaration(self, node: FunctionDeclaration):
173         """
174         Declaração de função:
175         1. Verifica se o tipo de retorno é válido.
176         2. Cria um símbolo da função e regista-o na tabela de símbolos.
177         3. Visita o corpo da função num novo scope (stack de scopes).
178         4. Verifica se funções não-void têm pelo menos um return de nível
            superior.
179         """
180         # Verificar se o tipo de retorno é válido
181         if node.return_type not in self.basic_types:
182             raise SemanticError(
183                 f"Invalid return type '{node.return_type}'",
184                 node
185             )
186
187         # Definir um símbolo para a função
188         func_symbol = Symbol(
189             name=node.name,
190             type=node.return_type,
191             is_function=True,
192             params=node.params,
193             return_type=node.return_type,
194         )
195         self.symbol_table.define(node.name, func_symbol)
196
197         # Variável para verificar se encontramos um 'return' de nível superior
198         found_return = False

```

```

199
200     # Entrar num novo scope para a função
201     with self.symbol_table.new_scope():
202         self.current_function = node.name
203
204     # Definir os parâmetros da função no scope
205     for param in node.params:
206         self.symbol_table.define(param["name"], Symbol(name=param["name"],
207                                                         type=param["type"]))
208
209     # Se o corpo for lista de instruções, visitar cada uma
210     if isinstance(node.body, list):
211         for stmt in node.body:
212             self.visit(stmt)
213             # Verificar se esta instrução é um ReturnStatement
214             if isinstance(stmt, ReturnStatement):
215                 found_return = True
216             else:
217                 # Corpo único
218                 self.visit(node.body)
219                 if isinstance(node.body, ReturnStatement):
220                     found_return = True
221
222     self.current_function = None
223
224     # Se a função não for 'void' e não tiver um return de topo, lança erro
225     if node.return_type != "void" and not found_return:
226         raise SemanticError(
227             f"Function '{node.name}' must have a top-level return statement",
228             node
229         )
230
231     def visit_VariableDeclaration(self, node: VariableDeclaration) -> None:
232         """
233         Declaração de variável (simples ou array).
234         1. Verifica dimensões dos arrays (se definido).
235         2. Verifica/inferir o tipo da variável a partir do initializer (se
236            existir).
237         3. Garante que o tipo da inicialização é compatível com var_type.
238         4. Define o símbolo na tabela.
239         """
240         # Se for um ArrayType, verificar se dimensões especificadas são > 0
241         if isinstance(node.var_type, ArrayType):
242             for i, dim in enumerate(node.var_type.dimensions):
243                 if dim is not None and dim <= 0:
244                     raise SemanticError(f"Array dimension {i} must be positive",
245                                         node)

```

```

243
244     # Visitar o initializer (se existir) para descobrir ou confirmar o tipo
245     init_type = None
246     if node.initializer:
247         init_type = self.visit(node.initializer)
248
249     # Se var_type for explicitamente fornecido
250     if node.var_type:
251         if init_type:
252             if not self.type_compatible(init_type, node.var_type):
253                 raise SemanticError(
254                     f"Type mismatch in variable '{node.name}' declaration: "
255                     f"expected {node.var_type}, got {init_type}",
256                     node
257                 )
258             # Se o array type tiver dimensões None, preencher com as dimensões
259             # obtidas do init_type
260             if isinstance(node.var_type, ArrayType) and isinstance(init_type,
261                             ArrayType):
262                 node.var_type.dimensions = [
263                     init_type.dimensions[i] if d is None else d
264                     for i, d in enumerate(node.var_type.dimensions)
265                 ]
266             # Se ainda houver discrepâncias, lançar erro
267             if node.var_type.dimensions != init_type.dimensions:
268                 raise SemanticError(
269                     f"Array initializer dimensions do not match array
270                     type: "
271                     f"expected {node.var_type.dimensions}, got
272                     {init_type.dimensions}",
273                     node
274                 )
275         else:
276             # Se var_type não for fornecido, inferir do initializer
277             node.var_type = init_type
278
279     # Registrar a variável na tabela de símbolos
280     var_symbol = Symbol(name=node.name, type=node.var_type)
281     self.symbol_table.define(node.name, var_symbol)
282
283     # Guardar o tipo final em node.inferred_type (opcional)
284     node.inferred_type = node.var_type
285
286     def visit_Assignment(self, node: Assignment) -> None:
287         """
288         Atribuição simples (target = value).
289         1. Visita a variável destino e o valor.

```

```

286         2. Verifica compatibilidade de tipos.
287         3. Armazena o tipo resultante em node.inferred_type (normalmente igual ao
288         tipo do destino).
289         """
290         target_type = self.visit(node.target)
291         value_type = self.visit(node.value)
292
293         if not self.type_compatible(value_type, target_type):
294             raise SemanticError(
295                 f"Type mismatch in assignment: cannot assign {value_type} to
296                 {target_type}",
297                 node
298             )
299
300         node.inferred_type = target_type
301
302     def visit_ArrayAssignment(self, node: ArrayAssignment) -> None:
303         """
304         Atribuição a elemento(s) de array: array[i][j] = valor.
305         1. Verifica se 'array' é mesmo do tipo ArrayType.
306         2. Verifica se número de índices coincide com o número de dimensões.
307         3. Verifica se todos os índices são int.
308         4. Verifica se o tipo do valor é compatível com o base_type do array.
309         """
310         array_type = self.visit(node.array)
311         if not isinstance(array_type, ArrayType):
312             raise SemanticError(f"Cannot index into non-array type {array_type}",
313                                 node)
314
315         if len(node.indices) != len(array_type.dimensions):
316             raise SemanticError(
317                 f"Wrong number of dimensions in array assignment: expected
318                 {len(array_type.dimensions)}, got {len(node.indices)}", node
319             )
320
321         for i, index_expr in enumerate(node.indices):
322             idx_type = self.visit(index_expr)
323             if idx_type != "int":
324                 raise SemanticError(f"Array index {i} must be int, got
325                                     {idx_type}", node)
326
327         value_type = self.visit(node.value)
328         if not self.type_compatible(value_type, array_type.base_type):
329             raise SemanticError(
330                 f"Cannot assign value of type {value_type} to array element of
331                 type {array_type.base_type}", node
332             )

```

```

327
328     node.inferred_type = None # Normalmente, array assignment não produz um
    tipo de expressão
329
330     def visit_IfStatement(self, node: IfStatement) -> None:
331         """
332         Instrução if:
333         - A condição tem de ser do tipo bool.
334         - O corpo do then e do else é visitado (possivelmente em novos scopes).
335         """
336         cond_type = self.visit(node.condition)
337         if cond_type != "bool":
338             raise SemanticError(f"If condition must be boolean, got {cond_type}",
                                node)
339
340         # then_branch em novo scope
341         with self.symbol_table.new_scope():
342             self._visit_block(node.then_branch)
343
344         # else_branch (se existir) em outro scope
345         if node.else_branch is not None:
346             with self.symbol_table.new_scope():
347                 self._visit_block(node.else_branch)
348
349     def visit_ForStatement(self, node: ForStatement) -> None:
350         """
351         Instrução for:
352         - init corre num scope (variável local se houver).
353         - condition deve ser bool se existir.
354         - body corre num contexto de loop (in_loop=True).
355         - post (se houver) é visitado após o body.
356         """
357         with self.symbol_table.new_scope():
358             if node.init:
359                 self.visit(node.init)
360
361             if node.condition:
362                 cond_type = self.visit(node.condition)
363                 if cond_type != "bool":
364                     raise SemanticError(f"For condition must be boolean, got
                                         {cond_type}", node)
365
366             # Temporariamente marcamos in_loop = True para permitir break/continue
367             prev_in_loop = self.in_loop
368             self.in_loop = True
369
370             self._visit_block(node.body)

```

```

371
372     self.in_loop = prev_in_loop
373
374     if node.post:
375         self.visit(node.post)
376
377     def visit_ReturnStatement(self, node: ReturnStatement) -> None:
378         """
379         Instrução return:
380         - Tem de estar dentro de uma função.
381         - Se a função não for void, tem de retornar um valor compatível.
382         - Se a função for void, não pode retornar valor.
383         """
384         if not self.current_function:
385             raise SemanticError("Return statement outside function", node)
386
387         func_symbol = self.symbol_table.lookup(self.current_function)
388         if not func_symbol:
389             raise SemanticError(f"Cannot find current function
390                                 {self.current_function}", node)
391
392         if node.value:
393             return_type = self.visit(node.value)
394             if not self.type_compatible(return_type, func_symbol.return_type):
395                 raise SemanticError(
396                     f"Return type mismatch: expected {func_symbol.return_type},
397                     got {return_type}",
398                     node
399                 )
400             node.inferred_type = return_type
401         else:
402             # Não há valor no return
403             if func_symbol.return_type != "void":
404                 raise SemanticError(
405                     f"Function {self.current_function} must return a value of type
406                     {func_symbol.return_type}",
407                     node
408                 )
409             node.inferred_type = "void"
410
411     def visit_BreakStatement(self, node: BreakStatement) -> None:
412         """
413         break só é válido dentro de loops.
414         """
415         if not self.in_loop:
416             raise SemanticError("Break statement outside loop", node)
417

```

```

415     def visit_ContinueStatement(self, node: ContinueStatement) -> None:
416         """
417         continue só é válido dentro de loops.
418         """
419         if not self.in_loop:
420             raise SemanticError("Continue statement outside loop", node)
421
422     def visit_BinaryOp(self, node: BinaryOp) -> str:
423         """
424         Operadores binários:
425         - +, -, *, /, % => aritméticos
426         - <, <=, >, >= => comparações
427         - ==, != => igualdade/ desigualdade
428         - &&, || => lógicos
429         Verifica compatibilidade de tipos e define node.inferred_type.
430         """
431         left_type = self.visit(node.left)
432         right_type = self.visit(node.right)
433         op = node.operator
434
435         # Operadores aritméticos
436         if op in {"+", "-", "*", "/", "%"}:
437             # Exceção: concatenação de strings usando +
438             if op == "+" and left_type == "string" and right_type == "string":
439                 node.inferred_type = "string"
440             return node.inferred_type
441
442         # Caso normal: ambos operandos têm de ser int ou float
443         if left_type not in {"int", "float"} or right_type not in {"int",
444             "float"}:
445             raise SemanticError(
446                 f"Arithmetic op '{op}' not supported between {left_type} and
447                 {right_type}",
448                 node
449             )
450         # Se um deles for float, o resultado é float; caso contrário, int
451         node.inferred_type = "float" if "float" in {left_type, right_type}
452         else "int"
453         return node.inferred_type
454
455         # Operadores de comparação (<, <=, >, >=)
456         elif op in {"<", "<=", ">", ">="}:
457             if left_type not in {"int", "float"} or right_type not in {"int",
458                 "float"}:
459                 raise SemanticError(
460                     f"Comparison '{op}' not supported between {left_type} and
461                     {right_type}",

```

```

457         node
458     )
459     node.inferred_type = "bool"
460     return node.inferred_type
461
462     # Operadores de igualdade (==, !=)
463     elif op in {"==", "!="}:
464         if not self.type_compatible(left_type, right_type):
465             raise SemanticError(f"Cannot compare {left_type} and
466                                 {right_type}", node)
467         node.inferred_type = "bool"
468         return node.inferred_type
469
470     # Operadores lógicos (&&, ||)
471     elif op in {"&&", "||"}:
472         if left_type != "bool" or right_type != "bool":
473             raise SemanticError(
474                 f"Logical op '{op}' not supported between {left_type} and
475                 {right_type}",
476                 node
477             )
478         node.inferred_type = "bool"
479         return node.inferred_type
480
481     raise SemanticError(f"Unrecognized binary operator: {op}", node)
482
483     def visit_UnaryOp(self, node: UnaryOp) -> str:
484         """
485         Operadores unários:
486         - '!' (NOT lógico),
487         - '-' (negativo aritmético),
488         - '++', '--' (incremento/decremento).
489         Verifica o tipo do operando e define node.inferred_type.
490         """
491         operand_type = self.visit(node.operand)
492         op = node.operator
493
494         if op == "!":
495             if operand_type != "bool":
496                 raise SemanticError(f"Logical NOT requires bool, got
497                                     {operand_type}", node)
498             node.inferred_type = "bool"
499             return node.inferred_type
500
501         elif op == "-":
502             if operand_type not in {"int", "float"}:

```



```

500         raise SemanticError(f"Unary minus requires numeric type, got
           {operand_type}", node)
501     node.inferred_type = operand_type
502     return node.inferred_type
503
504     elif op in {"++", "--"}:
505         # i++, i-- => Têm de ser variáveis do tipo int
506         if not isinstance(node.operand, Variable):
507             raise SemanticError("Increment/decrement requires variable
           operand", node)
508         if operand_type != "int":
509             raise SemanticError(
510                 f"Increment/decrement requires int operand, got
           {operand_type}",
511                 node
512             )
513         node.inferred_type = "int"
514         return node.inferred_type
515
516     raise SemanticError(f"Unrecognized unary operator: {op}", node)
517
518     def visit_Literal(self, node: Literal) -> str:
519         """
520         Visita literal (int, float, bool, string) e devolve o seu tipo.
521         Armazena em node.inferred_type.
522         """
523         node.inferred_type = node.type
524         return node.inferred_type
525
526     def visit_Variable(self, node: Variable) -> str:
527         """
528         Visita uma variável: procura na tabela de símbolos e obtém o tipo.
529         Armazena em node.inferred_type.
530         """
531         symbol = self.symbol_table.lookup(node.name)
532         if not symbol:
533             raise SemanticError(f"Undefined variable: {node.name}", node)
534         node.inferred_type = symbol.type
535         return node.inferred_type
536
537     def visit_ArrayAccess(self, node: ArrayAccess) -> str:
538         """
539         Acesso a array: array[i][j]...
540         - Verifica se 'array' é ArrayType
541         - Verifica se o número de índices corresponde ao número de dimensões
542         - Cada índice tem de ser int
543         - O tipo resultante é o base_type do array

```

```

544         """
545         array_type = self.visit(node.array)
546         if not isinstance(array_type, ArrayType):
547             raise SemanticError(f"Cannot index into non-array type {array_type}",
548                                 node)
549
550         if len(node.indices) != len(array_type.dimensions):
551             raise SemanticError(
552                 f"Wrong number of dimensions in array access: "
553                 f"expected {len(array_type.dimensions)}, got {len(node.indices)}",
554                 node
555             )
556
557         for i, idx_expr in enumerate(node.indices):
558             idx_type = self.visit(idx_expr)
559             if idx_type != "int":
560                 raise SemanticError(f"Array index {i} must be int, got
561                                     {idx_type}", node)
562
563         node.inferred_type = array_type.base_type
564         return node.inferred_type
565
566     def visit_FunctionCall(self, node: FunctionCall) -> str:
567         """
568         Chamada de função:
569         1. Verificar se a função existe na tabela de símbolos (e se é função).
570         2. Verificar contagem e tipos de argumentos.
571         3. Retornar o tipo de retorno da função.
572         """
573         func_symbol = self.symbol_table.lookup(node.name)
574         if not func_symbol:
575             raise SemanticError(f"Undefined function: {node.name}", node)
576         if not func_symbol.is_function:
577             raise SemanticError(f"{node.name} is not a function", node)
578
579         # Caso especial: print() pode ter vários argumentos de tipos variados
580         if node.name == "print":
581             for arg in node.arguments:
582                 arg_type = self.visit(arg)
583                 # Aceitamos vários tipos para print, não é feita verificação
584                 # estrita
585             node.inferred_type = "void"
586             return node.inferred_type
587
588         # Funções built-in com um único argumento (toInt, toFloat, toStr)
589         if node.name in {"toInt", "toFloat", "toStr"}:
590             if len(node.arguments) != 1:

```

```

587         raise SemanticError(f"{node.name}() expects exactly one argument",
                               node)
588     arg_type = self.visit(node.arguments[0])
589     if arg_type not in self.basic_types and arg_type != "string":
590         raise SemanticError(f"Cannot convert type {arg_type} to
                               {node.name}", node)
591     node.inferred_type = func_symbol.return_type
592     return node.inferred_type
593
594     # Verificar número de argumentos para funções normais
595     if len(node.arguments) != len(func_symbol.params):
596         raise SemanticError(
597             f"Function {node.name} expects {len(func_symbol.params)}
              arguments, got {len(node.arguments)}",
598             node
599         )
600
601     # Verificar cada argumento
602     for i, (arg, param) in enumerate(zip(node.arguments, func_symbol.params)):
603         arg_type = self.visit(arg)
604         # Se param["type"] == "any", não verificamos compatibilidade
605         if param["type"] != "any" and not self.type_compatible(arg_type,
                                                                    param["type"]):
606             raise SemanticError(
607                 f"Type mismatch in argument {i+1} of {node.name}: "
608                 f"expected {param['type']}, got {arg_type}",
609                 node
610             )
611
612     node.inferred_type = func_symbol.return_type
613     return node.inferred_type
614
615     def visit_ArrayInitializer(self, node: ArrayInitializer) -> ArrayType:
616         """
617         Inicializador de array (Ex.: {{1,2},{3,4}}):
618         1. Analisa recursivamente a estrutura para determinar dimensões e tipo
           base.
619         2. Cria um ArrayType correspondente e armazena em node.inferred_type.
620         """
621         if not node.elements:
622             raise SemanticError("Empty array initializer", node)
623
624         dimensions, base_type =
625         self._analyze_array_initializer_level(node.elements)
626         arr_type = ArrayType(base_type=base_type, dimensions=dimensions,
627                               lineno=node.lineno)
628         node.inferred_type = arr_type
629         return arr_type

```

```

628
629     def _analyze_array_initializer_level(self, elements: List[Any]):
630         """
631         Função recursiva para determinar dimensões e tipo base de um nível do
        array initializer.
632         Retorna (lista_de_dimensões, tipo_base_string).
633         """
634         if not elements:
635             return [], None
636
637         # A primeira dimensão é o tamanho de 'elements'
638         dimensions = [len(elements)]
639         first = elements[0]
640
641         # Se o primeiro elemento for outro ArrayInitializer, descer mais um nível
642         if isinstance(first, ArrayInitializer):
643             subdims, elem_type =
644                 self._analyze_array_initializer_level(first.elements)
645             dimensions.extend(subdims)
646         else:
647             # Caso contrário, é um literal ou algo que tem um tipo
648             elem_type = self.visit(first)
649
650         # Verificar consistência nos elementos seguintes
651         for elem in elements[1:]:
652             if isinstance(first, ArrayInitializer) != isinstance(elem,
653                 ArrayInitializer):
654                 raise SemanticError("Inconsistent array structure in initializer",
655                     node=elem)
656
657             if isinstance(elem, ArrayInitializer):
658                 subdims2, sub_type =
659                     self._analyze_array_initializer_level(elem.elements)
660                 if subdims2 != subdims or sub_type != elem_type:
661                     raise SemanticError("Inconsistent dimensions or types in array
662                         initializer", node=elem)
663             else:
664                 if self.visit(elem) != elem_type:
665                     raise SemanticError("Inconsistent types in array initializer",
666                         node=elem)
667
668         return dimensions, elem_type
669
670     def visit_ExpressionStatement(self, node: ExpressionStatement) -> None:
671         """
672         Instrução que é apenas uma expressão (por exemplo, chamada de função
673         sozinha).
674         Basta visitar a expressão para validar.

```

```

668         """
669         self.visit(node.expression)
670
671     def visit_SwitchStatement(self, node: SwitchStatement) -> None:
672         """
673         Instrução switch:
674         1. Visita a expressão do switch para determinar o tipo base,
675         2. Para cada case, verifica compatibilidade se for literal,
676         3. Verifica duplicados nos case (para literais),
677         4. Visita cada body em novo scope e marca in_loop=True para permitir
678            break.
679         5. Se houver default, visita o corpo.
680         """
681         switch_type = self.visit(node.expression)
682
683         with self.symbol_table.new_scope():
684             seen_values = set()
685
686             for case in node.cases:
687                 case_type = self.visit(case.value)
688                 if not self.type_compatible(case_type, switch_type):
689                     raise SemanticError(
690                         f"Switch case type mismatch: cannot compare {switch_type}
691                         with {case_type}",
692                         node=case
693                     )
694
695                 # Se o case for literal, verificar duplicados
696                 case_val = case.value.value if isinstance(case.value, Literal)
697                 else None
698                 if case_val in seen_values:
699                     raise SemanticError(f"Duplicate case value: {case_val}",
700                                         node=case)
701                 if case_val is not None:
702                     seen_values.add(case_val)
703
704             # Neste contexto, consideramos o switch como loop-friendly para
705             # permitir break
706             prev_in_loop = self.in_loop
707             self.in_loop = True
708
709             self._visit_block(case.body)
710
711             self.in_loop = prev_in_loop
712
713             # Visitar default (se existir)
714             if node.default:


```

710

`self._visit_block(node.default)`

## 8.5. Code Generator

### CODE GENERATOR

 Python

```

1  from c_ast.ast_nodes import (
2      ArrayInitializer,
3      VariableDeclaration,
4      FunctionDeclaration,
5      ArrayType,
6      BinaryOp,
7      Literal,
8      Variable,
9      ExpressionStatement,
10     FunctionCall,
11     ArrayAssignment,
12     ArrayAccess,
13     ReturnStatement,
14     IfStatement,
15     ForStatement,
16     SwitchStatement,
17     BreakStatement,
18     ContinueStatement,
19     Assignment,
20     UnaryOp,
21     Program
22 )
23
24
25 class CodeGenerator:
26     """
27     A classe CodeGenerator percorre uma AST semanticamente válida e produz código
28     assembly.
29     Assumimos que todas as verificações semânticas (tipagem, escopo, etc.) já
30     foram feitas,
31     e que cada nó de expressão possui um atributo 'inferred_type'.
32     """
33
34     def __init__(self):
35         # Lista de instruções assembly geradas
36         self.assembly = []
37
38         # Informação de contexto / escopo
39         self.current_scope = "global" # Escopo atual (p.e., global ou nome de
40         função)
41         self.label_counter = 0 # Contador para geração de rótulos únicos

```

```

39
40     # Variáveis globais e locais
41     self.global_vars = {} # Dicionário: nome_variável -> {address, type}
42     self.local_vars = {} # Dicionário: nome_variável -> {offset, type, ...}
                           para a função atual
43
44     # Função atualmente em visita
45     self.current_function = None
46
47     # Endereços / offsets para armazenamento
48     self.next_global_addr = 0
49     self.next_local_offset = 0
50
51     # Contador de parâmetros de função
52     self.param_count = 0
53
54     # Stacks para controlo de loops / switches (break/continue)
55     self.break_stack = [] # Stack de tuplos (tipo, end_label)
56     self.continue_stack = [] # Stack de rótulos de 'continue' para loops
57
58     # Tipos de retorno de funções (preenchido ao visitar declarações de
                           função)
59     self.return_types = {}
60
61     def init_builtins(self):
62         """
63         Inicializa funções internas (built-in) conhecidas e os seus tipos de
                           retorno,
64         para que o gerador de código consiga lidar corretamente com chamadas a
                           estas funções.
65         """
66         self.return_types["print"] = "void"
67         self.return_types["read"] = "string"
68         self.return_types["toInt"] = "int"
69         self.return_types["toFloat"] = "float"
70         self.return_types["toStr"] = "string"
71         self.return_types["len"] = "int"
72
73     def visit(self, node):
74         """
75         Função central de despacho de visitas.
76         Invoca dinamicamente o método 'visit_<nome_da_classe>' baseado no tipo de
                           nó da AST.
77         """
78         if node is None:
79             return None
80         method_name = f"visit_{node.__class__.__name__}"

```

```

81     visitor = getattr(self, method_name, self.generic_visit)
82     return visitor(node)
83
84     def generic_visit(self, node):
85         """
86         Chamado se não existir um método específico de visita para o tipo de nó.
87         """
88         raise Exception(f"Nenhum método visit_{type(node).__name__} definido")
89
90     def reset_local_context(self):
91         """
92         Limpa / reinicia as variáveis locais ao entrar numa nova função,
93         preparando para analisar a sua lista de variáveis.
94         """
95         self.local_vars = {}
96         self.next_local_offset = 0
97         self.param_count = 0
98
99     def emit(self, instruction, comment=None):
100         """
101         Adiciona uma instrução à lista self.assembly, com um comentário opcional.
102         O comentário aparece no código gerado, facilitando a depuração.
103         """
104         if comment:
105             self.assembly.append(f"\t{instruction}\t// {comment}")
106         else:
107             self.assembly.append(f"\t{instruction}")
108
109     def generate_label(self, prefix="L"):
110         """
111         Gera um rótulo único (ex.: L1, L2, ...) para uso em instruções de salto
112         (branch/loops).
113         """
114         self.label_counter += 1
115         return f"{prefix}{self.label_counter}"
116
117     def count_local_vars(self, statements):
118         """
119         Conta quantas variáveis locais são declaradas num corpo de função,
120         para saber quanta memória deve ser reservada na stack local.
121         """
122         count = 0
123         if not statements:
124             return 0
125         # Se statements for uma lista, percorre; caso contrário, põe numa lista e percorre
126         for stmt in statements if isinstance(statements, list) else [statements]:

```



```

126         if isinstance(stmt, VariableDeclaration):
127             count += 1
128         return count
129
130     def generate_code(self, ast) -> str:
131         """
132         Orquestra a geração de código para toda a AST:
133         1) Inicializa as funções built-in
134         2) Visita o nó raiz Program
135         3) Retorna o texto das instruções assembly unidas por nova linha
136         """
137         self.init_builtins()
138         self.visit(ast)
139         return "\n".join(self.assembly)
140
141     # -----
142     # Métodos de visita de alto nível (Program, Função, etc.)
143     # -----
144
145     def visit_Program(self, node: Program):
146         """
147         Visita o nó principal Program.
148         1) Recolhe tipos de retorno de cada função
149         2) Aloca e gera código para variáveis globais
150         3) Emite 'start' + chamada opcional para main
151         4) Emite 'stop'
152         5) Gera código para cada declaração de função
153         """
154         # 1) Recolher tipos de retorno das funções
155         for decl in node.declarations:
156             if isinstance(decl, FunctionDeclaration):
157                 self.return_types[decl.name] = decl.return_type
158
159         # 2) Declarar variáveis globais
160         self.emit("// Global variable declarations")
161         for decl in node.declarations:
162             if isinstance(decl, VariableDeclaration):
163                 addr = self.next_global_addr
164                 self.global_vars[decl.name] = {"address": addr, "type":
165                 decl.var_type}
166                 self.next_global_addr += 1
167                 self.visit_global_var_decl(decl)
168
169         # 3) Iniciar programa + chamada opcional a main
170         self.emit("start", "Program start")
171         if "main" in self.return_types:

```

```

171         self.emit("pusha main", "Call main function")
172         self.emit("call")
173
174     # 4) Finalizar programa
175     self.emit("stop", "Program end")
176
177     # 5) Gerar código para funções
178     for decl in node.declarations:
179         if isinstance(decl, FunctionDeclaration):
180             self.visit(decl)
181
182     def visit_global_var_decl(self, node: VariableDeclaration):
183         """
184         Lida com uma declaração global de variável (incluindo arrays):
185         Aloca memória, inicializa se necessário e guarda o endereço global.
186         """
187         addr = self.global_vars[node.name]["address"]
188         self.emit(f"// Global variable {node.name} at address {addr}")
189
190     # Se o tipo for array
191     if isinstance(node.var_type, ArrayType):
192         if node.initializer:
193             # Calcula o tamanho total a partir das dimensões
194             total_size = 1
195             for dim in node.initializer.dimensions:
196                 total_size *= dim
197             self.emit(f"alloc {total_size}", f"Allocate array of total size {total_size}")
198
199             # Inicializa cada elemento, se presente
200             if node.initializer.elements:
201                 for i, elem in enumerate(node.initializer.elements):
202                     if isinstance(elem, ArrayInitializer):
203                         # Caso de array aninhado
204                         for j, nested_elem in enumerate(elem.elements):
205                             self.emit("dup 1") # Duplica a referência ao array
206                             offset = i * node.initializer.dimensions[1] + j
207                             self.emit(f"pushi {offset}")
208                             self.visit(nested_elem)
209                             self.emit("storen")
210                         else:
211                             # Array simples (1D)
212                             self.emit("dup 1")
213                             self.emit(f"pushi {i}")
214                             self.visit(elem)
215                             self.emit("storen")

```

```

216         else:
217             # Sem inicializador => alocar espaço padrão
218             total_size = 1
219             for dim in node.var_type.dimensions:
220                 if dim is not None:
221                     total_size *= dim
222             self.emit(f"alloc {total_size}")
223
224         # Se for variável simples (não-array)
225         elif node.initializer:
226             self.visit(node.initializer)
227         else:
228             # Inicialização por omissão
229             if node.var_type == "float":
230                 self.emit("pushf 0.0")
231             elif node.var_type == "string":
232                 self.emit('pushs ""')
233             else:
234                 self.emit("pushi 0")
235
236         self.emit(f"storeg {addr}")
237
238     def visit_FunctionDeclaration(self, node: FunctionDeclaration):
239         """
240         Gera código para uma função:
241         1) Limpa o contexto de variáveis locais
242         2) Reserva espaço para variáveis locais
243         3) Emite o corpo da função
244         4) Se a função for void, emite um return
245         """
246         self.current_function = node
247         self.reset_local_context()
248
249         # Processa parâmetros: armazena-os em local_vars
250         for i, param in enumerate(node.params, start=1):
251             self.local_vars[param["name"]] = {
252                 "offset": i,
253                 "type": param["type"],
254                 "param": True,
255                 "param_num": i,
256             }
257
258         # Contagem das variáveis locais no corpo
259         local_var_count = self.count_local_vars(node.body)
260
261         # Emite o rótulo da função

```

```

262         self.emit(f"\n{node.name}:", f"Function {node.name} declaration")
263
264         # Prólogo (reserva espaço para variáveis locais)
265         if local_var_count > 0:
266             self.emit(
267                 f"pushn {local_var_count}",
268                 f"Reserve space for {local_var_count} local variables",
269             )
270
271         # Visita o corpo (que pode ser lista de statements ou único statement)
272         if isinstance(node.body, list):
273             for stmt in node.body:
274                 self.visit(stmt)
275         else:
276             self.visit(node.body)
277
278         # Se for void, garante um return
279         if node.return_type == "void":
280             self.emit("return", "Return from void function")
281
282         self.current_function = None
283
284     def visit_VariableDeclaration(self, node: VariableDeclaration):
285         """
286         Lida com declaração de variável local (com inicializador opcional).
287         """
288         if self.current_function:
289             # É variável local
290             offset = self.next_local_offset
291             self.local_vars[node.name] = {"offset": offset, "type": node.var_type}
292             self.next_local_offset += 1
293
294             # Se for array
295             if isinstance(node.var_type, ArrayType):
296                 if node.initializer:
297                     self.visit_array_initializer(node.initializer,
298                                                  node.initializer.dimensions)
299                 else:
300                     # Aloca para array com dimensões conhecidas
301                     total_size = 1
302                     for dim in node.var_type.dimensions:
303                         if dim is not None:
304                             total_size *= dim
305                     self.emit(f"alloc {total_size}")
306             elif node.initializer:

```

```

307         # Inicializador simples
308         self.visit(node.initializer)
309     else:
310         # Inicialização por omissão
311         if node.var_type == "float":
312             self.emit("pushf 0.0")
313         elif node.var_type == "string":
314             self.emit('pushs ""')
315         else:
316             self.emit("pushi 0")
317
318         self.emit(f"storel {offset}")
319
320     def visit_array_initializer(self, initializer: ArrayInitializer, dimensions,
321                               current_dim=0):
322         """
323         Inicializa um array multidimensional num único bloco contíguo.
324         'Flatten' (achatar) os inicializadores aninhados e aloca.
325         """
326         # Se o 'initializer' não for ArrayInitializer (pode ser Literal), apenas
327         # visita-o.
328         if not isinstance(initializer, ArrayInitializer):
329             self.visit(initializer)
330             return
331
332         actual_dimensions = initializer.dimensions
333         total_size = 1
334         for dim in actual_dimensions:
335             total_size *= dim
336
337         # Aloca bloco único
338         self.emit(f"alloc {total_size}")
339
340         # 'Flatten' todos os elementos e armazena
341         flat_elements = self._flatten_array_initializer(initializer)
342         for i, elem in enumerate(flat_elements):
343             self.emit("dup 1")
344             self.emit(f"pushi {i}")
345             self.visit(elem)
346             self.emit("storen")
347
348     def _flatten_array_initializer(self, initializer: ArrayInitializer):
349         """
350         Função auxiliar para achatar inicializadores de array aninhados num só
351         nível de lista.
352         """
353         if not isinstance(initializer, ArrayInitializer):

```

```

351         return [initializer]
352     flattened = []
353     for elem in initializer.elements:
354         if isinstance(elem, ArrayInitializer):
355             flattened.extend(self._flatten_array_initializer(elem))
356         else:
357             flattened.append(elem)
358     return flattened
359
360     # -----
361     # Visita de nós de instrução (Assignment, If, For, Switch, etc.)
362     # -----
363
364     def visit_Assignment(self, node: Assignment):
365         """
366         Atribuição a uma variável simples:
367         1) Avalia o lado direito (value)
368         2) Armazena no alvo (global ou local)
369         """
370         self.visit(node.value)
371         var_name = node.target.name
372         if var_name in self.global_vars:
373             addr = self.global_vars[var_name]["address"]
374             self.emit(f"storeg {addr}")
375         else:
376             offset = self.local_vars[var_name]["offset"]
377             self.emit(f"storel {offset}")
378
379     def visit_ArrayAssignment(self, node: ArrayAssignment):
380         """
381         Atribuição a um elemento de array:
382         1) Dá stack a referência base do array
383         2) Calcula o índice linear
384         3) Avalia o valor
385         4) Usa 'storen' para armazenar no local correto
386         """
387         # Dá stack á referência do array
388         self.visit(node.array)
389
390         # Calcula índice linear
391         array_type = node.array.inferred_type
392         if isinstance(array_type, ArrayType):
393             self.calculate_array_index(array_type.dimensions, node.indices)
394         else:
395             # Fallback para array de dimensão única ou cenário de erro semântico
396             self.visit(node.index)

```

```

397
398     # Avalia o valor e armazena
399     self.visit(node.value)
400     self.emit("storen")
401
402     def calculate_array_index(self, dimensions, indices):
403         """
404         Calcula o índice linear para acesso a um array multidimensional.
405         Ex.: para array[M][N], o acesso [i][j] => i * N + j
406         """
407         if len(dimensions) != len(indices):
408             # O analisador semântico já deve ter verificado isto
409             return
410
411         steps = []
412         step = 1
413         # Constrói lista 'steps' de trás para a frente
414         for dim in reversed(dimensions[1:]):
415             steps.append(step)
416             step *= dim
417         steps.append(step)
418         steps.reverse()
419
420         first = True
421         for s, idx_node in zip(steps, indices):
422             self.visit(idx_node)
423             if s != 1:
424                 self.emit(f"pushi {s}")
425                 self.emit("mul")
426             if not first:
427                 self.emit("add")
428             first = False
429
430     def visit_BinaryOp(self, node: BinaryOp):
431         """
432         Avalia operandos esquerdo e direito, depois emite a instrução
433         correspondente.
434
435         Utiliza node.left.inferred_type, node.right.inferred_type e
436         node.inferred_type
437         para selecção da instrução final (int, float, string, etc.).
438         """
439         self.visit(node.left)
440         self.visit(node.right)
441
442         left_type = node.left.inferred_type
443         right_type = node.right.inferred_type
444         operator = node.operator

```

```

442
443     # Mapeamento da instrução
444     self._emit_binary_op(operator, left_type, right_type, node.inferred_type)
445
446     def _emit_binary_op(self, operator, left_type, right_type, result_type):
447         """
448         Função auxiliar para escolher a instrução assembly adequada
449         a uma dada operação binária.
450         """
451         op_map = {
452             "+": "add",
453             "-": "sub",
454             "*": "mul",
455             "/": "div",
456             "%": "mod",
457             "<": "inf",
458             "<=": "infeq",
459             ">": "sup",
460             ">=": "supeq",
461             "==": "equal",
462             "!=": "equal\n\tnot",
463             "&&": "and",
464             "||": "or",
465         }
466
467         float_op_map = {
468             "+": "fadd",
469             "-": "fsub",
470             "*": "fmul",
471             "/": "fdiv",
472             "<": "finf",
473             "<=": "finfeq",
474             ">": "fsup",
475             ">=": "fsupeq",
476         }
477
478         # Concatenar strings com +
479         if left_type == "string" and right_type == "string" and operator == "+":
480             self.emit("swap")
481             self.emit("concat")
482             return
483
484         # Se o resultado for float, usar instruções float
485         if result_type == "float":
486             # Converter operandos int em float, se necessário
487             if left_type == "int":

```



```

488         self.emit("itof")
489     if right_type == "int":
490         self.emit("itof")
491     # Usar float_op_map se disponível, senão fallback
492     self.emit(float_op_map.get(operator, op_map[operator]))
493     elif result_type == "string":
494         # (Normalmente só para + entre strings, já tratado acima)
495         pass
496     else:
497         # Caso contrário, operações int / bool
498         inst = op_map.get(operator)
499         if inst:
500             self.emit(inst)
501
502     def visit_UnaryOp(self, node: UnaryOp):
503         """
504         Lida com operadores unários: '-', '!', '++', '--'.
505         Pressupõe que a análise semântica validou o uso.
506         """
507         op = node.operator
508         if op in ["++", "--"]:
509             # Pré-incremento/decremento
510             var_name = node.operand.name
511             self.visit(node.operand) # Stack o valor atual
512             self.emit("pushi 1")
513             if op == "++":
514                 self.emit("add")
515             else:
516                 self.emit("sub")
517
518             # Armazena de volta
519             if var_name in self.global_vars:
520                 addr = self.global_vars[var_name]["address"]
521                 self.emit(f"storeg {addr}")
522             else:
523                 offset = self.local_vars[var_name]["offset"]
524                 self.emit(f"storel {offset}")
525
526         elif op == "-":
527             # Negação aritmética
528             self.emit("pushi 0")
529             self.visit(node.operand)
530             self.emit("sub")
531         elif op == "!":
532             # Negação lógica
533             self.visit(node.operand)

```

```

534         self.emit("not")
535
536     def visit_Literal(self, node: Literal):
537         """
538         Dá stack do literal (int, float, string, bool).
539         """
540         if node.inferred_type == "int":
541             self.emit(f"pushi {node.value}")
542         elif node.inferred_type == "float":
543             self.emit(f"pushf {node.value}")
544         elif node.inferred_type == "string":
545             self.emit(f"pushs \"{node.value}\"")
546         elif node.inferred_type == "bool":
547             self.emit(f"pushi {1 if node.value else 0}")
548
549     def visit_Variable(self, node: Variable):
550         """
551         Dá stack ao valor de uma variável (global ou local).
552         Se for parâmetro de função, lida com offsets de FP conforme a convenção
553         usada.
554         """
555         var_name = node.name
556         if var_name in self.global_vars:
557             addr = self.global_vars[var_name]["address"]
558             self.emit(f"pushg {addr}")
559         elif var_name in self.local_vars:
560             var_info = self.local_vars[var_name]
561             if var_info.get("param"):
562                 # Parâmetro de função, offset pode ser negativo em certas
563                 # convenções
564                 param_num = var_info["param_num"]
565                 self.emit("pushfp", f"Access param {var_name}")
566                 self.emit(f"load -{param_num}", f"Load param at offset -
567                 {param_num}")
568             else:
569                 # Variável local normal
570                 offset = var_info["offset"]
571                 self.emit(f"pushl {offset}")
572
573     def visit_ArrayAccess(self, node: ArrayAccess):
574         """
575         Acesso a elemento de array (que pode ser multidimensional):
576         1) Stack á referência base
577         2) Calcula o índice linear
578         3) Usa 'loadn' para ler o valor
579         """
580         self.visit(node.array)

```

```

578     array_type = node.array.inferred_type
579     if not isinstance(array_type, ArrayType):
580         return # O analisador semântico terá reportado erro antes
581
582     # Calcula índice linear
583     self.calculate_array_index(array_type.dimensions, node.indices)
584
585     # Carrega valor (loadn)
586     self.emit("loadn")
587
588     def visit_FunctionCall(self, node: FunctionCall):
589         """
590         Chamadas de função:
591         1) Avaliar argumentos (em ordem inversa para dar stack corretamente)
592         2) pusha <function_name>
593         3) call
594         4) Se a função retorna valor mas o chamador é um ExpressionStatement,
595            descarta-se (pop 1).
596         """
597         # Se for função built-in, tratar separadamente
598         if node.name in ["print", "read", "toInt", "toFloat", "toStr", "len"]:
599             self.visit_builtin_function_call(node)
600             return
601
602         # Dá stack aos argumentos em ordem inversa
603         for arg in reversed(node.arguments):
604             self.visit(arg)
605
606         self.emit(f"pusha {node.name}")
607         self.emit("call")
608
609         # Verifica se retorna valor
610         returns_value = (node.inferred_type != "void")
611         parent = getattr(self, "current_statement", None)
612         is_expression_stmt = isinstance(parent, ExpressionStatement)
613
614         # Se retorna valor mas está a ser usado como statement, faz pop
615         if returns_value and is_expression_stmt:
616             self.emit("pop 1")
617
618     def visit_builtin_function_call(self, node: FunctionCall):
619         """
620         Trata separadamente as funções internas (built-in).
621         """
622         if node.name == "print":
623             # Imprimir cada argumento

```

```

624         for arg in node.arguments:
625             arg_type = arg.inferred_type
626             if arg_type is None:
627                 arg_type = "int" # Fallback
628
629             # Caso hipotético de imprimir um array
630             if isinstance(arg_type, ArrayType):
631                 self.visit(arg)
632                 self.emit("writeln") # Exemplo simples
633             else:
634                 self.visit(arg)
635                 if arg_type == "float":
636                     self.emit("writef")
637                 elif arg_type == "string":
638                     self.emit("writes")
639                 else:
640                     self.emit("writei")
641
642             # Se há argumentos e o último não for void, salta linha
643             if node.arguments and node.arguments[-1].inferred_type != "void":
644                 self.emit("writeln")
645
646         elif node.name == "read":
647             self.emit("read")
648
649         elif node.name == "toInt":
650             self.visit(node.arguments[0])
651             arg_type = node.arguments[0].inferred_type
652             if arg_type == "float":
653                 self.emit("ftoi")
654             elif arg_type == "string":
655                 self.emit("atoi")
656
657         elif node.name == "toFloat":
658             self.visit(node.arguments[0])
659             arg_type = node.arguments[0].inferred_type
660             if arg_type == "int":
661                 self.emit("itof")
662             elif arg_type == "string":
663                 self.emit("atof")
664
665         elif node.name == "toStr":
666             self.visit(node.arguments[0])
667             arg_type = node.arguments[0].inferred_type
668             if arg_type == "int":
669                 self.emit("stri")

```

```

670         elif arg_type == "float":
671             self.emit("strf")
672
673         elif node.name == "len":
674             self.visit(node.arguments[0])
675             arg_type = node.arguments[0].inferred_type
676             if isinstance(arg_type, ArrayType):
677                 # Para multidimensional, poderíamos retornar a 1ª dimensão
678                 self.emit(f"pushi {arg_type.dimensions[0]}")
679             elif arg_type == "string":
680                 self.emit("strlen")
681             else:
682                 pass # Caso improvável, analisador semântico teria tratado
683
684     def visit_ReturnStatement(self, node: ReturnStatement):
685         """
686         Instrução de retorno de função. Se houver valor, adiciona-o á primeiro.
687         """
688         if node.value:
689             self.visit(node.value)
690             self.emit("return")
691
692     def visit_ExpressionStatement(self, node: ExpressionStatement):
693         """
694         Visita uma expressão usada como statement.
695         Guarda em 'current_statement' para detetar se uma chamada de função
696         tem o valor de retorno não utilizado (caso em que faremos pop).
697         """
698         self.current_statement = node
699         self.visit(node.expression)
700         self.current_statement = None
701
702     def visit_IfStatement(self, node: IfStatement):
703         """
704         Estrutura: if <cond> then <then_branch> else <else_branch>
705         """
706         else_label = self.generate_label("else")
707         end_label = self.generate_label("endif")
708
709         self.visit(node.condition)
710         self.emit(f"jz {else_label}")
711
712         if isinstance(node.then_branch, list):
713             for stmt in node.then_branch:
714                 self.visit(stmt)
715         else:

```

```

716         self.visit(node.then_branch)
717
718         self.emit(f"jump {end_label}")
719         self.emit(f"{else_label}:")
720         if node.else_branch:
721             if isinstance(node.else_branch, list):
722                 for stmt in node.else_branch:
723                     self.visit(stmt)
724             else:
725                 self.visit(node.else_branch)
726
727         self.emit(f"{end_label}:")
728
729     def visit_ForStatement(self, node: ForStatement):
730         """
731         Estrutura: for (<init>; <cond>; <post>) { body }
732         Utiliza stacks de rótulos para tratar break/continue.
733         """
734         start_label = self.generate_label("for")
735         continue_label = self.generate_label("continue")
736         end_label = self.generate_label("endfor")
737
738         # Inicialização
739         if node.init:
740             self.visit(node.init)
741
742         # Dá stack contexto de loop
743         self.break_stack.append(("loop", end_label))
744         self.continue_stack.append(continue_label)
745
746         self.emit(f"{start_label}:")
747
748         if node.condition:
749             self.visit(node.condition)
750             self.emit(f"jz {end_label}")
751
752         if isinstance(node.body, list):
753             for stmt in node.body:
754                 self.visit(stmt)
755         else:
756             self.visit(node.body)
757
758         self.emit(f"{continue_label}:")
759
760         if node.post:
761             self.visit(node.post)

```

```

762
763     self.emit(f"jump {start_label}")
764     self.emit(f"{end_label}:")
765
766     # Sai do contexto de loop
767     self.break_stack.pop()
768     self.continue_stack.pop()
769
770     def visit_BreakStatement(self, node: BreakStatement):
771         """
772         Lida com instruções de break para loops e switch.
773         Sai do constructo mais interno (loop ou switch).
774         """
775         if not self.break_stack:
776             raise Exception("Break statement fora de loop ou switch")
777
778         # Obtém o rótulo de fim do constructo mais interno
779         _, end_label = self.break_stack[-1]
780         self.emit(f"jump {end_label}")
781
782     def visit_ContinueStatement(self, node: ContinueStatement):
783         """
784         Lida com instruções de continue em loops,
785         usando a stack de 'continue_stack'.
786         """
787         if not self.continue_stack:
788             raise Exception("Continue statement fora de loop")
789
790         continue_label = self.continue_stack[-1]
791         self.emit(f"jump {continue_label}")
792
793     def visit_SwitchStatement(self, node: SwitchStatement):
794         """
795         Estrutura: switch <expr> { case <val>: ...; default: ... }
796         Suporta instruções break e execução fall-through.
797         Conserve o valor do switch para cada comparação de caso.
798         """
799         end_label = self.generate_label("endswitch")
800
801         # Dá stack ao contexto de switch
802         self.break_stack.append(("switch", end_label))
803
804         # Avalia a expressão do switch apenas uma vez
805         self.visit(node.expression)
806
807         case_matched = False

```

```

808     next_case_label = None
809
810     for i, case in enumerate(node.cases):
811         # Para cada caso, gera um rótulo para possível fall-through
812         if i > 0:
813             self.emit(f"{next_case_label}:")
814
815         next_case_label = self.generate_label("case") if i < len(node.cases) -
            1 else None
816
817         # Duplica o valor do switch para comparar
818         self.emit("dup 1")
819         self.visit(case.value)
820         self.emit("equal")
821
822         # Se não corresponder e não for o último caso, salta para o próximo
823         if next_case_label:
824             self.emit(f"jz {next_case_label}")
825         else:
826             # Último caso: se não corresponder, salta para default
827             default_label = self.generate_label("default")
828             self.emit(f"jz {default_label}")
829
830         # Caso corresponde -> executa o corpo
831         # self.emit("pop 1") # Remove o resultado da comparação
832         if isinstance(case.body, list):
833             for stmt in case.body:
834                 self.visit(stmt)
835         else:
836             self.visit(case.body)
837
838         # Falha intencional sem jump => fall-through
839
840         # Caso default, se existir
841         if node.default:
842             if next_case_label:
843                 self.emit(f"{next_case_label}:")
844             self.emit(f"{default_label}:")
845             if isinstance(node.default, list):
846                 for stmt in node.default:
847                     self.visit(stmt)
848             else:
849                 self.visit(node.default)
850         elif next_case_label:
851             # Sem default, mas precisamos de um rótulo para o último salto
852             self.emit(f"{next_case_label}:")

```



```

853         self.emit(f"{default_label}:")
854
855     # Marca o fim do switch
856     self.emit(f"{end_label}:")
857
858     # Tira o valor do switch da stack (se ainda existir)
859     self.emit("pop 1")
860
861     # Sai do contexto de switch
862     self.break_stack.pop()

```

## 8.6. Symbol Table

SYMBOL TABLE		Python
1	<code>from contextlib import contextmanager</code>	
2	<code>from typing import Dict, Generator, List, Optional</code>	
3	<code>from c_ast.ast_nodes import Symbol</code>	
4		
5	<code>class SymbolTable:</code>	
6	<code>"""</code>	
7	Implementa uma tabela de símbolos simples, suportando contextos aninhados (nested scopes).	
8	Internamente, mantém uma stack (lista) de dicionários, onde cada dicionário representa um scope.	
9	<code>"""</code>	
10		
11	<code>def __init__(self):</code>	
12	<code># Iniciamos com um scope global (vazio).</code>	
13	<code>self.scopes: List[Dict[str, Symbol]] = [{}]</code>	
14		
15	<code>@contextmanager</code>	
16	<code>def new_scope(self) -&gt; Generator[None, None, None]:</code>	
17	<code>"""</code>	
18	Gerente de contexto (context manager) para lidar com a entrada e saída de um scope:	
19	<code>with self.new_scope():</code>	
20	<code># tudo aqui dentro está num novo scope</code>	
21	<code>"""</code>	
22	<code>self.enter_scope()</code>	
23	<code>try:</code>	
24	<code>yield</code>	
25	<code>finally:</code>	
26	<code>self.exit_scope()</code>	
27		
28	<code>def enter_scope(self):</code>	
29	<code>"""</code>	
30	Entrar num novo scope (push de um dicionário vazio).	
31	<code>"""</code>	

```

32     self.scopes.append({})
33
34     def exit_scope(self):
35         """
36         Sair do scope atual (pop), mantendo sempre pelo menos o scope global.
37         """
38         if len(self.scopes) > 1:
39             self.scopes.pop()
40
41     def define(self, name: str, symbol: Symbol) -> None:
42         """
43         Define um símbolo no scope atual (o mais interno).
44         Lança uma exceção se o símbolo já existir nesse mesmo scope.
45         """
46         if name in self.scopes[-1]:
47             raise Exception(f"Symbol '{name}' already defined in current scope")
48         self.scopes[-1][name] = symbol
49
50     def lookup(self, name: str) -> Optional[Symbol]:
51         """
52         Procura um símbolo de dentro para fora (do scope mais interno ao mais
53         externo).
54         Se encontrar, retorna o símbolo; caso contrário, retorna None.
55         """
56         for scope in reversed(self.scopes):
57             if name in scope:
58                 return scope[name]
59         return None

```