

# Projeto e Implementação de uma Ferramenta de Compilação para a Linguagem TPP

Marcos Rampaso<sup>1</sup>

<sup>1</sup>Departamento Acadêmico de Computação (DACOM)

Universidade Tecnológica Federal do Paraná (UTFPR)

## Abstract

This report delves into the intricacies of programming language implementations. It examines various methodologies, techniques, and best practices utilized in the development and deployment of programming languages. From parsing and lexing to code generation and optimization, the report explores the entire spectrum of language implementation, shedding light on both theoretical foundations and practical considerations. Through in-depth analysis and case studies, it offers valuable insights into the challenges faced by language designers and implementers, as well as innovative solutions driving advancements in the field. Whether you're a seasoned developer, a language enthusiast, or an academic researcher, this report serves as an invaluable resource for understanding and navigating the complex landscape of programming language implementation.

## Resumo

Este relatório abrangente explora as complexidades das implementações de linguagens de programação. Ele examina várias metodologias, técnicas e melhores práticas utilizadas no desenvolvimento e implantação de linguagens de programação. Desde análise sintática e léxica até geração de código e otimização, o relatório explora todo o espectro da implementação de linguagens, lançando luz sobre as bases teóricas e considerações práticas. Através de análises detalhadas e estudos de caso, ele oferece insights valiosos sobre os desafios enfrentados por designers e implementadores de linguagens, assim como soluções inovadoras que impulsionam avanços no campo. Se você é um desenvolvedor experiente, um entusiasta de linguagens ou um pesquisador acadêmico, este relatório serve como um recurso inestimável para compreender e navegar pelo complexo cenário da implementação de linguagens de programação.

## 1 Introdução

No cenário atual, uma ampla gama de implementações de linguagens está disponível, abrangendo desde linguagens compiladas até linguagens interpretadas *just-in-time*. Este trabalho se concentra na exploração do âmbito das linguagens compiladas, com um foco específico na linguagem chamada TPP. Esta linguagem possui todas as características de uma linguagem compilada, e será utilizada como base para os estudos em implementação de linguagens de programação.

Para realizar um processo de compilação, tornam-se necessários alguns passos que são utilizados para guiar um processo de compilação, sendo eles:

- Análise Léxica;
- Análise Sintática;
- Análise Semântica;
- Geração de código.

Cada um destes itens são imprescindíveis na atual implementação de códigos compilados e serão abordados no decorrer do projeto.

## 2 Análise Léxica

A análise léxica é o primeiro estágio do processo de compilação, onde o código fonte é dividido em unidades significativas chamadas tokens. Esses *tokens* são representações dos elementos léxicos da linguagem, como identificadores, palavras-chave e símbolos. O analisador léxico ajuda a identificar e classificar esses *tokens*, preparando o código para a próxima fase do processo de compilação.

### 2.1 Processo de Varredura

O processo de varredura, também conhecido como *scanning* ou *tokenization*, é a primeira etapa da análise léxica em um compilador. Nesta etapa, o código fonte é lido caractere por caractere e dividido em unidades significativas chamadas *tokens*.

O processo de varredura é fundamental para a análise léxica, pois prepara o código fonte para análises posteriores, como análise sintática e análise semântica. É importante que esta etapa seja eficiente e precisa para garantir que o compilador possa entender corretamente o código fonte e gerar saída válida.

### 2.2 Sobre *tokens*

Tokens são unidades básicas de significado em uma linguagem de programação. Eles são os blocos de construção fundamentais que formam o código fonte e são reconhecidos pelo compilador durante o processo de análise léxica. Cada token representa um tipo específico de elemento léxico na linguagem, como palavras-chave, identificadores, números, símbolos e literais. Exemplos de *tokens* são descritos como:

- Palavras-Chave: `if`, `else`, `for`, `while`, `class`, `function`, etc.
- Identificadores: `x`, `valor`, `soma`, `MinhaClasse`, etc.
- Números: `123`, `3.14`, `0xFF`, etc.
- Símbolos: `+`, `-`, `*`, `/`, `=`, `;`, `{`, `}`, `(`, `)`, etc.
- Literais: `"hello"`, `'a'`, `true`, `false`, `null`, etc.

Logo, os *tokens* são essenciais para o processo de compilação, pois fornecem uma representação estruturada e significativa do código fonte, facilitando sua análise, interpretação e transformação em código executável.

### 2.3 Expressões Regulares

Para identificar os *tokens*, comumente são utilizados padrões de teoria da computação, com evidência para o uso de expressões regulares. Uma expressão regular é uma sequência de caracteres que define um padrão de busca, JARGAS (2012). Elas são muito úteis para encontrar padrões complexos em texto, o que as torna ideais para identificar *tokens* em código fonte. Durante a análise léxica, o compilador utiliza expressões regulares para percorrer o código fonte e encontrar correspondências para esses padrões. Quando um padrão é encontrado, o compilador reconhece o trecho correspondente como um token válido e o classifica de acordo com seu tipo.

Por exemplo, ao encontrar a sequência de caracteres `if` em um código fonte, o compilador pode reconhecê-la como uma palavra-chave e criar um token correspondente. Da mesma forma, ao encontrar uma sequência de dígitos, o compilador pode reconhecê-la como um número inteiro e criar um token para o mesmo.

As expressões regulares fornecem uma maneira flexível de identificar tokens em código fonte, permitindo que o compilador reconheça uma ampla variedade de padrões de forma eficiente.

## 2.4 Tokens em TPP

Na linguagem TPP, que será utilizada como base para os estudos deste trabalho, é necessário que se haja o entendimento dos *tokens* da mesma, a fim de gerar a base de conhecimento da linguagem para o bom entendimento do trabalho. Os *tokens* da linguagem estão expressos na Tabela 1.

Token	Descrição
ID	Identificador
NUM_NOTACAO_CIENTIFICA	Número em notação científica
NUM_PONTO_FLUTUANTE	Número em ponto flutuante
NUM_INTEIRO	Número inteiro
MAIS	Operador de adição (+)
MENOS	Operador de subtração (-)
VEZES	Operador de multiplicação (*)
DIVIDE	Operador de divisão (/)
E	Operador lógico E (&&)
OU	Operador lógico OU (  )
DIFERENTE	Operador de diferença (<> ou !=)
MENOR_IGUAL	Operador de menor ou igual (<=)
MAIOR_IGUAL	Operador de maior ou igual (>=)
MENOR	Operador de menor (<)
MAIOR	Operador de maior (>)
IGUAL	Operador de igualdade (=)
NAO	Operador de negação (!)
ABRE_PARENTESE	Símbolo de abre parêntese (()
FECHA_PARENTESE	Símbolo de fecha parêntese ())
ABRE_COLCHETE	Símbolo de abre colchete ([)
FECHA_COLCHETE	Símbolo de fecha colchete (])
VIRGULA	Símbolo de vírgula (,)
DOIS_PONTOS	Símbolo de dois pontos (:)
ATRIBUICAO	Operador de atribuição (:=)

Tabela 1: Tabela de Regras Semânticas

## 2.5 Palavras Reservadas

Palavras reservadas, também conhecidas como palavras-chave, são termos que têm significados específicos e predefinidos em uma linguagem de programação. Elas são utilizadas para representar estruturas de controle, declarações, tipos de dados e outras construções fundamentais da linguagem.

As palavras reservadas são parte da sintaxe da linguagem e não podem ser usadas como identificadores (nomes de variáveis, funções, etc.) pelo programador. Isso significa que elas não podem ser redefinidas ou alteradas de forma alguma, pois são parte integrante da linguagem e têm significados fixos e bem definidos pelo compilador.

O uso de palavras reservadas ajuda a garantir consistência e clareza na linguagem de programação, tornando o código mais legível e compreensível. Elas também são importantes para o processo de análise léxica, onde são identificadas como tokens pelo compilador.

Após a análise dos tokens, é crucial compreender as palavras reservadas da linguagem, que desempenham um papel essencial na estruturação e na semântica do código. Essas palavras, expressas na Tabela 2, possuem significados específicos e predefinidos, sendo utilizadas para representar

estruturas de controle, tipos de dados e outras construções fundamentais. Uma compreensão profunda das palavras reservadas é vital para o desenvolvimento e a interpretação precisa do código fonte.

Palavra Reservada	Token
se	SE
então	ENTAO
senão	SENAO
fim	FIM
repita	REPITA
flutuante	FLUTUANTE
retorna	RETORNA
até	ATE
leia	LEIA
escreva	ESCREVA
inteiro	INTEIRO

Tabela 2: Palavras Reservadas na Linguagem TPP

### 3 Implementação da análise léxica em Python com Ply

Para implementar a análise léxica, optamos pelo uso da biblioteca Ply, Beazley (2024) em Python. O Ply (Python Lex-Yacc) é uma ferramenta que combina as funcionalidades do Lex (para análise léxica) e Yacc (para análise sintática) em uma única ferramenta. Ele permite a especificação de gramáticas e regras diretamente em Python, facilitando o desenvolvimento de analisadores léxicos e sintáticos de forma eficiente e flexível.

#### 3.1 Definição dos Tokens

No início do código do arquivo *tpplex.py*, são definidos os *tokens* que serão reconhecidos pelo *scanner* (Código 1), incluindo identificadores, números (notação científica, ponto flutuante e inteiro), operadores binários e unários, símbolos como parênteses e colchetes, e palavras reservadas da linguagem.

```

1 tokens = [
2     "ID", # identificador
3     # numerais
4     "NUM_NOTACAO_CIENTIFICA", # ponto flutuante em notação científica
5     "NUM_PONTO_FLUTUANTE", # ponto flutuante
6     "NUM_INTEIRO", # inteiro
7     # operadores binarios
8     "MAIS", # +
9     "MENOS", # -
10    "VEZES", # *
11    "DIVIDE", # /
12    "E", # &&
13    "OU", # ||
14    "DIFERENTE", # <>
15    "MENOR_IGUAL", # <=
16    "MAIOR_IGUAL", # >=
17    "MENOR", # <
18    "MAIOR", # >
19    "IGUAL", # =
20    # operadores unarios

```

```

21     "NAO", # !
22     # simbolos
23     "ABRE_PARENTESE", # (
24     "FECHA_PARENTESE", # )
25     "ABRE_COLCHETE", # [
26     "FECHA_COLCHETE", # ]
27     "VIRGULA", # ,
28     "DOIS_PONTOS", # :
29     "ATRIBUICAO", # :=
30     # 'COMENTARIO', # {***}
31 ]
32
33 reserved_words = {
34     "se": "SE",
35     "então": "ENTAO",
36     "senão": "SENAO",
37     "fim": "FIM",
38     "repita": "REPITA",
39     "flutuante": "FLUTUANTE",
40     "retorna": "RETORNA",
41     "até": "ATE",
42     "leia": "LEIA",
43     "escreva": "ESCREVA",
44     "inteiro": "INTEIRO",
45 }

```

Código 1: Definição dos Tokens

Após a definição dos tokens, são especificadas as palavras reservadas que são intrínsecas à linguagem TPP. Tanto os tokens quanto as palavras reservadas estão detalhados na Tabela 1 e na Tabela 2, respectivamente.

### 3.2 Implementação das expressões regulares

A implementação de expressões regulares no código é responsável por definir padrões de texto que serão utilizados para identificar diferentes tipos de elementos dentro do código fonte da linguagem TPP (Código 2). Esses elementos incluem identificadores, números (notação científica, ponto flutuante e inteiros) e operadores.

```

1 tokens = tokens + list(reserved_words.values())
2
3 digito = r"([0-9])"
4 letra = r"([a-zA-ZáÁãÃäÄéÉíÍóÓõÕ])"
5 sinal = r"([\-\+\?]*)"
6
7 """
8     id deve começar com uma letra
9 """
10 id = (
11     r"(" + letra + r"(" + digito + r"+|_|" + letra + r")*)"
12 ) # o mesmo que '((letra)(letra|_|([0-9]))*)'
13
14 # inteiro = r"(" + sinal + digito + r"+)"
15 # inteiro = r"(" + digito + r"+)"
16 inteiro = r"\d+"
17
18 flutuante = (
19     # r"(" + digito + r"+\." + digito + r"+?)"
20     # '(([+-]?)([0-9]+)\.([0-9]+))'

```



```
3 token.type = reserved_words.get(token.value, "ID")
4 # não é necessário fazer regras/regex para cada palavra reservada
5 # se o token não for uma palavra reservada automaticamente é um id
6 # As palavras reservadas têm precedências sobre os ids
7
8 return token
9
10
11 @TOKEN(notacao_cientifica)
12 def t_NUM_NOTACAO_CIENTIFICA(token):
13     return token
14
15
16 @TOKEN(flutuante)
17 def t_NUM_PONTO_FLUTUANTE(token):
18     return token
19
20
21 @TOKEN(inteiro)
22 def t_NUM_INTEIRO(token):
23     return token
24
25 t_ignore = " \t"
```

Código 3: Definição dos objetos-tokens

Além disso, o código também define regras para ignorar espaços em branco e tabulações (`t_ignore`) e para contabilizar o número de linhas (`t_newline`).

### 3.4 Captura de Erros

Num analisador léxico, é necessário capturar os erros que foram identificados para futuras depurações tanto do usuário quanto do desenvolvedor, para isso foi feita a função `t_error` que foi criada com a biblioteca PLY, que lida com erros de análise léxica (Código 4).

```
1 def t_error(token):
2
3     # file = token.lexer.filename
4     line = token.lineno
5     # column = define_column(token.lexer.backup_data, token.lexpos)
6     message = le.newError(checkKey, 'ERR-LEX-INV-CHAR', valor=token.value[0])
7     # print(f"[{file}]:[{line},{column}]: {message}.")
8     print(message)
9
10    token.lexer.skip(1)
11
12    # token.lexer.has_error = True
```

Código 4: Definição da função que captura erros

Quando um caractere inesperado é encontrado durante a análise, essa função é acionada, recebendo o token onde o erro ocorreu como parâmetro. Ela extrai o número da linha do token e cria uma mensagem de erro, indicando o tipo de erro e o valor do caractere que causou o problema. Em seguida, imprime essa mensagem e instrui o lexer a avançar para o próximo caractere, ignorando o caractere problemático. Isso permite que o analisador continue sua análise, lidando com erros de forma apropriada durante o processo.

### 3.5 Função Principal

A função *main* do programa inicia-se declarando duas variáveis globais, *checkTpp* e *checkKey*, para controle de erros do programa.

```

1 def main():
2     global checkTpp, checkKey
3
4     #Implementando a verificação do -k
5     for idx, arg in enumerate(argv):
6         if(arg == '-k'):
7             checkKey = True
8             aux = arg.split('.')
9             if aux[-1] == 'tpp':
10                 checkTpp = True
11                 idxTpp = idx
12
13     if(len(sys.argv) < 3):
14         raise TypeError(le.newError(checkKey, 'ERR-LEX-USE'))
15     if not checkTpp:
16         raise IOError(le.newError(checkKey, 'ERR-LEX-NOT-TPP'))
17     elif not os.path.exists(argv[idxTpp]):
18         raise IOError(le.newError(checkKey, 'ERR-LEX-FILE-NOT-EXISTS'))
19     else:
20         data = open(argv[idxTpp])
21
22         source_file = data.read()
23         lexer.input(source_file)
24
25         # Tokenize
26         while True:
27             tok = lexer.token()
28             if not tok:
29                 break          # No more input
30 #             print(tok)
31             print(tok.type)
32 #             print(tok.value)

```

Código 5: Definição da função Principal

Em seguida, o código implementa a verificação da opção *-k* passada pela linha de comando, iterando sobre os argumentos fornecidos. Se a opção *-k* estiver presente, a variável *checkKey* é configurada como verdadeira, tal informação é necessária para montar as mensagens de erro (Código 6). Além disso, é verificado se existe um arquivo com extensão *.tpp* entre os argumentos, e se sim, *checkTpp* é definido como verdadeiro e o índice desse arquivo é armazenado para referência posterior. Após essas verificações, o código valida se o número de argumentos passados é adequado e se o arquivo *.tpp* foi especificado. Se não for o caso, são lançadas exceções apropriadas, contendo mensagens de erro específicas. Se todas as verificações forem bem-sucedidas, o código abre o arquivo *.tpp*, lê seu conteúdo e o passa para o lexer. Em seguida, inicia-se o processo de tokenização, onde o lexer é chamado repetidamente para gerar tokens a partir do código fonte. Cada token gerado é impresso, exibindo seu tipo. Uma vez que não há mais tokens para serem processados, a execução do programa termina. Este trecho de código encapsula a lógica principal do programa, desde a inicialização até a análise léxica do arquivo fornecido.



### 3.6 Montagem da mensagem de erro

A classe `MyError` tem a finalidade de gerenciar e construir mensagens de erro com base em um conjunto predefinido de mensagens armazenadas em um arquivo de propriedades denominado *ErrorMessages.properties*.

```

1 import configparser
2 import io
3
4 config = None
5
6 class MyError():
7
8     def __init__(self, et):
9         self.config = configparser.RawConfigParser()
10        # Abre o arquivo com a codificação UTF-8
11        with io.open('ErrorMessages.properties', 'r', encoding='utf-8') as f:
12            self.config.read_file(f) # Use read_file para ler o arquivo
13        self.errorType = et
14    def newError(self, key, **data):
15        message = ''
16        #if optkey:
17        #    return(key)
18        if(key):
19            message = self.config.get(self.errorType, key)
20        if(data):
21            for key, value in data.items():
22                message = message + ", " f"{key}: {value}"
23        return message

```

Código 6: Montagem da mensagem de erro.

Ao ser inicializada, a classe lê este arquivo para acessar as mensagens de erro correspondentes a diferentes tipos de erro. O método `newError` é responsável por montar uma nova mensagem de erro, podendo receber uma chave de mensagem específica, um indicador booleano `optkey` para determinar se a mensagem de erro deve ser retornada sem substituição de argumentos, e dados adicionais que podem ser incluídos na mensagem de erro. Inicialmente, verifica-se se `optkey` é verdadeiro, e se for, retorna imediatamente a chave da mensagem de erro fornecida. Em seguida, obtém-se a mensagem de erro correspondente à chave do arquivo de configuração, se essa chave não for vazia. Se houver dados adicionais, eles são incluídos na mensagem de erro, formatados como pares "chave: valor". Finalmente, a mensagem de erro completa é retornada, incluindo a mensagem base e quaisquer dados adicionais, se existirem, ou apenas a chave da mensagem de erro se nenhum dado adicional for fornecido e `optkey` for verdadeiro. Assim, a classe `MyError` facilita a geração de mensagens de erro de forma estruturada e flexível para diferentes situações dentro de um programa.

```

1 python tpplex.py tests/teste-003.tpp -k

```

Código 7: Comando para execução

A saída esperada será:

```

1 INTEIRO
2 DOIS_PONTOS
3 ID
4 VIRGULA
5 ID
6 INTEIRO
7 ID

```

```
8 ABRE_PARENTESE
9 INTEIRO
10 DOIS_PONTOS
11 ID
12 FECHA_PARENTESE
13 FLUTUANTE
14 DOIS_PONTOS
15 ID
16 ID
17 ATRIBUICAO
18 NUM_PONTO_FLUTUANTE
19 INTEIRO
20 DOIS_PONTOS
21 ID
22 SE
23 ID
24 MAIOR
25 NUM_INTEIRO
26 ENTAO
27 SE
28 ID
29 MAIOR
30 NUM_INTEIRO
31 ENTAO
32 ID
33 ATRIBUICAO
34 NUM_INTEIRO
35 REPITA
36 REPITA
37 ID
38 ATRIBUICAO
39 ID
40 VEZES
41 ID
42 ATE
43 ID
44 IGUAL
45 NUM_INTEIRO
46 ID
47 ATRIBUICAO
48 ID
49 VEZES
50 ID
51 ATE
52 ID
53 IGUAL
54 NUM_INTEIRO
55 SENAO
56 ID
57 ATRIBUICAO
58 NUM_INTEIRO
59 FIM
60 FIM
61 INTEIRO
62 DOIS_PONTOS
63 ID
64 ID
65 ATRIBUICAO
66 ABRE_PARENTESE
67 NUM_INTEIRO
```

```
68 MAIS
69 NUM_INTEIRO
70 FECHA_PARENTESE
71 VEZES
72 NUM_INTEIRO
73 ID
74 ATRIBUICAO
75 NUM_INTEIRO
76 MAIS
77 NUM_INTEIRO
78 VEZES
79 NUM_INTEIRO
80 ID
81 ATRIBUICAO
82 MENOS
83 NUM_INTEIRO
84 MENOS
85 ABRE_PARENTESE
86 NUM_INTEIRO
87 MAIS
88 NUM_INTEIRO
89 FECHA_PARENTESE
90 ID
91 ATRIBUICAO
92 NUM_INTEIRO
93 MENOS
94 NUM_INTEIRO
95 ID
96 ATRIBUICAO
97 NUM_INTEIRO
98 VEZES
99 NUM_INTEIRO
100 FIM
```

Código 8: Saída Esperada

### 3.7 Considerações Sobre a Análise léxica

A implementação da análise léxica em Python utilizando a biblioteca Ply oferece uma abordagem eficiente e flexível para o reconhecimento de tokens em um código fonte. A definição dos tokens e expressões regulares permite uma análise precisa e robusta do código, garantindo que cada elemento léxico seja corretamente identificado e classificado. Além disso, a utilização de palavras reservadas da linguagem TPP e a definição de padrões específicos contribuem para uma análise ainda mais precisa e completa. No entanto, é importante ressaltar que a análise léxica é apenas o primeiro passo no processo de compilação, e que outros estágios, como análise sintática, análise semântica e geração de código, são igualmente importantes para o funcionamento adequado de um compilador. Portanto, a implementação da análise léxica deve ser complementada por esses outros estágios para garantir a correta tradução e execução do código fonte.

## 4 Análise Sintática

A análise sintática é uma fase crucial em compiladores, responsável por determinar a estrutura sintática de um programa. Essa estrutura é verificada com base nas regras gramaticais de uma linguagem de programação, que normalmente são definidas por gramáticas livres de contexto (GLC). Assim como as expressões regulares são usadas para descrever a estrutura léxica de uma

linguagem (*tokens*), as GLCs são usadas para descrever a estrutura sintática, permitindo lidar com estruturas recursivas e complexas, como o aninhamento de *if* e expressões aritméticas. O poder das GLCs supera o das expressões regulares devido à sua capacidade de representar sintaxes mais elaboradas.

#### 4.1 O Processo de Análise Sintática

O processo de análise sintática começa com o analisador sintático, também conhecido como *parser*, que transforma a sequência de *tokens* gerados pelo analisador léxico em uma árvore sintática. Essa árvore representa a estrutura hierárquica do programa e é essencial para as fases subsequentes do compilador. Existem duas abordagens principais para a análise sintática: a análise sintática ascendente e a análise sintática descendente. A análise ascendente constrói a árvore sintática das folhas para a raiz, começando pelos *tokens* e tentando identificar as regras gramaticais que formam a estrutura do programa. Já a análise descendente constrói a árvore da raiz para as folhas, iniciando com o símbolo inicial da gramática e tentando derivar a sequência de *tokens* conforme as regras gramaticais.

Diversos métodos podem ser usados para implementar essas abordagens, cada um com suas características e aplicabilidades. Por exemplo, a análise LL(1) é uma técnica de análise sintática descendente que requer uma gramática livre de contexto que seja livre de ambiguidade e fatoração à esquerda, permitindo a decisão da regra de produção a ser aplicada apenas com base no próximo token. Por outro lado, as técnicas de análise ascendente, como LR(1), LALR(1), e SLR(1), constroem a árvore sintática a partir dos tokens e são mais poderosas, pois conseguem lidar com um conjunto maior de gramáticas, incluindo aquelas que não são passíveis de serem analisadas por parsers LL(1).

Neste projeto, foi utilizada a técnica LALR(1) (Look-Ahead LR), que é uma versão otimizada do método LR(1). A escolha por LALR(1) deve-se ao seu balanço entre eficiência e poder de expressão, sendo capaz de lidar com a maioria das gramáticas usadas em linguagens de programação enquanto mantém o uso de memória relativamente baixo, o que é crucial para a implementação prática do compilador. A análise sintática LALR(1) permite que o *parser* resolva conflitos de redução e derivação em situações onde o LL(1) não seria capaz, sem o overhead associado às tabelas mais complexas do LR(1) completo.

#### 4.2 Gramáticas Livres de Contexto em Linguagens de Programação

As regras de uma **Gramática Livre de Contexto (GLC)** são especificadas sobre um alfabeto de símbolos, que representa os *tokens* da linguagem de programação. Esses *tokens* incluem palavras-chave, identificadores, operadores e outros símbolos significativos.

Por exemplo, para a linguagem T++, o alfabeto pode incluir *tokens* como:

```
{inteiro, principal, se, então, senão, fim, repita,  
até, leia, escreva, retorna, identificador, +, -, *, /, =, <, >, (, ), {, }, :=}
```

A especificação clara dessas regras permite que o analisador sintático distinga entre programas válidos e inválidos, utilizando a GLC para verificar se a sequência de *tokens* segue a estrutura sintática correta.

#### 4.3 Exemplo de Produções

Considere as seguintes produções para um segmento da linguagem T++:

```

Programa → inteiro principal() {Corpo}
Corpo → declaração Corpo | comando Corpo | ε
declaração → inteiro : identificador
comando → identificador Atribuição | retorna(expressão)
Atribuição → := expressão
expressão → Termo Operador Termo | Termo
Operador → + | - | * | /
Termo → inteiro | flutuante | identificador

```

#### 4.4 Exemplo de Análise Sintática

Vamos considerar o seguinte trecho de código em T++:

```

inteiro: a

inteiro principal()
    inteiro: b

    a := 10

    b := a

    retorna(b)
fim

```

Este trecho é analisado como segue:

1. inteiro: a é uma declaração de uma variável inteira chamada a.
2. inteiro principal() inicia a função principal do programa.
3. inteiro: b é uma declaração de uma variável inteira chamada b.
4. a := 10 é um comando de atribuição que atribui o valor 10 à variável a.
5. b := a é um comando de atribuição que copia o valor de a para a variável b.
6. retorna(b) indica que a função principal retorna o valor de b.
7. fim indica o final da função principal.

A sequência de tokens é analisada pela GLC para garantir que o código segue a estrutura sintática correta, baseada nas regras definidas. Se a sequência não corresponder a nenhuma derivação válida, o programa é considerado inválido.

#### 4.5 Produções e Geração de Linguagem

As produções em uma GLC funcionam como regras que definem a substituição de um não-terminal por uma cadeia de terminais e não-terminais. Por exemplo, na regra SomOvelha -> bée SomOvelha | bée, o não-terminal SomOvelha pode ser substituído por bée seguido por outro SomOvelha ou simplesmente por bée. Essa capacidade de definição recursiva é crucial para a geração de linguagens complexas. As GLCs atuam como geradores de cadeias de linguagens, começando pelo símbolo inicial e aplicando substituições conforme as produções, até se obter uma cadeia formada apenas por terminais. Um exemplo clássico é a gramática para expressões aritméticas simples: Expr -> (Expr) | Expr Op Expr | numero e Op -> + | - | \* | /, que mostra como expressões aritméticas podem ser geradas de maneira formal.

## 4.6 Derivações e Árvores Sintáticas

O processo de geração de uma cadeia por uma **Gramática Livre de Contexto (GLC)** envolve derivações, que são sequências de substituições que transformam o símbolo inicial em uma cadeia de terminais. A sequência de derivações pode ser representada por uma **árvore sintática**, onde o símbolo inicial é a raiz e cada produção representa uma ramificação na árvore.

Por exemplo, para a gramática  $G_1$  gerando a cadeia 000#111, as derivações seriam:

$$\begin{aligned} A &\Rightarrow 0A1 \\ &\Rightarrow 00A11 \\ &\Rightarrow 000A111 \\ &\Rightarrow 000B111 \\ &\Rightarrow 000\#111 \end{aligned}$$

O exemplo acima mostra como as regras de produção foram aplicadas sucessivamente para gerar a cadeia final. As árvores sintáticas são fundamentais para capturar a estrutura hierárquica do código e são uma das saídas mais importantes da fase de análise sintática. Elas permitem que os compiladores interpretem a relação entre diferentes partes do código, facilitando a detecção de erros sintáticos e a geração de código intermediário.

## 4.7 Linguagens Livres de Contexto

A linguagem gerada por uma **Gramática Livre de Contexto (GLC)** é o conjunto de todas as cadeias que podem ser derivadas a partir do símbolo inicial utilizando as regras de produção da gramática. Chamamos essa linguagem de **Linguagem Livre de Contexto (LLC)**.

Por exemplo, para a gramática  $G_1$ , a linguagem seria expressa como:

$$\{0^n\#1^n \mid n \geq 0\}$$

Essa notação representa todas as cadeias possíveis formadas por um número igual de zeros seguidos por um número igual de uns, onde  $n$  é um número inteiro não negativo. Esse conceito é essencial na construção de compiladores, pois a GLC define de maneira formal quais cadeias de caracteres são consideradas válidas na linguagem de programação alvo.

A capacidade de definir cadeias válidas formalmente permite que os compiladores reconheçam e processem o código fonte de maneira precisa e eficiente, garantindo que a sintaxe da linguagem seja seguida rigorosamente. As LLCs desempenham um papel fundamental na análise sintática, que é uma das etapas críticas na tradução de linguagens de programação.

## 4.8 Recursão em Gramáticas

Um aspecto fundamental das gramáticas livres de contexto é a capacidade de lidar com recursão, que é a repetição de estruturas dentro de uma linguagem. As regras gramaticais em BNF (Backus-Naur Form) podem expressar concatenação e escolha, mas não possuem uma operação de repetição equivalente ao operador  $*$  das expressões regulares. Em vez disso, a repetição é representada através da recursão.

Por exemplo, a regra

$$A \rightarrow Aa \mid a$$

ou

$$A \rightarrow aA \mid a$$

gera a linguagem

$$\{a^n \mid n \text{ inteiro } \geq 1\},$$

que representa cadeias de um ou mais caracteres 'a'. A recursão pode ser à esquerda ou à direita, dependendo de como o não-terminal é substituído em relação a outros elementos na regra.

## 4.9 Construção da Análise Sintática através do Ply e Yacc em python

Uma ferramenta amplamente utilizada para implementar parsers é o módulo yacc da biblioteca `PLY` em Python, que permite a definição de gramáticas e ações semânticas diretamente em funções Python. No yacc, as regras de gramática são definidas como strings na documentação das funções, e as operações semânticas são realizadas no corpo dessas funções. Ao combinar as regras de gramática e as ações semânticas, yacc gera automaticamente as tabelas de parsing necessárias para processar a linguagem especificada, tratando de detalhes complexos como precedência e associatividade de operadores. Dessa forma, yacc facilita a criação de parsers eficientes e personalizados para uma ampla variedade de linguagens e aplicações. Foi introduzido na aplicação o arquivo `tppparser.py`, neste arquivo foram adicionadas as funções que tratam a análise sintática de um arquivo `.tpp`.

## 4.10 Mecanismo de funcionamento do analisador sintático

O analisador sintático funciona com diversas funções, que agem como componentes essenciais para identificar e estruturar os elementos da linguagem de programação conforme as regras gramaticais definidas. Essas funções interpretam o código-fonte, verificando se ele segue a sintaxe correta e construindo uma árvore sintática abstrata (AST) que representa a estrutura hierárquica do código.

## 4.11 Funcionamento da aplicação de Análise Sintática

A aplicação realiza o *parsing* dos *tokens* gerados pelo `tpplex.py` (conforme discutido nas seções anteriores). A partir desses *tokens*, o programa verifica as entradas e constrói uma árvore sintática. A verificação é conduzida através da aplicação de regras gramaticais, onde existem dois tipos de funções: uma que recebe os *tokens* e confirma que a análise sintática foi concluída com sucesso, e outra que retorna um erro em caso de falha na análise. No código 1, a função `p_declaracao_variaveis` ilustra uma regra gramatical correta, enquanto a função `p_declaracao_variaveis_error` demonstra uma regra gramatical com erro.

```

1 def p_declaracao_variaveis(p):
2     """declaracao_variaveis : tipo DOIS_PONTOS lista_variaveis"""
3
4     pai = MyNode(name='declaracao_variaveis', type='DECLARACAO_VARIAVEIS')
5     p[0] = pai
6
7     p[1].parent = pai
8
9     filho = MyNode(name='DOIS_PONTOS', type='DOIS_PONTOS', parent=pai)
10    filho_sym = MyNode(name=p[2], type='SIMBOLO', parent=filho)
11    p[2] = filho
12
13    p[3].parent = pai

```

```

14
15 def p_declaracao_variaveis_error(p):
16     """declaracao_variaveis : error DOIS_PONTOS lista_variaveis
17                               | tipo error lista_variaveis
18                               | tipo DOIS_PONTOS error
19
20     """
21     error_type = 'ERR-SYN-LISTA-DECLARACAO-VARIAVEIS'
22     error_handler = MyError("ParserErrors")
23     # Cria a mensagem de erro
24     error_name = error_handler.newError(error_type)
25     print(error_name) # Imprime a mensagem de erro
26     pai = MyNode(name=error_name, type=error_type)
27     p[0] = pai

```

Código 9: Evidenciação das funções de análise de gramática e tratamento de erro sintático

As funções `p_declaracao_variaveis` e `p_declaracao_variaveis_error` desempenham papéis distintos no processo de parsing. Ambas utilizam o token `p` como argumento principal, que contém as informações dos tokens reconhecidos e gerados pelo analisador léxico.

## Funcionamento de `p_declaracao_variaveis`

A função `p_declaracao_variaveis` é responsável por reconhecer e construir a árvore sintática para a declaração de variáveis sem erros. Quando o token `p` é recebido:

- **Verificação dos Tokens:** A função verifica se os tokens correspondem à regra sintática definida, que, neste caso, é a sequência de um tipo, seguido por um símbolo `DOIS_PONTOS`, e uma `lista_variaveis`.
- **Construção da Árvore Sintática:** Se a verificação for bem-sucedida, a função começa a construir a árvore sintática:
  - Um nó principal (`pai`) é criado para representar a declaração de variáveis.
  - Cada componente da regra (tipo, `DOIS_PONTOS`, lista de variáveis) é associado ao nó principal como filhos.
  - Para o token `DOIS_PONTOS`, um nó filho específico é criado (`filho`), ao qual é associado um símbolo (`filho_sym`), representando o token `DOIS_PONTOS`.
- **Resultado:** Se tudo correr bem, o token `p[0]` é associado ao nó principal (`pai`), e a árvore é gerada no terminal do sistema operacional e também, através da biblioteca *Anytree*, Meyer (2024), é gerado um arquivo de imagem PNG da árvore completa, vide Figura 1 e Figura 2 (ao final do documento), respectivamente.

## Tratamento de Erros com `p_declaracao_variaveis_error`

A função `p_declaracao_variaveis_error` é responsável por identificar e tratar erros sintáticos que ocorrem durante a análise da declaração de variáveis. O funcionamento é o seguinte:

- **Identificação do Erro:** A função verifica se houve algum erro durante a correspondência dos tokens. Existem três possíveis pontos onde o erro pode ocorrer:
  - No início da declaração (`error DOIS_PONTOS lista_variaveis`).
  - Após o tipo de variável (`tipo error lista_variaveis`).
  - Antes da lista de variáveis (`tipo DOIS_PONTOS error`).
- **Manipulação do Erro:** Quando um erro é detectado:
  - Um tipo de erro é identificado (`ERR-SYN-LISTA-DECLARACAO-VARIAVEIS`).
  - Um gerenciador de erros (`error_handler`) é utilizado para gerar uma mensagem de erro correspondente. Essa mensagem é carregada até o nó principal da árvore.



- **Propagação e Exibição do Erro:** O erro é então propagado através dos nós da árvore sintática. Se o erro não puder ser tratado, o terminal exibirá uma mensagem de erro específica, derivada do token problemático, conforme configurado no arquivo `ErrorMessages.properties`. Por exemplo, o token "ERR-SYN-LISTA-ARGUMENTOS" geraria uma mensagem de erro correspondente. O arquivo `ErrorMessages.properties` possui as relações de chave e valor entre as mensagens de erro e os *tokens*:

```

1 [ParserErrors]
2 ERR-SYN-USE=Uso: python tppparser.py file.tpp
3 ERR-SYN-NOT-TPP = Não é um arquivo .tpp.
4 ERR-SYN-FILE-NOT-EXISTS=Arquivo .tpp não existe.
5 WAR-SYN-NOT-GEN-SYN-TREE=Não foi possível gerar a Árvore Sintática.
6 ERR-SYN-CHAMADA-FUNCAO-LISTA-PARAM=Erro na lista de parâmetros.
7 ERR-SYN-LISTA-ARGUMENTOS=Erro na lista de Argumentos.
8 ERR-SYN-LISTA-DECLARACOES=Erro na lista de Declarações.
9 ERR-SYN-LISTA-COMANDOS=Erro na lista de Comandos.
10 ERR-SYN-DECLARACAO-VAR=Erro na Declaração de Variável.
11 ERR-SYN-LISTA-DECLARACAO-VARIAVEIS=Erro na lista de Declaração de Variáveis.
12 ERR-SYN-LISTA-VARIAVEIS=Erro na lista de Variáveis.
13 ERR-SYN-VARIAVEL=Erro na Variável.
14 ERR-SYN-INDICE=Erro no Índice.
15 ERR-SYN-DECLARACAO-FUNCAO=Erro na Declaração de Função.
16 ERR-SYN-CABECALHO=Erro no Cabeçalho.
17 ERR-SYN-LISTA-PARAMETROS=Erro na lista de Parâmetros.
18 ERR-SYN-PARAMETRO=Erro no Parâmetro.
19 ERR-SYN-CORPO=Erro no Corpo.
20 ERR-SYN-ACAO=Erro na Ação.
21 ERR-SYN-SE=Erro no Se.
22 ERR-SYN-REPITA=Erro no Repita.
23 ERR-SYN-ATRIBUICAO=Erro na Atribuição.
24 ERR-SYN-LEIA=Erro no Leia.
25 ERR-SYN-ESCREVA=Erro no Escreva.
26 ERR-SYN-RETORNA=Erro no Retorna.
27 ERR-SYN-EXPRESSAO-LOGICA=Erro na Expressão Lógica.
28 ERR-SYN-EXPRESSAO-SIMPLES=Erro na Expressão Simples.
29 ERR-SYN-EXPRESSAO-ADITIVA=Erro na Expressão Aditiva.
30 ERR-SYN-EXPRESSAO-MULTIPLICATIVA=Erro na Expressão Multiplicativa.
31 ERR-SYN-EXPRESSAO-UNARIA=Erro na Expressão Unária.
32 ERR-SYN-FATOR=Erro no Fator.
33 ERR-SYN-CHAMADA-FUNCAO=Erro na Chamada de Função.

```

Código 10: `ErrorMessages.properties`

## 4.12 Entrada e saída da análise sintática

para executar a aplicação, abra o terminal da sua máquina e execute o comando:

```

1 python tppparser.py tests/teste-1.tpp

```

Código 11: Comando para execução da análise sintática

a saída esperada será como vista nas Figuras 1 e 2.

## 5 Análise Semântica

A análise semântica é a terceira fase no processo de compilação. Ela ocorre após a análise sintática e visa garantir que o código fonte seja semanticamente válido, isto é, que ele faça sentido dentro do contexto da linguagem de programação utilizada.



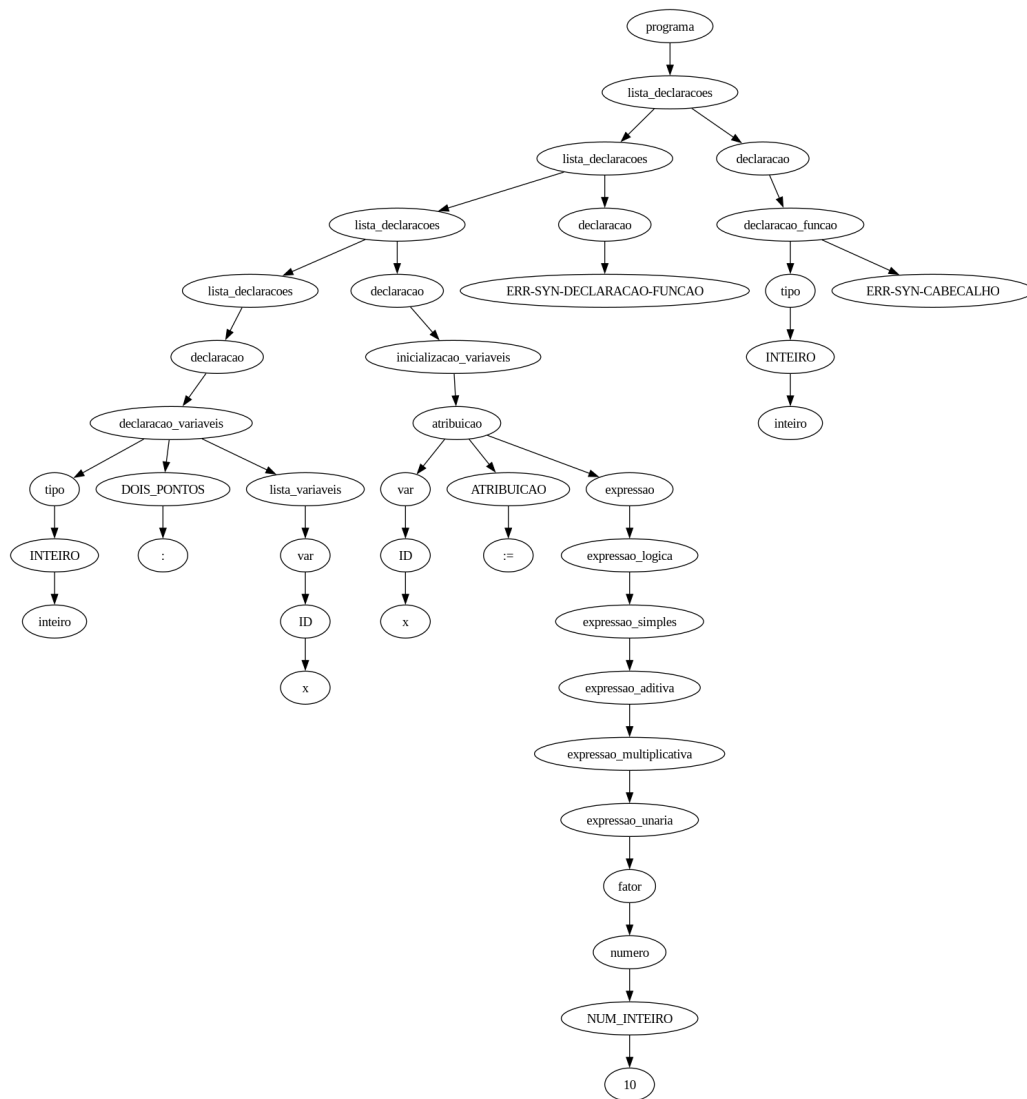


Figura 2: Imagem da árvore gerada do arquivo Erro-008.tpp.

### 5.2.4 Checagem de Funções e Parâmetros

A análise semântica também verifica se as funções são chamadas com o número correto e tipos de parâmetros. Por exemplo, se uma função é definida para aceitar dois parâmetros inteiros e é chamada com uma string e um inteiro, a análise semântica deve identificar esse erro.

### 5.2.5 Verificação de Coerência de Estruturas

As estruturas de controle, como loops e condicionais, devem ser usadas de forma coerente. Por exemplo, uma expressão condicional deve avaliar para um tipo booleano. Caso contrário, isso deve ser sinalizado como um erro.

### 5.2.6 Análise de Atribuições e Operações

Esta etapa verifica se as operações e atribuições são realizadas de acordo com as regras da linguagem. Variáveis imutáveis devem ser tratadas corretamente, e atribuições devem respeitar as propriedades dos tipos de dados.

## 5.3 Tabela de Símbolos

A tabela de símbolos é uma estrutura de dados essencial utilizada na análise semântica. Ela armazena informações sobre variáveis, funções e seus atributos. Criada e gerida pelo analisador léxico e o parser, a tabela de símbolos fornece a base para a análise semântica, permitindo a verificação das regras de tipo e escopo, entre outras.

## 5.4 Regras Semânticas

As regras semânticas da linguagem TPP definem restrições e verificações que garantem a correção e coerência dos programas. Elas cobrem diversos aspectos da programação, desde a declaração e uso de funções e variáveis até a manipulação de tipos e índices de arranjos. A tabela abaixo resume as principais regras semânticas aplicáveis:

Categoria	Tipo
<b>Funções e Procedimentos</b>	
Sem Função Principal	Erro
Erro na Declaração de Função	Erro
Identificador de Função (Poucos Parâmetros)	Erro
Identificador de Função (Muitos Parâmetros)	Erro
Função Declarada e Não Utilizada	Aviso
Retorno de Função (Tipo Incorreto)	Erro
Chamada da Função Principal	Erro
Chamada Recursiva da Função Principal	Aviso
<b>Variáveis</b>	
Variável Não Declarada	Erro
Variável Declarada e Não Inicializada	Aviso
Variável Declarada e Não Utilizada	Aviso
Variável Declarada Mais de Uma Vez	Aviso
<b>Atribuição de Tipos Distintos e Coerções Implícitas</b>	

Categoria	Tipo
Atribuição de Tipos Distintos (Variáveis)	Aviso
Atribuição de Tipos Distintos (Números)	Aviso
Atribuição de Tipos Distintos (Expressões)	Aviso
Atribuição de Tipos Distintos (Retorno de Função)	Aviso
Atribuição de Tipos Distintos (Argumentos de Função)	Aviso
<b>Arranjos</b>	
Índice de Array Não Inteiro	Erro
Índice de Array Fora do Intervalo	Erro

Tabela 3: Tabela de Regras Semânticas

#### 5.4.1 Codificação de chave e valor das regras semânticas

Para que o analisador semântico funcione e evidencie as mensagens de erro na execução, é necessário alterar o arquivo das propriedades dos erros (*ErrorMessages.properties*), após a alteração, o arquivo ficou como se segue:

```

1 [MainErrors]
2 ERR-MAIN-USE=Uso: python tppmain.py file.tpp
3 ERR-MAIN-NOT-TPP=Não é um arquivo .tpp.
4 ERR-MAIN-FILE-NOT-EXISTS=Arquivo .tpp não existe.
5 ERR-MAIN-LEX-ERR=Erro na Análise Léxica.
6 ERR-MAIN-SYN-ERR=Erro na Análise Sintática.
7 ERR-MAIN-SEM-ERR=Erro na Análise Semântica.
8
9 [SemaErrors]
10 ERR-SEM-MAIN-NOT-DECL=Função 'principal' não declarada.
11 ERR-SEM-VAR-NOT-DECL=Variável '{}' não declarada.
12 WAR-SEM-VAR-DECL-NOT-INIT=Variável '{}' declarada e não inicializada.
13 WAR-SEM-VAR-DECL-NOT-USED=Variável '{}' declarada e não utilizada.
14 ERR-SEM-ARRAY-INDEX-NOT-INT=Índice de array '{}' não 'inteiro'.
15 ERR-SEM-ARRAY-INDEX-OUT-OF-RANGE=Índice de array 'A' fora do intervalo (out of
   range).
16 ERR-SEM-FUNC-RET-TYPE-ERROR=Função '{}' deveria retornar '{}', mas retorna '{}'.
17 ERR-SEM-CALL-FUNC-NOT-DECL=Chamada à função '{}' que não foi declarada.
18 ERR-SEM-CALL-FUNC-WITH-FEW-ARGS=Chamada à função '{}' com número de parâmetros
   menor que o declarado.
19 ERR-SEM-CALL-FUNC-WITH-MANY-ARGS=Chamada à função '{}' com número de parâmetros
   maior que o declarado.
20 ERR-SEM-CALL-FUNC-MAIN-NOT-ALLOWED=Chamada à função 'principal' não permitida.
21 WAR-SEM-VAR-DECL-PREV=Variável '{}' declarada anteriormente com o tipo '{}'.
22 WAR-SEM-VAR-DECL-INIT-NOT-USED=Variável '{}' declarada e inicializada, mas não
   utilizada.
23 WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-VAR=Atribuição de tipos distintos. Coerção
   implícita do valor de '{}' do tipo '{}' para '{}' que é '{}'.
24 WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-EXP=Atribuição de tipos distintos. Coerção
   implícita do valor de '{}' do tipo '{}' para '{}' que é '{}'.
25 WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-RET-VAL=Atribuição de tipos distintos.
   Coerção implícita do valor retornado por '{}' do tipo '{}' para '{}' que é '
   {}'.
26 WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-NUM=Atribuição de tipos distintos. Coerção
   implícita do valor atribuído '{}' do tipo '{}' para '{}' que é '{}'.
27 WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-FUNC-ARG=Chamada à função '{}' com Coerção
   implícita do valor do argumento tipo '{}' diferente do parâmetro declarado '
   {}'.
28 WAR-SEM-FUNC-DECL-NOT-USED=Função '{}' declarada, mas não utilizada.

```

29 WAR-SEM-CALL-REC-FUNC-MAIN=Chamada recursiva para 'principal'.

Código 12: Erros semânticos na ErrorMessage.properties

Note que foi criada também a aba de *MainErros*, tal aba foi criada pois para a seguir os padrões de um compilador real, é necessário que o *parser* crie a tabela e chame o analisador seântico, portanto uma main foi criada para guiar os arquivos, segue o novo arquivo chamado *main.py*, onde ele é o arquivo que executa os outros.

```

1 import tppparser
2 import tppsema
3 from myerror import MyError
4 from sys import argv
5 import os
6
7 # Inicializa o manipulador de erros com o arquivo de erros adequado
8 error_handler = MyError('MainErrors')
9
10 if __name__ == "__main__":
11     numParameters = len(argv) # Número de parâmetros
12
13     if numParameters != 2:
14         error = "Número de parâmetros Inválido, verifique a sintaxe. "
15         if numParameters < 2:
16             error += "Envie um arquivo .tpp."
17             raise IOError(error_handler.newError(False, 'ERR-MAIN-USE'))
18             raise IOError(error_handler.newError(False, 'ERR-MAIN-USE'))
19
20     aux = argv[1].split('.')
21     if aux[-1] != 'tpp':
22         raise IOError(error_handler.newError(False, 'ERR-MAIN-NOT-TPP'))
23     elif not os.path.exists(argv[1]):
24         raise IOError(error_handler.newError(False, 'ERR-MAIN-FILE-NOT-EXISTS'))
25     else:
26         try:
27             tppparser.main()
28         except Exception as e:
29             raise IOError(error_handler.newError(False, 'ERR-MAIN-SYN-ERR'))
30
31     if tppparser.root is not None and tppparser.root.children != ():
32         # Análise semântica
33         try:
34             tppsema.root = tppparser.root
35             tppsema.checkRules() # Realiza a análise semântica
36         except Exception as e:
37             raise IOError(error_handler.newError(False, 'ERR-MAIN-SEM-ERR'))
38             tppsema.podaArvore() # Realiza a poda da árvore sintática
39     else:
40         raise IOError(error_handler.newError(False, 'ERR-MAIN-SYN-ERR'))

```

Código 13: Arquivo Main.py

### 5.4.2 Explicação do Código do novo Arquivo main.py

O script *main.py* serve como o ponto de entrada para a execução de um compilador ou interpretador para arquivos no formato TPP. A primeira parte do código lida com importações e inicializações necessárias para a execução. Ele importa módulos essenciais, como *tppparser* e *tppsema*, que são responsáveis, respectivamente, pela análise sintática e semântica do código fonte.

Além disso, o script importa a classe `MyError` para gerenciar mensagens de erro e os módulos `sys` e `os` para manipulação de argumentos e verificação de arquivos no sistema operacional.

Na inicialização, o script cria uma instância do manipulador de erros, `error_handler`, usando um arquivo de propriedades chamado `MainErrors`. Esse arquivo contém as mensagens de erro que serão usadas para informar o usuário sobre problemas encontrados durante a execução do programa.

O bloco principal do código é executado quando o script é chamado diretamente. Primeiramente, ele verifica o número de argumentos passados para o script. Se o número de argumentos estiver incorreto (diferente de dois), o script gera uma mensagem de erro apropriada. Se o número de argumentos for menor que o esperado, o erro indica que o arquivo de entrada está faltando; se for maior, o erro aponta para uma sintaxe inválida na chamada do script.

Depois disso, o script verifica se o arquivo passado como argumento tem a extensão `.tpp` e se o arquivo realmente existe no sistema. Se a extensão estiver errada ou se o arquivo não for encontrado, são geradas mensagens de erro correspondentes. Se todas essas verificações passarem, o script tenta executar a análise sintática chamando a função `tppparser.main()`. Caso ocorra uma exceção durante essa análise, um erro de análise sintática é reportado.

Após a análise sintática, o script verifica se a árvore sintática foi corretamente gerada (isto é, se não está vazia). Se a árvore não for vazia, o script prossegue para a análise semântica. Ele configura a raiz da árvore sintática no módulo de análise semântica e tenta realizar a verificação de regras semânticas com `tppsema.checkRules()`. Se houver qualquer problema durante a análise semântica, um erro de análise semântica é gerado. Após a execução, o script realiza a poda da árvore sintática com `tppsema.podaArvore()`, o que pode ser necessário para remover informações desnecessárias ou otimizar a estrutura da árvore.

## 6 Funcionamento Geral do Código

O código da análise semântica pode ser dividido em várias etapas principais, que são detalhadas a seguir.

Primeiramente, o processo de compilação inicia com a análise léxica e sintática do código-fonte. A análise léxica é responsável por transformar o código-fonte em uma sequência de tokens, que são as menores unidades de significado, como identificadores, operadores e números. Esses tokens são identificados e classificados, permitindo que o compilador compreenda a estrutura básica do código.

Em seguida, a análise sintática verifica a estrutura do código em relação à gramática da linguagem TPP. Durante essa fase, é gerada uma árvore sintática, que representa a organização hierárquica das expressões e instruções no código. A árvore sintática é crucial para a compreensão da estrutura do programa e é utilizada em etapas posteriores para garantir que o código esteja correto e siga as regras da linguagem.

Após a análise sintática, a construção da Tabela de Símbolos é iniciada. Esta tabela é fundamental para o processamento semântico do código. Ela armazena informações detalhadas sobre variáveis e funções, incluindo seus nomes, tipos e escopos. A Tabela de Símbolos é atualizada à medida que o compilador analisa o código, garantindo que todas as variáveis e funções sejam corretamente declaradas e utilizadas.

Durante a análise semântica, o compilador verifica se as regras semânticas são seguidas. Isso inclui a verificação de declarações de funções, tipos de retorno e a coerção de tipos em atribuições e chamadas de funções. Mensagens de erro e avisos são gerados para qualquer violação dessas regras, garantindo que o código esteja correto e siga as normas estabelecidas pela linguagem TPP.

Por fim, com base na árvore sintática e na Tabela de Símbolos, o código é preparado para a geração de código intermediário. Esta etapa envolve a conversão das instruções em um formato

que pode ser otimizado e eventualmente traduzido para o código de máquina executável.

Essas etapas asseguram que o código-fonte em TPP seja corretamente analisado e convertido em um formato que possa ser executado, respeitando todas as regras semânticas e estruturais da linguagem.

## 6.1 Funções de Análise Semântica e Geração de Tabelas

A seguir, serão detalhadas algumas das funções responsáveis pela geração da tabela de símbolos e pela verificação semântica no código.

```

1 # Gera a tabela de símbolos a partir da árvore sintática
2 # Tal tabela foi criada a partir do slide do professor Rogério Gonçalves
3 # descrita no relatório anexado à este programa
4 def tabelaDeSimbolos():
5     resultados = findall_by_attr(root, "declaracao")
6     variaveis = []
7     for p in resultados:
8         item = [noh for pre, fill, noh in RenderTree(p)]
9         if item[1].name == "declaracao_variaveis":
10             variavel = processaVariavel(primeiroNo=item[1], escopo="global")
11             if declaracaoVariavel(tabelaSimbolica=variaveis, name=variavel['name'], escopo='global'):
12                 tipoVariavel = buscaTipo(tabelaSimbolica=variaveis, name=variavel['name'], escopo='global')
13                 print(error_handler.newError(False, 'WAR-SEM-VAR-DECL-PREV').format(variavel['name'], tipoVariavel))
14             else:
15                 variaveis.append(variavel)
16         elif item[1].name == "declaracao_funcao":
17             if item[2].name == "tipo":
18                 name = item[7].name
19                 token = item[6].name
20                 type = item[4].name
21                 line = item[4].line
22             else:
23                 name = item[4].name
24                 token = item[3].name
25                 type = 'vazio'
26                 line = item[4].line
27
28             variavel = {
29                 "tipoDeclaracao": 'func',
30                 "type": type,
31                 "line": line,
32                 "used": "S" if name == "principal" else "N",
33                 "dimension": 0,
34                 "tamanhoDimensao1": 1,
35                 "token": token,
36                 "name": name,
37                 "escopo": "global",
38                 "tamanhoDimensao2": 0,
39                 "parametros": declaracaoParams(item)
40             }
41             if declaracaoVariavel(tabelaSimbolica=variaveis, name=name, escopo='global'):
42                 tipoVariavel = buscaTipo(tabelaSimbolica=variaveis, name=name, escopo='global')
43                 print(error_handler.newError(False, 'WAR-SEM-FUNC-DECL-PREV').format(name, tipoVariavel))

```



```

44         else:
45             variaveis.append(variavel)
46             declaracaoFunc(primeiroNo=item[1], escopo=name, tabelaSimbolica=
variaveis)
47     return variaveis

```

Código 14: Geração da Tabela de Símbolos

A função `tabelaDeSimbolos` é responsável por criar a tabela de símbolos a partir da árvore sintática gerada pelo parser. Ela procura por declarações de variáveis e funções na árvore e as adiciona à tabela de símbolos. Para cada declaração de variável encontrada, a função `processaVariavel` é chamada para processar a variável e verificar se ela já foi declarada anteriormente, gerando um aviso caso necessário. Se a variável ou função já estiver declarada, um aviso é impresso. Caso contrário, a nova variável ou função é adicionada à tabela.

```

1  # Processa a declaração de uma variável, determinando suas propriedades
2  def processaVariavel(primeiroNo, escopo):
3      # Inicializa variáveis para armazenar as dimensões
4      unidimensional = 1
5      bidimensional = 0
6      dimensao = 0
7
8      # Cria uma lista com todos os nós da árvore
9      renderNodeTree = [noh for pre, fill, noh in RenderTree(primeiroNo)]
10
11     # Inicializa variáveis para armazenar propriedades da variável
12     tipo = None
13     line = None
14     token = None
15     name = None
16
17     for i, node in enumerate(renderNodeTree):
18         if node.name == 'tipo':
19             tipo = renderNodeTree[i + 2].name
20             line = renderNodeTree[i + 2].line
21         elif node.name == 'ID':
22             token = node.name
23             name = renderNodeTree[i + 1].name
24         elif node.name == 'fecha_colchete':
25             dimensao += 1
26
27         # Verifica se o índice é um número ponto flutuante
28         if renderNodeTree[i - 2].name == 'NUM_PONTO_FLUTUANTE':
29             if not variavelComErro(name, escopo):
30                 adicionaErroVariavel(name, escopo)
31                 print(error_handler.newError(False, 'ERR-SEM-ARRAY-INDEX-NOT
-INT').format(name))
32
33             indice = renderNodeTree[i - 1].name
34             if dimensao == 2:
35                 bidimensional = indice
36             else:
37                 unidimensional = indice
38
39     # Cria e retorna o dicionário com as propriedades da variável
40     variavel = {
41         'tipoDeclaracao': 'var',
42         'type': tipo,
43         'line': line,
44         'token': token,

```

```

45         'name': name,
46         'escopo': escopo,
47         'init': 'N',
48         'usado': 'N',
49         'dimensao': dimensao,
50         'tamanhoDimensao1': unidimensional,
51         'tamanhoDimensao2': bidimensional,
52         'errors': 0
53     }
54
55     return variavel

```

Código 15: Processamento da Declaração de Variáveis

A função `processaVariavel` lida com a declaração de variáveis, processando suas propriedades, como tipo, nome, e dimensões. Ela constrói um dicionário com as informações da variável e verifica erros associados, como o uso de índices de array inválidos. Se algum erro for encontrado, uma mensagem de erro é impressa.

```

1  # Gera a lista de parâmetros de uma função a partir da árvore sintática
2  def declaracaoParams(primeiroNo):
3      parametros = []
4      for item in primeiroNo:
5          if item.name == 'cabecalho':
6              # Obtém a lista de parâmetros
7              listaParametros = findall_by_attr(item.children[2], "parametro")
8              for parametro in listaParametros:
9                  # Extrai o tipo e o nome do parâmetro
10                 tipo = parametro.children[0].children[0].children[0].name
11                 nome = parametro.children[2].children[0].name
12                 parametros.append({
13                     'type': tipo,
14                     'name': nome
15                 })
16     return parametros

```

Código 16: Geração da Lista de Parâmetros

A função `parametersDeclaration` extrai e retorna uma lista de parâmetros de uma função a partir da árvore sintática. Ela localiza a seção de cabeçalho da função e coleta os parâmetros encontrados, registrando o tipo e o nome de cada parâmetro.

```

1  # Processa a declaração de variáveis dentro do escopo de uma função
2  def declaracaoFunc(primeiroNo, escopo, tabelaSimbolica):
3      variaveisDeclaradas = findall_by_attr(primeiroNo, "declaracao_variaveis")
4
5      for variavelNoh in variaveisDeclaradas:
6          variavel = processaVariavel(primeiroNo=variavelNoh, escopo=escopo)
7
8          if declaracaoVariavel(tabelaSimbolica=tabelaSimbolica, name=variavel['name'], escopo=escopo):
9              tipoVariavel = buscaTipo(tabelaSimbolica=tabelaSimbolica, name=variavel['name'], escopo=escopo)
10             print(error_handler.newError(False, 'WAR-SEM-VAR-DECL-PREV').format(
11                 variavel['name'], tipoVariavel))
12         else:
13             tabelaSimbolica.append(variavel)

```

Código 17: Processamento da Declaração de Funções

A função `declaracaoFunc` processa a declaração de variáveis dentro do escopo de uma função. Ela utiliza a função `processaVariavel` para processar cada variável e verifica se a variável já foi declarada no escopo atual, adicionando-a à tabela de símbolos se ainda não estiver presente.

Essas funções são essenciais para a verificação e gestão semântica do código, garantindo que as declarações de variáveis e funções estejam corretas e conforme as regras do compilador. Elas ajudam a construir e manter a tabela de símbolos, que é fundamental para a análise e execução correta do código.

## 6.2 Verificação de Coerção de Tipos e Inicialização de Variáveis

A seguir, serão detalhadas as funções adicionais que garantem a coerção de tipos e a correta inicialização das variáveis.

```

1 # Verifica coerções de tipos em atribuições e operações, emitindo avisos se
  necessário
2 def verificarCoercao(tabelaSimbolica, name, escopo, noh):
3     type = None
4     fatores = buscaFator(noh, escopo, tabelaSimbolica)
5     for i in range(len(tabelaSimbolica)):
6         type = None
7         try:
8             parametros = tabelaSimbolica[i]['parametros']
9         except KeyError:
10            parametros = None
11
12     # Verifica o tipo da variável ou parâmetro na tabela de símbolos
13     if tabelaSimbolica[i]['name'] == name and (tabelaSimbolica[i]['escopo']
14 == 'global' or tabelaSimbolica[i]['escopo'] == escopo):
15         type = tabelaSimbolica[i]['type']
16         elif parametros is not None and len(parametros) > 0:
17             for parametro in parametros:
18                 if parametro['name'] == name:
19                     type = parametro['type']
20
21     if type is not None:
22         # Se a expressão contém um único fator, verifica se o tipo precisa
23         de coerção
24         if len(fatores) == 1:
25             tipoFator = fatores[0]['type']
26             if tipoFator != type:
27                 valorFator = fatores[0]['value']
28                 fator = fatores[0]['factor']
29                 if fator == 'var':
30                     print(error_handler.newError(False, 'WAR-SEM-ATR-DIFF-
31 TYPES-IMP-COERC-OF-VAR').format(valorFator, tipoFator, name, type))
32                 elif fator == 'func':
33                     print(error_handler.newError(False, 'WAR-SEM-ATR-DIFF-
34 TYPES-IMP-COERC-OF-RET-VAL').format(valorFator, tipoFator, name, type))
35                 else:
36                     print(error_handler.newError(False, 'WAR-SEM-ATR-DIFF-
37 TYPES-IMP-COERC-OF-NUM').format(valorFator, tipoFator, name, type))
38             else:
39                 # Se a expressão contém múltiplos fatores, determina o tipo
40                 predominante
41                 tipoFator = buscaTipoFator(fatores, type)
42                 if tipoFator != type:
43                     valorFator = 'expressao'
44                     print(error_handler.newError(False, 'WAR-SEM-ATR-DIFF-TYPES-

```

```
IMP-COERC-OF-EXP').format(valorFator, tipoFator, name, type))
```

Código 18: Verificação de Coerção de Tipos

A função `verificarCoercacao` é utilizada para verificar se há coerção de tipos necessária em atribuições e operações. Ela compara o tipo da variável ou parâmetro com o tipo da expressão ou fator utilizado. Se houver incompatibilidade de tipos, são gerados avisos de coerção. Caso a expressão contenha múltiplos fatores, a função também determina o tipo predominante da expressão.

```
1 # Inicializa a variável na tabela de símbolos e verifica coerção de tipos
2 def inicializarVariavel(tabelaSimbolica, name, escopo, noh):
3     if declaracaoVariavel(tabelaSimbolica=tabelaSimbolica, name=name, escopo=
4         escopo):
5         verificarCoercacao(tabelaSimbolica=tabelaSimbolica, name=name, escopo=
6             escopo, noh=noh)
7         for i in range(len(tabelaSimbolica)):
8             if tabelaSimbolica[i]['name'] == name and (tabelaSimbolica[i]['
9                 escopo'] == 'global' or tabelaSimbolica[i]['escopo'] == escopo):
10                 tabelaSimbolica[i]['init'] = 'Y' # Marca a variável como
11                 inicializada
12             else:
13                 # Se a variável não está declarada, verifica se não é uma chamada de
14                 função antes de reportar erro
15                 resultados = findall_by_attr(noh, 'chamada_funcao')
16                 if not resultados and not variavelComErro(name, escopo):
17                     adicionaErroVariavel(name, escopo)
18                 print(error_handler.newError(False, 'ERR-SEM-VAR-NOT-DECL').format(
19                     name))
```

Código 19: Inicialização de Variáveis

A função `inicializarVariavel` é responsável por inicializar uma variável na tabela de símbolos e verificar a coerção de tipos. Ela marca a variável como inicializada e, se a variável não estiver declarada, verifica se a referência é de uma chamada de função. Caso contrário, reporta um erro de variável não declarada.

```
1 # Verifica todas as variáveis em uso no código, identificando e inicializando ou
2   marcando-as como usadas
3 def verificarVariavel(tabelaSimbolica):
4     resultados = findall_by_attr(root, "acao")
5     for p in resultados:
6         renderNodeTree = [noh for pre, fill, noh in RenderTree(p)]
7         for primeiroNo in renderNodeTree:
8             renderNode1Tree = [noh for pre, fill, noh in RenderTree(primeiroNo)]
9             if primeiroNo.name == 'expressao':
10                 if renderNode1Tree[1].name == 'atribuicao':
11                     escopo = buscaEscopo(primeiroNo)
12                     name = renderNode1Tree[4].name
13                     inicializarVariavel(tabelaSimbolica=tabelaSimbolica, name=
14                         name, escopo=escopo, noh=primeiroNo)
15                 else:
16                     for indice in range(len(renderNode1Tree)):
17                         if renderNode1Tree[indice].name == 'ID':
18                             escopo = buscaEscopo(primeiroNo)
19                             name = renderNode1Tree[indice+1].name
20                             variavelUsada(tabelaSimbolica=tabelaSimbolica, name=
21                                 name, escopo=escopo, noh=primeiroNo)
22                     elif primeiroNo.name == 'leia':
23                         for indice in range(len(renderNode1Tree)):
24                             if renderNode1Tree[indice].name == 'ID':
```

```

22         escopo = buscaEscopo(primeiroNo)
23         name = renderNode1Tree[indice+1].name
24         inicializarVariavel(tabelaSimbolica=tabelaSimbolica,
name=name, escopo=escopo, noh=primeiroNo)
25         elif primeiroNo.name in ['se', 'repita', 'escreva', 'retorna']:
26             for indice in range(len(renderNode1Tree)):
27                 if renderNode1Tree[indice].name == 'ID':
28                     escopo = buscaEscopo(primeiroNo)
29                     name = renderNode1Tree[indice+1].name
30                     variavelUsada(tabelaSimbolica=tabelaSimbolica, name=name
, escopo=escopo, noh=primeiroNo)
31         elif primeiroNo.name == 'chamada_funcao':
32             escopo = buscaEscopo(primeiroNo)
33             name = primeiroNo.children[0].children[0].name
34             variavelUsada(tabelaSimbolica=tabelaSimbolica, name=name, escopo
=escopo, noh=primeiroNo)

```

Código 20: Verificação de Variáveis em Uso

A função `verificarVariavel` percorre todas as ações no código e verifica as variáveis em uso. Ela chama `inicializarVariavel` para inicializar variáveis em atribuições e para entradas. Para variáveis usadas em expressões, ela utiliza `variavelUsada` para marcar a variável como usada.

### 6.3 Poda de Inicializações e Expressões

Além da verificação e inicialização, o código inclui funções para podar a árvore sintática, simplificando a estrutura para análises subsequentes.

```

1 # Função para podar a inicialização de variáveis
2 def podaInicilizacaoVariavel(tree):
3     podaInicilizacao(tree.children[0])
4
5 def podaInicilizacao(tree):
6     descritorArvore = ()
7     descritorArvore += (podaVariavel(tree.children[0]),)
8     descritorArvore += (tree.children[1].children[0],)
9
10    # Poda da expressão de inicialização
11    tree.children[2].children = podaExpressao(tree.children[2])
12    descritorArvore += (tree.children[2],)
13
14    # Atualiza a árvore com a inicialização podada
15    tree.children = descritorArvore
16    return tree

```

Código 21: Poda da Inicialização de Variáveis

As funções `podaInicilizacaoVariavel` e `podaInicilizacao` são responsáveis por simplificar a estrutura de inicialização de variáveis na árvore sintática. A poda remove elementos desnecessários e organiza a estrutura para análise mais eficiente.

```

1 # Função para podar uma variável
2 def podaVariavel(tree):
3     variavelAuxiliar = tree
4     descritorArvore = ()
5
6     # Adiciona o primeiro filho da árvore à variável 'descritorArvore'
7     descritorArvore += (variavelAuxiliar.children[0].children[0],)
8
9     # Verifica se a variável é um array (possui colchetes)

```

```

10     if len(variavelAuxiliar.children) > 1:
11         variavelAuxiliar1 = variavelAuxiliar.children[1].children
12         elementosPoda = ()
13
14         if len(variavelAuxiliar1) == 4:
15             # Poda o array com dois colchetes
16             elementosPoda += (variavelAuxiliar1[0].children[0].children[0],)
17             variavelAuxiliar1[0].children[1].children = podaExpressao(
18                 variavelAuxiliar1[0].children[1])
19             elementosPoda += (variavelAuxiliar1[0].children[1],)
20             elementosPoda += (variavelAuxiliar1[0].children[2].children[0],)
21             elementosPoda += variavelAuxiliar1[1].children
22             variavelAuxiliar1[2].children = podaExpressao(variavelAuxiliar1[2])
23             elementosPoda += (variavelAuxiliar1[2],)
24             elementosPoda += variavelAuxiliar1[3].children
25         else:
26             # Poda o array com um colchete
27             elementosPoda += variavelAuxiliar1[0].children
28             variavelAuxiliar1[1].children = podaExpressao(variavelAuxiliar1[1])
29             elementosPoda += (variavelAuxiliar1[1],)
30             elementosPoda += variavelAuxiliar1[2].children
31
32             # Atualiza os filhos da variável com a expressão podada
33             variavelAuxiliar.children[1].children = elementosPoda
34
35             # Atualiza a árvore com a variável podada
36             tree.children = descritorArvore
37     return tree

```

Código 22: Poda de Variáveis e Expressões

As funções `podaVariavel` e `podaExpressao` simplificam ainda mais a árvore sintática, removendo nós desnecessários e simplificando as expressões para facilitar a análise posterior.

Essas funções desempenham um papel crucial na análise semântica e na otimização da estrutura do código, garantindo que as variáveis e expressões estejam corretamente representadas e manipuladas no processo de compilação.

## 6.4 Entrada e saída da análise semântica

Para executar a aplicação, execute o código

```
1 python main.py tests/<arquivo_desejado>
```

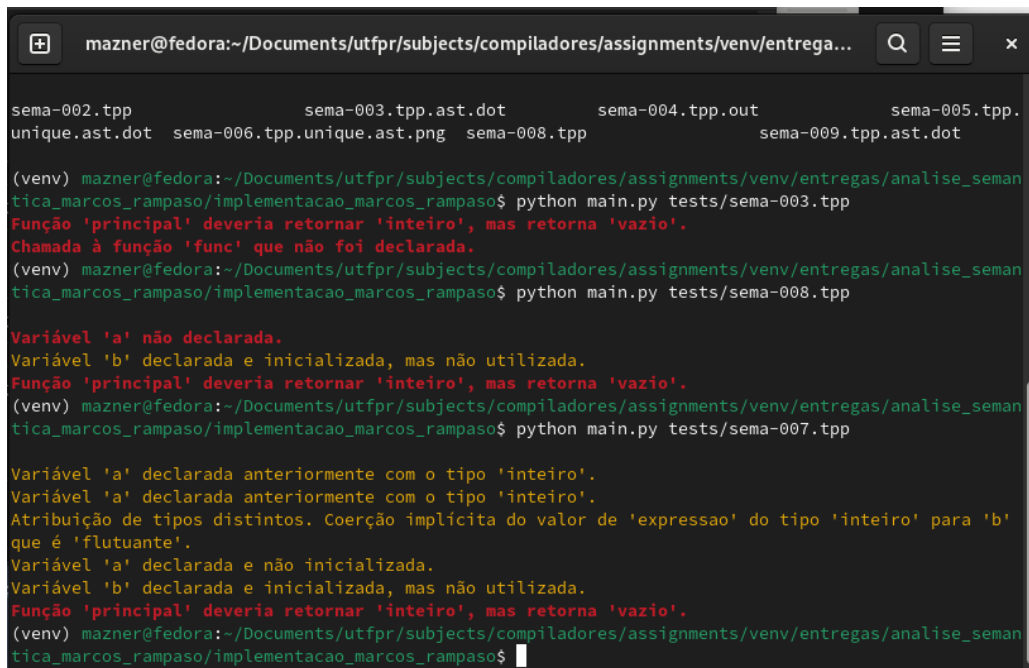
Código 23: Comando para execução da análise sintática

A saída esperada será como vista na Figura 3

## 6.5 Conclusões sobre a análise semântica

A análise semântica é uma etapa crucial no processo de compilação, onde o código fonte é verificado para garantir que ele adere às regras e restrições da linguagem de programação. Durante esta fase, é realizada a construção da Tabela de Símbolos, uma estrutura de dados fundamental que armazena informações sobre variáveis e funções, permitindo que o compilador valide o escopo e os tipos de dados utilizados no código.

A implementação das funções para análise semântica, como as descritas neste documento, exemplifica a complexidade envolvida na verificação de coerção de tipos, na detecção de erros em



```
mazner@fedora:~/Documents/utfpr/subjects/compiladores/assignments/venv/entrega...
sema-002.tpp      sema-003.tpp.ast.dot      sema-004.tpp.out      sema-005.tpp.
unique.ast.dot    sema-006.tpp.unique.ast.png  sema-008.tpp          sema-009.tpp.ast.dot

(venv) mazner@fedora:~/Documents/utfpr/subjects/compiladores/assignments/venv/entregas/analise_seman
tica_marcos_rampaso/implementacao_marcos_rampaso$ python main.py tests/sema-003.tpp
Função 'principal' deveria retornar 'inteiro', mas retorna 'vazio'.
Chamada à função 'func' que não foi declarada.
(venv) mazner@fedora:~/Documents/utfpr/subjects/compiladores/assignments/venv/entregas/analise_seman
tica_marcos_rampaso/implementacao_marcos_rampaso$ python main.py tests/sema-008.tpp

Variável 'a' não declarada.
Variável 'b' declarada e inicializada, mas não utilizada.
Função 'principal' deveria retornar 'inteiro', mas retorna 'vazio'.
(venv) mazner@fedora:~/Documents/utfpr/subjects/compiladores/assignments/venv/entregas/analise_seman
tica_marcos_rampaso/implementacao_marcos_rampaso$ python main.py tests/sema-007.tpp

Variável 'a' declarada anteriormente com o tipo 'inteiro'.
Variável 'a' declarada anteriormente com o tipo 'inteiro'.
Atribuição de tipos distintos. Coerção implícita do valor de 'expressao' do tipo 'inteiro' para 'b'
que é 'flutuante'.
Variável 'a' declarada e não inicializada.
Variável 'b' declarada e inicializada, mas não utilizada.
Função 'principal' deveria retornar 'inteiro', mas retorna 'vazio'.
(venv) mazner@fedora:~/Documents/utfpr/subjects/compiladores/assignments/venv/entregas/analise_seman
tica_marcos_rampaso/implementacao_marcos_rampaso$
```

Figura 3: Saída após a execução da análise semântica

declarações de variáveis e funções, e na aplicação de regras semânticas específicas da linguagem TPP. Através dessas verificações, o compilador assegura que o código não apenas segue a sintaxe correta, mas também que a lógica semântica subjacente é válida, prevenindo potenciais erros de execução.

Por fim, o processo de compilação culmina na preparação do código para ser traduzido em um formato executável, garantindo que ele esteja livre de erros semânticos e pronto para ser otimizado e executado eficientemente.

## 7 Geração de Código

Por fim, o processo de compilação culmina na preparação do código para ser traduzido em um formato executável, garantindo que ele esteja livre de erros semânticos e pronto para ser otimizado e executado eficientemente. Esta fase, conhecida como *geração de código*, é crucial para a transformação da representação intermediária do programa em um formato que possa ser executado pelo sistema operacional.

A geração de código é o estágio final da compilação onde o código intermediário (IR) é convertido em código de máquina ou em outra forma intermediária adequada para a execução. Para a geração de código LLVM, esse processo envolve a criação de uma representação em LLVM IR (Intermediate Representation), que é um formato de código de baixo nível utilizado pelo compilador LLVM.

A geração do código IR LLVM pode envolver as seguintes etapas:

- **Criação de Funções e Blocos:** Durante a geração de código, funções são criadas e associadas a blocos de código. Blocos de entrada e saída, como `entry_block` e `exit_block`, são fundamentais para garantir que a execução do código seja bem estruturada e livre de erros. O uso apropriado desses blocos ajuda a definir o fluxo de controle do programa.
- **Conversão de Tipos e Variáveis:** Os tipos de dados e variáveis presentes na representação intermediária são convertidos para o formato IR LLVM. Isso inclui a definição de tipos primitivos e a alocação de espaço para variáveis no código gerado.
- **Otimizações de Código:** Durante a geração do IR LLVM, pode-se aplicar otimizações



específicas para melhorar a eficiência do código gerado. Essas otimizações podem incluir a eliminação de código morto, a propagação de constantes e a simplificação de expressões.

- **Gerenciamento de Funções e Parâmetros:** A forma como funções e parâmetros são tratados na geração de código pode variar. Estratégias diferenciadas podem incluir a criação de funções com parâmetros adequados e a manipulação eficiente desses parâmetros no IR LLVM.
- **Estruturas de Controle:** Blocos condicionais e estruturas de laços são traduzidos em instruções de controle no IR LLVM. O tratamento correto dessas estruturas é essencial para garantir que o fluxo de controle do programa seja preservado.

## 7.1 Funcionamento geral de uma geração de código LLVM

Como visto no site da LLVM Foundation (2024), o LLVM é um conjunto de ferramentas e bibliotecas para a construção de compiladores e outras ferramentas de processamento de código. A geração de código LLVM envolve várias etapas, que permitem transformar o código-fonte de uma linguagem de alto nível em código executável eficiente. A seguir, descrevemos o funcionamento geral deste processo:

- **Frontend:** A primeira etapa é a análise do código-fonte, onde o compilador lê o código escrito na linguagem de programação original e o converte em uma representação intermediária, conhecida como *Intermediate Representation* (IR). Essa etapa pode envolver a análise léxica, sintática e semântica.
- **Intermediate Representation (IR):** O LLVM usa uma representação intermediária de baixo nível e de tipagem estática, que é independente da arquitetura do processador. O código IR é mais abstrato do que o código de máquina e permite várias otimizações antes da geração do código final. A IR é uma etapa crucial na qual são realizadas transformações e otimizações para melhorar a eficiência do código.
- **Otimizações:** Durante a fase de otimização, o LLVM aplica uma série de técnicas para melhorar o desempenho do código IR. Essas otimizações podem incluir eliminação de código morto, propagação de constantes, e outras melhorias que reduzem o tempo de execução e o consumo de recursos.
- **Backend:** Finalmente, o código IR otimizado é convertido em código de máquina específico para a arquitetura do processador alvo. O backend do LLVM gera o código assembly ou binário que será executado no hardware. Esta etapa envolve a tradução da IR para instruções específicas da arquitetura e a organização eficiente do código gerado.
- **Emissão de Código:** O código gerado é então emitido em um formato executável, que pode ser diretamente executado pelo sistema operacional ou vinculado com outras partes do programa para formar um executável completo.

A flexibilidade e a modularidade do LLVM permitem que desenvolvedores e pesquisadores criem compiladores e ferramentas de análise de código personalizados com facilidade, aproveitando a robustez da infraestrutura fornecida pelo LLVM.

## 7.2 Exemplo de Código em Linguagem de Máquina

O código a seguir ilustra a conversão de um programa básico em C para um código de baixo nível em LLVM IR. O código em C original é:

```
1 int main(){
2     int a = 1;
3     float b = 1.0;
4     float c = 2.0;
5     a = a + a;
```



```
6   b = b + c;  
7   return 0;  
8 }
```

Código 24: Código de soma básica em C

Após a conversão para LLVM IR, o código resultante é:

```
1 define i32 @main() #0 {  
2     ; Aloca espaço para variáveis locais  
3     %1 = alloca i32, align 4  
4     %a = alloca i32, align 4  
5     %b = alloca float, align 4  
6     %c = alloca float, align 4  
7  
8     ; Inicializa as variáveis  
9     store i32 0, i32* %1  
10    store i32 1, i32* %a, align 4  
11    store float 1.000000e+00, float* %b, align 4  
12    store float 2.000000e+00, float* %c, align 4  
13  
14    ; Executa a operação a = a + a  
15    %2 = load i32* %a, align 4  
16    %3 = load i32* %a, align 4  
17    %4 = add nsw i32 %2, %3  
18    store i32 %4, i32* %a, align 4  
19  
20    ; Executa a operação b = b + c  
21    %5 = load float* %b, align 4  
22    %6 = load float* %c, align 4  
23    %7 = fadd float %5, %6  
24    store float %7, float* %b, align 4  
25  
26    ; Retorna 0  
27    ret i32 0  
28 }
```

Código 25: Código LLVM

Neste código LLVM IR:

- `alloca` é usado para alocar espaço na pilha para variáveis locais.
- `store` armazena valores em variáveis.
- `load` carrega valores de variáveis.
- `add` realiza uma adição de inteiros.
- `fadd` realiza uma adição de ponto flutuante.
- `ret` retorna um valor da função.

A transformação do código C para LLVM IR permite que o compilador aplique otimizações e gere o código de máquina específico para a arquitetura de destino.

### 7.3 Estratégias Utilizadas

A estratégia para a geração de código IR LLVM no projeto envolve uma abordagem detalhada e recursiva para traduzir a árvore sintática gerada pelo analisador semântico em código LLVM. O processo é implementado na classe ‘GenCode’, e as principais estratégias incluem:

## 7.4 Inicialização do código

O código começa com a inicialização dos componentes necessários da biblioteca LLVM e a configuração do ambiente para a geração de código. Este processo é essencial para garantir que o código gerado seja compatível com o ambiente de execução e que as operações básicas possam ser realizadas corretamente. Abaixo está uma descrição detalhada das etapas envolvidas:

- `llvm.initialize()` : Inicializa a biblioteca LLVM.
- `llvm.initialize_all_targets()` : Inicializa todos os alvos suportados pela LLVM.
- `llvm.initialize_native_target()` : Inicializa o alvo nativo do sistema.
- `llvm.initialize_native_asmprinter()` : Inicializa o impressor de assembly nativo, necessário para gerar código assembly.

### 7.4.1 Configuração do Módulo LLVM

- **Criação do Módulo:**

```
self.module = ir.Module('main_module')
```

Cria um módulo LLVM com o nome 'main\_module'.

- **Configuração do Triplo e Layout de Dados:**

```
self.module.triple = llvm.get_process_triple()
target = llvm.Target.from_triple(self.module.triple)
target_machine = target.create_target_machine()
self.module.data_layout = target_machine.target_data
```

Configura o triplo do módulo com o triplo do processo atual, define a máquina alvo e o layout de dados. Isso assegura que o código gerado seja otimizado para o ambiente de execução.

## 7.5 Definição dos Tipos de Dados

- **Tipos de Dados Básicos:**

```
self.FLOAT = ir.FloatType()
self.INT = ir.IntType(32)
self.ZERO = ir.Constant(ir.IntType(32), 0)
```

Define tipos básicos como `FloatType` e `IntType` e inicializa uma constante zero para inteiros.

### 7.5.1 Declaração de Funções Externas

- **Funções de Entrada e Saída:**

```
1 self.escrevaInteiro = ir.Function(
2     self.module,
3     ir.FunctionType(ir.VoidType(), [ir.IntType(32)]),
4     name="escrevaInteiro"
5 )
6 self.escrevaFlutuante = ir.Function(
7     self.module,
8     ir.FunctionType(ir.VoidType(), [ir.FloatType()]),
9     name="escrevaFlutuante"
```

```

10 )
11 self.leiaInteiro = ir.Function(
12     self.module,
13     ir.FunctionType(ir.IntType(32), []),
14     name="leiaInteiro"
15 )
16 self.leiaFlutuante = ir.Function(
17     self.module,
18     ir.FunctionType(ir.FloatType(), []),
19     name="leiaFlutuante"
20 )

```

Código 26: Funções de Entrada e Saída

Declara funções externas para escrita e leitura de inteiros e flutuantes. Essas funções são usadas para operações de entrada e saída.

### 7.5.2 Inicialização e Configuração

O código começa com a inicialização dos componentes necessários da biblioteca LLVM e a configuração do ambiente para a geração de código. Este processo é essencial para garantir que o código gerado seja compatível com o ambiente de execução e que as operações básicas possam ser realizadas corretamente. Abaixo está uma descrição detalhada das etapas envolvidas:

### 7.5.3 Inicialização da Biblioteca LLVM

```

1 import os
2 import sys
3 import logging
4 from myerror import MyError
5
6 from llvmlite import ir
7 from llvmlite import binding as llvm
8
9 logging.basicConfig(
10     level = logging.DEBUG,
11     filename = "gencode.log",
12     filemode = "w",
13     format = "%(filename)10s:%(lineno)4d:%(message)s"
14 )
15 log = logging.getLogger()
16 error_handler = MyError('GenCodeErrors')
17 root = None
18
19 class GenCode():
20     def __init__(self):
21         # Inicialização da Biblioteca LLVM
22         llvm.initialize()
23         llvm.initialize_all_targets()
24         llvm.initialize_native_target()
25         llvm.initialize_native_asmprinter()

```

Código 27: Inicialização da Biblioteca LLVM

### 7.5.4 Configuração do Módulo LLVM

```
1      # Configuração do Módulo LLVM
2      self.module = ir.Module('main_module')
3      self.module.triple = llvm.get_process_triple()
4      target = llvm.Target.from_triple(self.module.triple)
5      target_machine = target.create_target_machine()
6      self.module.data_layout = target_machine.target_data
```

Código 28: Configuração do Módulo LLVM

### 7.5.5 Definição dos Tipos de Dados

```
1      # Definição dos Tipos de Dados
2      self.FLOAT = ir.FloatType()
3      self.INT = ir.IntType(32)
4      self.ZERO = ir.Constant(ir.IntType(32), 0)
```

Código 29: Definição dos Tipos de Dados

### 7.5.6 Declaração de Funções Externas

```
1      # Declaração de Funções Externas
2      self.escrevaInteiro = ir.Function(self.module, ir.FunctionType(ir.
VoidType(), [ir.IntType(32)]), name="escrevaInteiro")
3      self.escrevaFlutuante = ir.Function(self.module, ir.FunctionType(ir.
VoidType(), [ir.FloatType()]), name="escrevaFlutuante")
4      self.leiaInteiro = ir.Function(self.module, ir.FunctionType(ir.IntType
(32), []), name="leiaInteiro")
5      self.leiaFlutuante = ir.Function(self.module, ir.FunctionType(ir.
FloatType(), []), name="leiaFlutuante")
```

Código 30: Declaração de Funções Externas

### 7.5.7 Declaração de Blocos e Listas

```
1      # Declaração de Blocos
2      self.entry_block = None
3      self.exit_block = None
4      self.block = None
5
6      # Lista de Funções e Variáveis
7      self.funcoes = []
8      self.variaveisGlobais = []
9      self.variaveisLocais = []
10     self.argumentosFuncao = []
```

Código 31: Declaração de Blocos e Listas

### 7.5.8 Processamento de Declarações

- Criação do Módulo:  
self.module = ir.Module('main\_module')

Cria um módulo LLVM com o nome 'main\_module'.

- **Configuração do Triplo e Layout de Dados:**

```
self.module.triple = llvm.get_process_triple()
target = llvm.Target.from_triple(self.module.triple)
target_machine = target.create_target_machine()
self.module.data_layout = target_machine.target_data
```

Configura o triplo do módulo com o triplo do processo atual, define a máquina alvo e o layout de dados. Isso assegura que o código gerado seja otimizado para o ambiente de execução.

### 7.5.9 Definição dos Tipos de Dados

- **Tipos de Dados Básicos:**

```
self.FLOAT = ir.FloatType()
self.INT = ir.IntType(32)
self.ZERO = ir.Constant(ir.IntType(32), 0)
```

Define tipos básicos como FloatType e IntType e inicializa uma constante zero para inteiros.

### 7.5.10 Declaração de Funções Externas

- **Funções de Entrada e Saída:**

```
1 self.escrevaInteiro = ir.Function(self.module, ir.FunctionType(ir.VoidType
2   (), [ir.IntType(32)]), name="escrevaInteiro")
3 self.escrevaFlutuante = ir.Function(self.module, ir.FunctionType(ir.
4   VoidType(), [ir.FloatType()]), name="escrevaFlutuante")
5 self.leiaInteiro = ir.Function(self.module, ir.FunctionType(ir.IntType(32),
6   []), name="leiaInteiro")
7 self.leiaFlutuante = ir.Function(self.module, ir.FunctionType(ir.FloatType
8   (), []), name="leiaFlutuante")
```

Código 32: Funções de Entrada e Saída

Declara funções externas para escrita e leitura de inteiros e flutuantes. Essas funções são usadas para operações de entrada e saída.

## 7.6 Estrutura da geração de código

- **Inicialização e Configuração:** O código começa com a inicialização dos componentes necessários da biblioteca LLVM, incluindo a configuração do módulo, definição de tipos e declaração de funções externas para entrada e saída. O módulo LLVM é configurado com o triplo de destino e layout de dados apropriado, assegurando que o código gerado seja compatível com o ambiente de execução.
- **Processamento de Declarações:** A classe 'GenCode' processa diferentes tipos de declarações, como funções e variáveis. As funções são declaradas com seus tipos e parâmetros, e as variáveis são gerenciadas como globais ou locais, dependendo do contexto. Arrays e variáveis são inicializados de forma apropriada, com suporte para arrays unidimensionais e bidimensionais.

- **Geração de Código para Funções:** Cada função é processada para criar blocos básicos de entrada e saída, e o corpo da função é gerado de forma recursiva. O código para o corpo da função inclui o processamento de variáveis locais, atribuições, chamadas de funções, e operações condicionais e de repetição. O código é gerado em blocos básicos, que são conectados através de instruções de salto condicional e incondicional.
- **Tratamento de Estruturas Condicionais e Loops:** O código trata estruturas condicionais (if-else) e loops (repita) utilizando blocos básicos distintos para cada parte da estrutura. As instruções de comparação e controle de fluxo são geradas para garantir que o comportamento das estruturas condicionais e de repetição seja corretamente representado no código LLVM.
- **Chamada de Funções e Expressões:** A geração de código para chamadas de funções e expressões é feita utilizando a API da biblioteca LLVM. Expressões aritméticas e lógicas são traduzidas em instruções LLVM apropriadas, e as chamadas de funções são gerenciadas com a criação de blocos de código e parâmetros corretos.
- **Armazenamento do Código Gerado:** Após a geração do código, o módulo LLVM é salvo em um arquivo '.ll' na pasta 'resultadosLlvm'. Este arquivo contém o código IR LLVM resultante, pronto para ser compilado para código de máquina ou utilizado para outras análises.

A abordagem recursiva e a organização em blocos básicos permitem uma tradução eficaz e estruturada da árvore sintática em código LLVM, garantindo que o código gerado seja bem estruturado e facilmente manipulável.

## 7.7 Otimizações

Durante o processo de implementação da geração de código LLVM, várias otimizações foram aplicadas para melhorar a eficiência e a legibilidade do código gerado. A seguir estão as principais otimizações empregadas:

- **Uso Eficiente de Blocos Básicos:** A estrutura do código LLVM é organizada em blocos básicos, permitindo a criação de fluxos de controle claros e a reutilização eficiente de blocos para diferentes estruturas de controle, como loops e condicionais.
- **Inicialização de Variáveis com Zero:** Para garantir que as variáveis sejam inicializadas com um valor conhecido, variáveis globais e locais são inicializadas com zero. Esta abordagem evita problemas relacionados a valores não inicializados e melhora a previsibilidade do código gerado.
- **Alinhamento de Variáveis:** As variáveis são alinhadas a um byte, garantindo a eficiência de acesso e evitando problemas de alinhamento durante a execução do código. O alinhamento é definido como 4 bytes, que é uma prática comum para tipos de dados inteiros e flutuantes.
- **Funções de Entrada/Saída Reutilizáveis:** Funções externas como 'escrevaInteiro', 'escrevaFlutuante', 'leiaInteiro' e 'leiaFlutuante' são definidas para realizar operações comuns de entrada e saída. Isso evita a repetição de código e facilita a manutenção e extensibilidade.
- **Uso de Constantes para Valores Fixos:** Constantes são usadas para representar valores fixos, como zero. Isso reduz a quantidade de código gerado e melhora a eficiência, evitando a necessidade de recriar constantes em cada operação.
- **Geração Condicional de Código:** O código é gerado de forma condicional, utilizando instruções como `cbranch` para saltar entre diferentes blocos básicos com base em condições. Isso otimiza o controle de fluxo e melhora a performance geral do código.
- **Gerenciamento de Arrays:** Arrays são gerenciados de forma eficiente, tanto para arrays unidimensionais quanto bidimensionais. O uso de blocos básicos separados para lidar com diferentes tipos de arrays e o ajuste dinâmico do tamanho garantem uma alocação adequada de memória.

- **Otimização de Expressões Aritméticas e Lógicas:** As expressões aritméticas e lógicas são traduzidas em instruções LLVM apropriadas, utilizando operações eficientes e evitando a criação desnecessária de variáveis temporárias.
- **Organização de Funções e Parâmetros:** As funções são organizadas de forma clara, com a declaração de parâmetros e tipos apropriados. Isso melhora a legibilidade do código e garante que as funções sejam chamadas com os argumentos corretos.

## 7.8 Entrada e saída

Para executar a aplicação, você precisará de um arquivo `.tpp` como o exemplo do Código 33. Após criar o arquivo, execute no terminal de sua máquina o comando `python main.py <caminho_para_seu_arquivo>` e então, será gerada uma saída tal como o Código 34.

```

1 inteiro: T
2 T:= 4
3
4 inteiro: V1[T]
5
6 inteiro somavet(inteiro: vet[], inteiro: tam)
7     inteiro: result
8     result := 0
9
10    inteiro: i
11    i := 0
12
13    repita
14        result := result + vet[i]
15        i := i + 1
16    até i = tam - 1
17
18    retorna(result)
19 fim
20
21 inteiro principal ()
22     inteiro: x
23     x := somavet(V1,T)
24     retorna(0)
25 fim

```

Código 33: Exemplo de código `.tpp`

```

1 ; ModuleID = "main_module"
2 target triple = "x86_64-unknown-linux-gnu"
3 target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
4 :16:32:64-S128"
5
6 declare void @"escrevaInteiro"(i32 %".1")
7
8 declare void @"escrevaFlutuante"(float %".1")
9
10 declare i32 @"leiaInteiro"()
11
12 declare float @"leiaFlutuante"()
13
14 @"T" = global i32 0, align 4
15 @"V1" = global i32 0, align 4
16 define i32 @"somavet"(float %"id")
17 {

```

```
17 bloco_entrada:  
18 bloco_saida:  
19     ret i32 0  
20 }  
21  
22 define i32 @"main"()  
23 {  
24     bloco_entrada:  
25     bloco_saida:  
26         ret i32 0  
27 }
```

Código 34: Saída gerada pelo código .tpp

## Referências

- Beazley, David. 2024. Ply (python lex-yacc). Versão 3.11, disponível em Python Package Index (PyPI). <http://www.dabeaz.com/ply/>.
- Foundation, LLVM. 2024. Llvm language reference manual. Acessado em: 11/09/2024. <https://llvm.org/docs/LangRef.html>.
- JARGAS, Aurélio Marinho. 2012. *Expressões regulares: uma abordagem divertida*. São Paulo, SP: Novatec 4th edn.
- Meyer, C. 2024. anytree: Powerful and lightweight tree data structure. Versão 2.8.0, disponível em Python Package Index (PyPI). <https://anytree.readthedocs.io/>.