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

Danilo Balman Garcia
Ra: 2482088
[1]Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

This article is a study of the process of implementing programming languages and compilers. The language utilized for this project is the *TPP* language. This study emphasizes the importance of the knowledge of compilers and their behavior. This project will follow some steps towards the abstraction of a compiler, like lexical analysis and development, syntax analysis, semantic analysis, and code generation.

Resumo

Este artigo é um estudo do processo de implementação de linguagens de programação e compiladores. A linguagem utilizada para este projeto é a linguagem *TPP*. Este estudo enfatiza a importância do conhecimento sobre compiladores e seu comportamento. Este projeto seguirá alguns passos em direção à abstração de um compilador, como análise e desenvolvimento léxico, análise sintática, análise semântica e geração código.

1 Introdução

Originalmente, os computadores eram programados diretamente em binário, ou seja, em 0's e 1's, resultando em baixa produtividade dos programadores. Assim, a primeira linguagem de alto nível foi criada em 1953. Porém, o seu desempenho era 10x a 20x inferior em relação ao código de máquina. Contudo, quatro anos depois, a mesma linguagem lançou a primeira versão do primeiro compilador moderno, que gerava um desempenho semelhante, sendo ele uma inspiração para os próximos compiladores.
Dito isso, os compiladores têm como objetivo transformar um programa executável de uma linguagem fonte para um programa executável em uma linguagem destino, normalmente sendo código de máquina ou código de montagem, com a expectativa de que o programa resultante, de alguma maneira, seja melhor que o original, e sem alterar o significado do programa original. Este artigo apresentará os compiladores e as ferramentas de compilação para a linguagem *TPP* em quatro fases: a análise léxica, que consiste na leitura do código-fonte, analisando sua estrutura e realizando a leitura de *tokens* (marcas), utilizando expressões regulares, autômatos finitos e a ferramenta *Lex*; a análise sintática, que cria uma estrutura usando a biblioteca *anytree* e os *tokens* da etapa anterior para ler o arquivo de entrada e criar uma estrutura que respeita as regras sintáticas da linguagem *TPP*; a análise semântica, que verifica o contexto de cada operação para assegurar a coesão e o sentido do código; e, por fim, a geração de código, onde ocorre a tradução do código de entrada TPP.

Recursos

- Python 3.10.12
  - Biblioteca: Ply 3.11
  - Biblioteca: Regex 2023.12.25
  - Biblioteca: anytree 2.12.1
  - Biblioteca: llvmlite 0.43.0
- Editor de texto: Visual Studio code 1.85.2
- Sistema operacional: Linux Mint 21.3
- Plataforma de Edição de Documentos: Overleaf

2 Análise Léxica

Nossa linguagem de estudo para esse projeto será a linguagem *TPP*, uma linguagem de programação em português e quase fortemente tipificada. Os tipos das funções podem ser omitidos. Ela suporta vetores unidimensionais e bidimensionais. A objetivo é ser uma linguagem simples para o estudo e desenvolvimento de um compilador.

2.1 Varredura

A análise léxica de um compilador é o momento em que o compilador lê o programa-fonte como um arquivo de caracteres e o separa em *tokens* (marcas) do arquivo, sendo esses *tokens* os identificadores, palavras-chave, símbolos especiais do programa original. As denominações Sistema de Varredura, Analisador Léxico e Scanner são equivalentes.

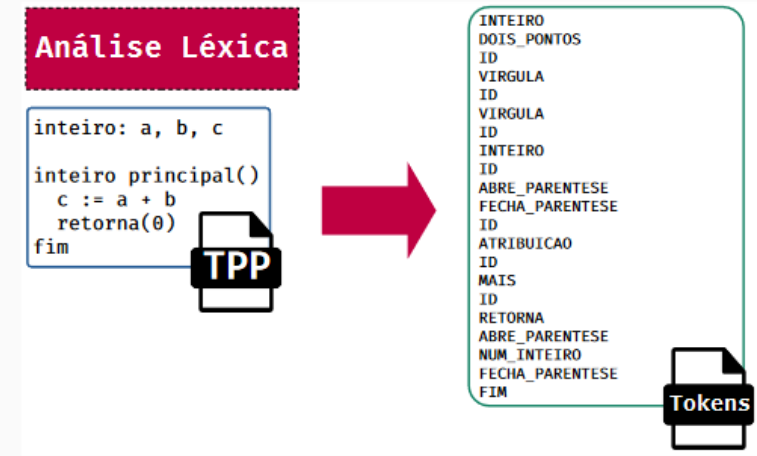

Figura 1: Leitura léxica

O processo de varredura pode ser realizado por expressões regulares, ou implementando o analisador com autômatos finitos. No caso do compilador para a linguagem *TPP* usada para esse estudo, a leitura é realizada utilizando a biblioteca *Ply*, uma biblioteca que adiciona *Lex*, um gerador de leitor léxico para o *python*, e acompanhado utilizamos *Regex*, uma biblioteca de expressões regulares para Python, e a tabela 1, 2 e 3 apresenta as marcas e suas expressões regulares correspondentes. Uma expressão regular é uma sequência de caracteres que define um padrão de busca, utilizado para identificar sequências específicas de caracteres em um texto.

2.2 Autômatos finitos

Autômatos são uma forma matemática de descrever certos tipos de algoritmos, principalmente utilizados para descrever reconhecimento de padrões em cadeias de entrada, podendo ser utilizados para a implementação de sistemas de varreduras, por exemplo, para o reconhecimento de identificadores e facilitando a criação de expressões regulares. A figura abaixo apresenta um autômato que pode ser utilizado para esse caso.

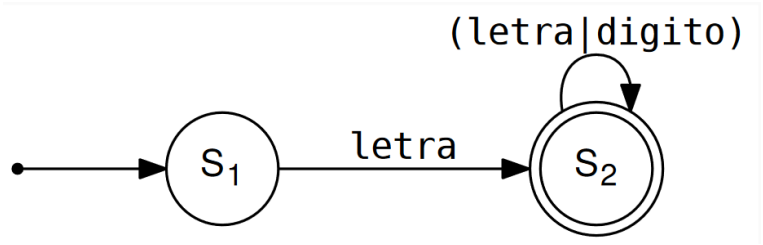

Figura 2: Autômato para identificadores

A expressão regular para esse autômato seria letra(letra\|dígito)*, sendo ”letra”um conjunto que contém as letras do alfabeto e suas variações com acento, e dígito é o conjunto de 0 a 10, e isso significa que o compilador aceite qualquer nome para o identificador, desde que ele comece com uma letra, podendo ter letras e número após.

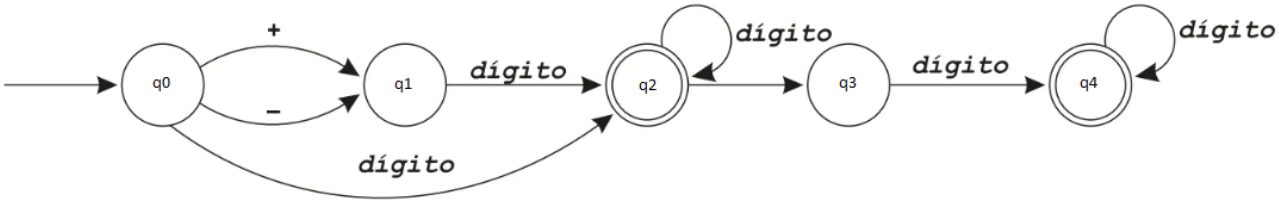

Figura 3: Autômato para flutuante

Esse autômato começa recebendo um sinal e um número, ou um número direto para chegar ao estado **q3**, onde ele entra em um estado de aceitação caso seja um número inteiro. Caso contrário, eventualmente ele entrará no estado **q4**, transitando para o estado **q5** na presença de um ponto (‘.’).

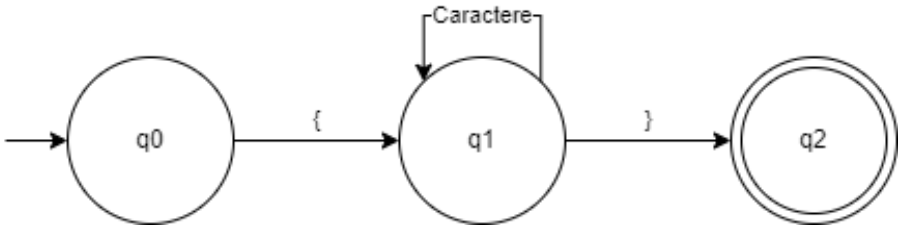

Figura 4: Autômato para comentarios

Este é mais simples: um autômato que identifica os comentários. Ele transita do estado **q0** para o estado **q1** com ‘{’ (que inicia o comentário) e entra em um loop ao receber um caractere, finalizando apenas ao receber ‘}’. Os comentários não se aninham e, normalmente, em uma varredura, eles são ignorados.

2.3 Implementação da varredura

O programa que faz a varredura nos arquivos *TPP* foi desenvolvido em Python, utilizando a biblioteca *PLY*, que é utilizado para a criação de analises léxicas e sintaticas, facilitando a im-

plementação do programa, que le um código-fonte e devolve as marcas ou retorna um erro da compilação. As tabelas abaixo apresentam as expressões utilizadas nesse projeto.

Token	Expressão Regular
t_MAIS	`r'\+'`
t_MENOS	`r'-'`
t_VEZES	`r'\*'`
t_DIVIDE	`r'/'`
t_ABRE_PARENTESE	`r'\('`
t_FECHA_PARENTESE	`r'\)'`
t_ABRE_COLCHETE	`r'\['`
t_FECHA_COLCHETE	`r'\]'`
t_VIRGULA	`r','`
t_ATRIBUICAO	`r':='`
t_DOIS_PONTOS	`r':'`
t_OU	`r'||'`
t_E	`r'&&'`
t_NAO	`r'!'`
t_DIFERENTE	`r'<>'`
t_MENOR_IGUAL	`r'<='`
t_MAIOR_IGUAL	`r'>='`
t_MENOR	`r'<'`
t_MAIOR	`r'>'`
t_IGUAL	`r'='`

Tabela 1: Expressões regulares dos símbolos aceitos

A linguagem *TPP* diferente do padrão das linguagens de programação, tem suas palavras-chave (palavras reservadas para que o compilador não as utilize como identificador) em português, as mesmas tem prioridade na leitura do arquivo, para não causar nenhum tipo de ambiguidade na compilação. As identificações podem ser criadas com acentos, exigindo apenas letra no começo e que não tenham símbolos.

Token	Expressão Regular
flutuante	`r'\d+[eE][-+]?\d+|(\.\d+|\d+\.\d*)([eE][-+]?\d+)?'`
id	`r'(' + letra + r'(' + digito + r'+|_|' + letra + r')*)'`
inteiro	`r'\d+'`
notacao_cientifica	`r'(' + sinal + r'([1-9])\.' + digito + r'+[eE]' + sinal + digito + r'+)'`
digito	`r'([0-9])'`
letra	`r'([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ])'`
sinal	`r'([\-\+]?)'`

Tabela 2: *Tokens* dos identificadores e expressões dos números e notação

Palavra Reservada	Expressão Regular
se	`r'se'`
então	`r'então'`
senão	`r'senão'`
fim	`r'fim'`
repita	`r'repita'`
flutuante	`r'flutuante'`
retorna	`r'retorna'`
até	`r'até'`
leia	`r'leia'`
escreva	`r'escreva'`
inteiro	`r'inteiro'`

Tabela 3: Palavras reservadas e seus *tokens* correspondentes

E por ultimo, os comentarios tem a seguinte composição Comentario, sendo possivel comentarios que cubram varias linhas, a expressão regular utilizada para a leitura de comentarios é `r'(\{((.|\n)*?)\})'`.

Função *main()*

Algoritmo 1: tpplex.py

```
1   def main():
2
3       global showKey
4       global haveTPP
5
6       locationTTP = None
7
8       for i in range(len(sys.argv)):
9           aux = argv[i].split('.')
10          if aux[-1] == 'tpp':
11              haveTPP = True
12              locationTTP = i
13          if(argv[i] == '-k'):
14              showKey = True
15
16      if(len(sys.argv) < 3 and showKey == True):
17          raise TypeError(le.newError(showKey, 'ERR-LEX-USE'))
18
19      if haveTPP == False:
20        raise IOError(le.newError(showKey, 'ERR-LEX-NOT-TPP'))
21      elif not os.path.exists(argv[locationTTP]):
22          raise IOError(le.newError(showKey, 'ERR-LEX-FILE-NOT-EXISTS'))
23      else:
24          data = open(argv[locationTTP])
25
26          source_file = data.read()
27          lexer.input(source_file)
28
29          # Tokenize
30          while True:
31              tok = lexer.token()
32              if not tok:
33                  #raise IOError(le.newError(showKey, 'ERR-LEX-INV-CHAR'))
34                  break       # No more input
```

```
35              #print(tok)
36              print(tok.type)
37              #print(tok.value)
```

A função *main* é onde tudo é colocada na pratica, primeiramente é feito um tratamento de erro que será discutido no próximo tópico, e após o arquivo é lido e seus dados armazenados na variavel *source_file*, e inserido no lexer, que foi criado no escopo global, e então entra em loop onde os tokens são lidos e imprimidos

2.4 Exemplos e tratamento de erros

O pasta **tests** da implementação em 37 códigos com arquivos *TPP* para a realização de testes, para este exemplo, utilizaremos o arquivo "teste-005.tpp".

Algoritmo 2: tpplex.py

```
1  inteiro principal()
2    a :=+1
3    c := a + b
4    b:= 3 + a
5    c:= +3
6    a:= +3.5
7    a := 3.5 + 4.5
8  fim
```

O analizador irá ler o arquivo e separar os tokens, gerando esse resultado

Algoritmo 3: teste-005.tpp

```
1  INTEIRO
2  ID
3  ABRE_PARENTESE
4  FECHA_PARENTESE
5  ID
6  ATRIBUICAO
7  MAIS
8  NUM_INTEIRO
9  ID
10 ATRIBUICAO
11 ID
12 MAIS
13 ID
14 ID
15 ATRIBUICAO
16 NUM_INTEIRO
17 MAIS
18 ID
19 ID
20 ATRIBUICAO
21 MAIS
22 NUM_INTEIRO
23 ID
24 ATRIBUICAO
25 MAIS
26 NUM_PONTO_FLUTUANTE
27 ID
28 ATRIBUICAO
```

```
29 NUM_PONTO_FLUTUANTE
30 MAIS
31 NUM_PONTO_FLUTUANTE
32 FIM
```

A análise léxica também tem o objetivo de apontar erros na leitura, como símbolos e caracteres inválidos, ou quando os parâmetros não são compatíveis com o programa. No analisador léxico do nosso compilador, é possível enviar um parâmetro '-k' para que os erros dos arquivos analisados retornem apenas um código de erro.

Código de Erro	**Significado**
ERR-LEX-USE	Uso: python tpplex.py file.tpp
ERR-LEX-NOT-TPP	Não é um arquivo .tpp.
ERR-LEX-FILE-NOT-EXISTS	Arquivo .tpp não existe.
ERR-LEX-INV-CHAR	Caracter inválido.

Tabela 4: Códigos de erro e seus significados

Classe MyError

Esta classe é responsável por criar mensagens com as informações necessárias. No código principal, é utilizado o método *newError(showKey, key, linha=None, coluna=None, **data)*. Todos esses parâmetros são utilizados para enviar uma mensagem apropriada para cada caso. O parâmetro *showKey*, se for verdadeiro, invés de imprimir uma mensagem de erro, ele apenas envia um código de erro. Os parâmetros linha e coluna são para apontar onde o erro ocorre, e data é um vetor de informações que preencherão a mensagem de erro.

Nesse exemplo utilizaremos o arquivo *teste-002.tpp* da pasta de testes do analisador léxico do nosso compilador, o código é curto e simples, mas tem um caractere invalido.

Algoritmo 4: teste-002.tpp

```
1 inteiro: a,b
2 ç
```

Resultado da análise:

```
INTEIRO                                  INTEIRO
DOIS_PONTOS                              DOIS_PONTOS
ID                                       ID
VIRGULA                                  VIRGULA
ID                                       ID
Erro[2][1]: Caracter inválido, valor: ç  ERR-LEX-INV-CHAR
```

Figura 5: Resultado da Analise do arquivo teste-002.tpp

O uso de códigos de erro tem o objetivo de automatizar os testes do algoritmo. A automatização dos testes é realizada pelo GitHub, onde o repositório dos algoritmos desse projeto é armazenado. Toda vez que o código é atualizado no GitHub, um teste automático ocorre, testando a integridade do código. E quando ocorre um erro, o erro é legivel e claro para quem que está tentando compilar o código, e com um simples parametro, o erro pode ser alterado para que facilite a automatização dos testes.

Algoritmo 5: tpplex.py

```
1     global showKey
```

```
2       global haveTPP
3
4       locationTTP = None
5
6       for i in range(len(sys.argv)):
7           aux = argv[i].split('.')
8           if aux[-1] == 'tpp':
9               haveTPP = True
10              locationTTP = i
11          if(argv[i] == '-k'):
12              showKey = True
13
14      if(len(sys.argv) < 3 and showKey == True):
15          raise TypeError(le.newError(showKey, 'ERR-LEX-USE'))
16
17      if haveTPP == False:
18        raise IOError(le.newError(showKey, 'ERR-LEX-NOT-TPP'))
19      elif not os.path.exists(argv[locationTTP]):
20          raise IOError(le.newError(showKey, 'ERR-LEX-FILE-NOT-EXISTS'))
```

Esse trecho de código ocorre um pouco antes da varredura. Primeiramente, são lidos todos os argumentos do código. Caso haja um *arquivo.tpp*, a variável global *haveTPP* é alterada para verdadeiro e sua posição na lista de argumentos é salva. Se for encontrado o argumento '-k', a variável *showKey* é definida como verdadeira. Após a leitura dos argumentos, o programa verifica se a quantidade de argumentos é suficiente e, em seguida, verifica se o segundo argumento é um arquivo *TPP* e se ele existe.

3 Análise Sintática

A gramática é um conjunto de regras que determinam o uso considerado correto da língua escrita, dito isso, a **Análise Sintática** determina a sintaxe e estrutura de um programa, que são as regras gramaticais ou **gramática livre de contexto** (GLC) de uma linguagem de programação. Uma GLC utiliza nomes e operações parecidas com as expressões regulares, a diferença está em que as regras da gramática livre de contexto podem ser recursivas. E o ato de analisar sequências dadas e verificar sua estrutura e integridade se chama *Parsing*.

3.1 Padrão BNF

O Formalismo de Backus-Naur (BNF) foi criado no final dos ano de 1950 para descrever a lingugagem ALGOL, sendo uma metassintaxe usada para expressar gramáticas GLC, e é agora ela e suas variantes são amplamente usada como notação para linguagem de programação. Uma especificação BNF é um conjunto de regras de derivação, escritas como:

Algoritmo 6: Exemplo de uma especificação BNF

```
1 <símbolo> ::= <expressão com símbolos>
```

Sendo o símbolo não terminal, e a expressão uma sequência de símbolos possívelmente separadas por uma barra vertical, '|', indicando uma escolha. O exemplo abaixo é uma gramática livre de contexto com a norma padrão BNF utilizado na implementação da análise sintática do compilador.

Algoritmo 7: Especificação da gramática "SE"

```
1
2     se : SE expressao ENTAO corpo FIM
3        | SE expressao ENTAO corpo SENAO corpo FIM
```

Neste exemplo, a estrutura das condições da linguagem *TPP* é definida pelas seguintes regras:

- A gramática começa com a palavra-chave *SE*.
- Expressão representa a condição que vai ser avaliada, sendo qualquer uma que represente um valor booleano.
- *ENTAO* é a palavra-chave que indica o início do bloco do código que será executado caso a Expressão seja verdadeira.
- Corpo é o código que pode ser executado.
- *SENAO* é a palavra-chave que indica o início do bloco de código que será executado caso a Expressão seja falsa.
- Corpo é o código que pode ser executado.
- *FIM* é a palavra-chave que indica o fim da gramática.

Note que a barra vertical '|' indica a possibilidade de mais uma forma para a gramática. Nesse exemplo, existe a possibilidade da existência da palavra-chave *SENAO*, que pode mudar o comportamento do código escrito.

Veja por exemplo outra função interessante:

Algoritmo 8: Especificação da gramática "SE"

```
1 def p_expressao_logica(p):
2     """expressao_logica : expressao_simples
3                         | expressao_logica operador_logico expressao_simples
4     """
```

Nesse exemplo, a gramática tem as seguintes características:

- A gramática pode ter duas formas, sendo uma expressão simples e a outra um formato recursivo.
- O formato recursivo permite que a expressão seja expandida quantas vezes for necessário, assim todas as expressões, se gramaticalmente corretas, serão aceitas.

3.2 Formato da análise

A execução e implementação da análise é feita utilizando novamente a biblioteca *ply*, que além de trazer ferramentas para análise léxica, trouxe o *yacc* (que significa "Yet Another Compiler Compiler", um nome emprestado da ferramenta Unix). A técnica que o projeto utiliza para o processo é chamada de *LALR* (Look-Ahead LR), que é uma forma complexa de implementar o processo, mas sua eficiência é garantida em comparação em outras formas de *parsing*. A visualização desse processo é visível quando é realizada a criação de uma árvore com o resultado da análise sintática.

Código exemplo:

Algoritmo 9: Especificação da gramática "SE"

```
1 inteiro principal ()
2  inteiro: i
3  i := 0
4  i := 1 + 1
5
6 fim
```

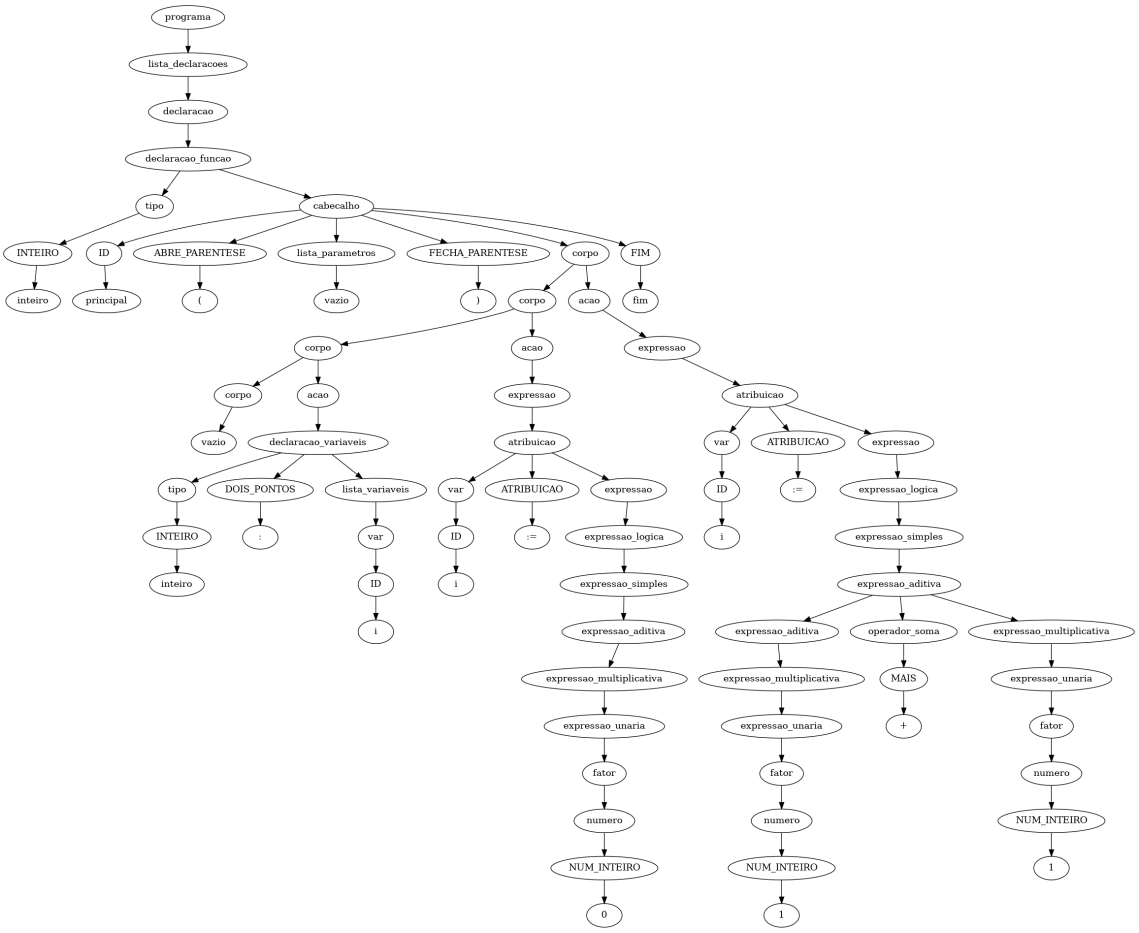

Figura 6: Resultado da Analise do código teste

Algoritmo 10: Especificação da gramática "SE"

```
1 # Build the parser.
2 parser = yacc.yacc(method="LALR", optimize=True, start='programa', debug=True,
3                     debuglog=log, write_tables=False, tabmodule='tpp_parser_tab')
```

No código apresentado acima, é possível ver a criação do parser usando a ferramenta *yacc* e a escolha do método *LALR* para sua execução. O símbolo inicial da gramática é "programa", além de outras configurações.

Desvantagens

As desvantagens do processo *LALR* incluem a complexidade da implementação à mão, que é consideravelmente maior em comparação com outras formas de implementação de *parsing*. No entanto, no caso deste projeto, a ferramenta *yacc* é importada de uma biblioteca.

3.3 Implementação

A implementação da ferramenta *yacc* foi feita importando-a da biblioteca *ply*, a mesma utilizada na análise léxica. Ela nos permite criar funções que contêm uma especificação de gramática livre de contexto apropriada e o tratamento para cada especificação, por exemplo:

Algoritmo 11: Função da especificação da gramática de soma

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #    ^            ^        ^    ^
    #  p[0]          p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

Os valores de p[i] são mapeados de acordo com os símbolos da forma apresentada acima, e o tratamento é realizado abaixo da especificação. Neste exemplo, quando a análise encontrar uma expressão de soma, p[0], p[1], p[2] e p[3] serão mapeados de acordo, e será feita a soma de p[1] e p[2] na variável p[0].

Obs: O uso de número negativo no índice tem um significado especial na ferramenta *yacc*. Normalmente, em Python, um número negativo te levaria ao final do array.

3.3.1 Combinação de regras gramáticais

Quando as regras são similares, elas podem ser combinadas em uma função só. Como exemplo, considere estas duas regras:

Algoritmo 12: Função da especificação da gramática de soma e subtração

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(t):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]
```

A forma mais prática de aplicar essas regras seria desta maneira:

Algoritmo 13: Especificação da gramática de soma e subtração na mesma função

```
def p_expression(p):
    '''expression : expression PLUS term
                  | expression MINUS term'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
```

usando condições nos valores de p[2] é possível resolver duas regras na mesma função.

Vazio

A ferramenta *yacc* consegue lidar com corpos vazios, usando uma regra simples, por exemplo:

Algoritmo 14: Função da especificação da gramática de soma

```
def p_vazio(p):
    """vazio : """
    # corpo do código
```

Erros

Ao criar um compilador, lidar com erros de sintaxe é fundamental. Em vez de simplesmente reportar o erro e encerrar a execução, é interessante que o erro seja reportado e que a análise continue, se possível. Apesar de a documentação oficial do *ply* afirmar que a recuperação de erros em analisadores *LR* é um tópico delicado, ela explica como tratar esses erros.

Veja, por exemplo uma função para tratar erros:

Algoritmo 15: Função da especificação da gramática de soma

```
1 def p_lista_argumentos_error(p):
2     """lista_argumentos : lista_argumentos error expressao
3                         | lista_argumentos VIRGULA error
4                         | error VIRGULA expressao
5         """
```

Quando a função *p_lista_argumentos* falhar, o *yacc* tentará encaixar a sintaxe em uma das gramáticas da função de erro. Se encaixar, o erro será tratado na classe *MyError* e será atribuído um código de erro para cada caso.

Código de Erro	**Significado**
ERR-SYN-USE	Uso: python tppparser.py file.tpp
ERR-SYN-NOT-TPP	Não é um arquivo .tpp.
ERR-SYN-FILE-NOT-EXISTS	Arquivo .tpp não existe.
WAR-SYN-NOT-GEN-SYN-TREE	Não foi possível gerar a Árvore Sintática.
ERR-SYN-LISTA-ARGUMENTOS	Erro na lista de Argumentos.
ERR-SYN-FATOR	Erro no fator.
ERR-SYN-LEIA	Erro no leia.
ERR-SYN-INDICE	Erro no índice.
ERR-SYN-REPITA	Erro no repita.
ERR-SYN-SE	Erro no se da gramática do se.
ERR-SYN-SE-ENTAO	Erro no então da gramática do se.
ERR-SYN-SE-FIM	Erro no fim da gramática do se.
ERR-SYN-PARAMETRO	Erro no parâmetro.
ERR-SYN-CABECALHO	Erro no Cabeçalho.

Tabela 5: Códigos de erro de sintáxe e seus significados

Nos testes, ao dar *raise* em um erro, foi notado que a criação da árvore era encerrada imediatamente. Para solucionar esse problema, foi feita uma alteração no comportamento do código. Quando um erro for detectado, o código do erro será guardado em um *array*. Assim, se o *array* não estiver vazio no final da execução, haverá um *raise* com todos os erros guardados.

Algoritmo 16: Solução para o tratamento de erro

```
1 if len(arrError) > 0:
2             raise IOError(arrError)
3     except Exception as e:
4         for i in range(len(e.args[0])):
5             print(e.args[0][i])
```

E a implementação dos erros nas funções serão dessa maneira:

Algoritmo 17: Solução para o tratamento de erro

```
1 def p_fator_error(p):
2     """fator : ABRE_PARENTESE error FECHA_PARENTESE
3              | error expressao FECHA_PARENTESE
```

```
4               | ABRE_PARENTESE expressao error
5       """
6
7       pai = MyNode(name='ERR-SYN-FATOR', type='ERROR')
8       p[0] = pai
9
10      global arrError
11      global showKey
12
13      arrError.append(error_handler.newError(showKey, 'ERR-SYN-FATOR'))
```

Dessa forma a árvore pode ser construida e o erro é declarado normalmente.

3.4 Arvore Sintática

O objetivo da análise sintática, além da verificação da integridade da gramática do código, é também a criação da árvore sintática. A partir da leitura dos *tokens*, uma árvore com a estrutura do código é criada, que pode ser usada futuramente pelo compilador. Ela fornece a estrutura e a associatividade das operações que a cadeia original não mostra. A ferramenta *yacc* da biblioteca *ply* não tem uma função especial para criação de árvores, mas sua documentação mostra maneiras fáceis de implementá-la. A árvore criada nesse projeto utiliza a biblioteca *anytree*, uma biblioteca própria para criação, manipulação e visualização de árvores.

Cada função foi tratada para que a árvore seja criada, por exemplo, esta função que tem a gramática.

Algoritmo 18: Função da especificação da gramática de soma

```
1  def p_cabecalho(p):
2      """cabecalho : ID ABRE_PARENTESE lista_parametros FECHA_PARENTESE corpo FIM
           """
3
4      pai = MyNode(name='cabecalho', type='CABECALHO')
5      p[0] = pai
6
7      filho1 = MyNode(name='ID', type='ID', parent=pai)
8      filho_id = MyNode(name=p[1], type='ID', parent=filho1)
9      p[1] = filho1
10
11     filho2 = MyNode(name='ABRE_PARENTESE', type='ABRE_PARENTESE', parent=pai)
12     filho_sym2 = MyNode(name='(', type='SIMBOLO', parent=filho2)
13     p[2] = filho2
14
15     p[3].parent = pai  # lista_parametros
16
17     filho4 = MyNode(name='FECHA_PARENTESE', type='FECHA_PARENTESE', parent=pai)
18     filho_sym4 = MyNode(name=')', type='SIMBOLO', parent=filho4)
19     p[4] = filho4
20
21     p[5].parent = pai  # corpo
22
23     filho6 = MyNode(name='FIM', type='FIM', parent=pai)
24     filho_id = MyNode(name='fim', type='FIM', parent=filho6)
25     p[6] = filho6
```

Cada *Mynode* é mais uma "folha"da árvore. Veja, por exemplo, uma imagem da árvore com o cabeçalho presente:

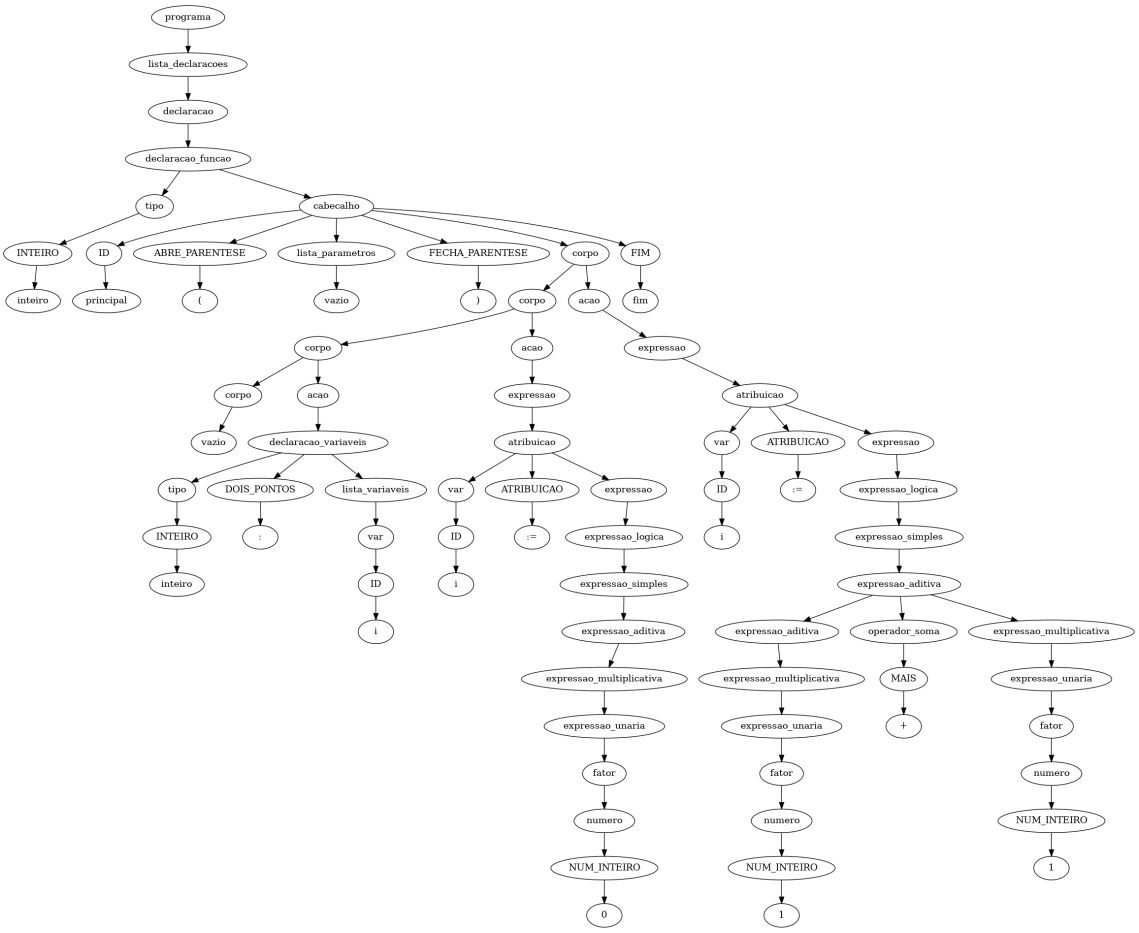

Figura 7: Árvore com um cabeçalho

Tratamento de erros

Nas análises, erros de sintaxe podem ser encontrados, incluindo erros de gramática e estrutura. Esses erros têm impacto na árvore e na sua estrutura. Por exemplo:

Algoritmo 19: Código de entrada

```
1  inteiro: a
2  inteiro: b
3  inteiro: c[]
4  flutuante: d[10][]
5  flutuante: e[1024][
6
7  inteiro principal()
8      leia(a)
9      escreva(b)
10 fim
```

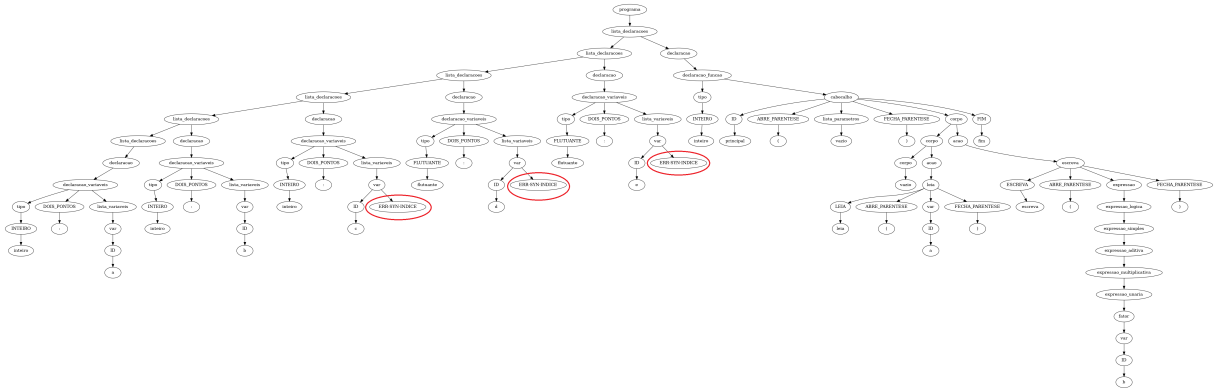

Figura 8: Saída, com erro de índice

Ao realizar a leitura, a análise identifica o erro e coloca um código no local onde ele foi encontrado, facilitando a busca pelo erro. No exemplo anterior, entre o *token* de abre e fecha parênteses, há o código "*ERR-SYN-LISTA-ARGUMENTOS*", que indica o erro.

4 Análise Semântica

A análise semântica é uma etapa essencial no processo de compilação, realizada logo após a análise sintática. Enquanto a análise sintática foca na estrutura gramatical do programa, garantindo a conformidade com as regras da linguagem, a análise semântica vai além, verificando a correção lógica e a coerência dos elementos do código.

Durante essa fase, o compilador examina as declarações e o uso de variáveis, funções e tipos. Por exemplo, ele verifica se as variáveis foram declaradas antes de serem usadas, se os tipos de dados são compatíveis em atribuições e operações, e se as funções são chamadas com o número e tipo corretos de argumentos.

No contexto da linguagem de programação TPP, a análise semântica é crucial para detectar erros que poderiam passar despercebidos em etapas anteriores. Implementar essa fase envolve criar estruturas como tabelas semânticas e navegar pela árvore sintática gerada na análise anterior, garantindo que o programa siga as regras semânticas da linguagem.

Em resumo, a análise semântica é fundamental para assegurar que o código não só esteja correto gramaticalmente, mas também faça sentido logicamente, prevenindo erros que poderiam comprometer o funcionamento do programa.

4.1 Estratégias utilizadas para a realização da Análise Semântica

A análise semântica é uma etapa sofisticada do processo de compilação, essencial para garantir que o código-fonte siga as regras lógicas e semânticas da linguagem. Diferente da análise sintática, que foca na estrutura do código, e da análise léxica, que se concentra na identificação dos tokens, a análise semântica se preocupa com o significado dos elementos, assegurando a coerência e a correção lógica das operações.

Uma das principais estratégias na análise semântica é a navegação pela árvore sintática, gerada durante a análise sintática, que serve como uma representação intermediária do programa. Durante a análise semântica, essa árvore é percorrida para verificar a correção das expressões e instruções do código. Cada nó da árvore corresponde a um elemento do programa, como uma operação aritmética, uma chamada de função ou uma declaração de variável. Ao navegar pela árvore, o compilador realiza verificações de tipo, assegura que as operações entre diferentes tipos de dados sejam válidas e confirma que as variáveis e funções estejam sendo usadas dentro de seus escopos apropriados.

Essa navegação pela árvore sintática também auxilia na construção e utilização de uma tabela de símbolos, outra estratégia fundamental na análise semântica. Por exemplo, essa tabela armazena informações sobre variáveis, funções e tipos, associando a cada um deles atributos como tipo, escopo e endereço na memória. Durante a análise, a tabela de símbolos é consultada e atualizada constantemente para verificar se as declarações e usos dos identificadores estão corretos, ajudando a detectar variáveis usadas antes de serem declaradas ou funções chamadas com o número incorreto de argumentos.

Token	Lexema	Tipo	dim	tam_dim1	tam_dim2	escopo	init	linha
ID	"a"	int	0	1	0	global	N	1
ID	"b"	int	1	10	0	global	N	1
ID	"c"	int	2	3	5	global	N	1

Figura 9: Exemplo de Tabela Semântica

A verificação de escopo e a gestão de contextos também são fundamentais. A análise semântica deve garantir que variáveis e funções sejam acessadas apenas dentro de seus respectivos escopos. Para isso, o compilador precisa manter um controle cuidadoso dos contextos em que cada identificador é válido, especialmente em linguagens que suportam escopo aninhado. Isso foi implementado utilizando uma pilha de tabelas de símbolos, onde cada *ID* tem um atributo de escopo que mostra de onde vem este *ID*.

Finalmente, a gestão de tipos e coerção de tipos é essencial para garantir a segurança e correção do código. O compilador deve assegurar que as operações sejam realizadas entre tipos compatíveis ou, quando permitido pela linguagem, aplicar coerção de tipos para converter automaticamente entre tipos diferentes de forma segura. A verificação de tipos previne erros como a adição de uma variável do tipo inteiro com uma string, e a coerção de tipos permite operações seguras, como a conversão de um inteiro para um ponto flutuante antes de uma divisão.

Em resumo, essas são as estratégias que são utilizadas na análise semântica do projeto.

4.2 Especificações da análise

A primeira coisa necessária ao criar um algoritmo de análise semântica é decidir o escopo dele, ou seja, determinar o que ele deve analisar. Em resumo, o escopo, nesse projeto, vai ser dividido em funções e procedimentos, variáveis, atribuição de tipos distintos e coerções implícitas, e arranjos, separando em tipos de erro e avisos.

Funções e Procedimentos

Verificações relacionadas a funções e procedimentos.

1. **Função Principal:** Todo programa escrito em TPP deve ter uma função principal declarada que inicializa a execução do código. Verificar a existência de uma função com o nome de `principal`, e se não existir, deve apresentar a seguinte mensagem:
   - Tipo: Erro.
   - Chave: `ERR-SEM-MAIN-NOT-DECL`.
2. **Declaração de Função:** Funções precisam ser declaradas antes de serem chamadas. Caso contrário, a seguinte mensagem de erro deve ser emitida:
   - Tipo: Erro.
   - Chave: `ERR-SEM-CALL-FUNC-NOT-DECL`.
3. **Identificador de Função:** Uma função ou procedimento é identificado pelo seu tipo de retorno, seu nome e seus parâmetros formais. A quantidade de parâmetros reais de uma

chamada de função/procedimento `func` deve ser igual à quantidade de parâmetros formais da sua definição. Caso contrário, gerar as seguintes mensagens:
   - Tipo: Erro.
   - Chave: `ERR-SEM-CALL-FUNC-WITH-FEW-ARGS`
   - Chave: `ERR-SEM-CALL-FUNC-WITH-MANY-ARGS`.
4. **Declarada e não utilizada:** Uma função pode ser declarada e não utilizada. Se isto acontecer, um aviso deverá ser emitido:
   - Tipo: Aviso.
   - Chave: `WAR-SEM-FUNC-DECL-NOT-USED`.
   - Mensagem: Função `func` declarada, mas não utilizada.
5. **Retorno de Função:** Uma função deve retornar um valor de tipo compatível com o tipo de retorno declarado. A função principal normalmente é do tipo inteiro, então é esperado que seu retorno seja um valor inteiro. Se não apresentar um retorno `return(0)`, a seguinte mensagem deve ser gerada:
   - Tipo: Erro.
   - Chave: `ERR-SEM-FUNC-RET-TYPE-ERROR`.

   Uma função qualquer não pode fazer uma chamada à função principal. Devemos verificar se existe alguma chamada para a função principal partindo de qualquer outra função do programa.
   - Tipo: Erro.
   - Chave: `ERR-SEM-CALL-FUNC-MAIN-NOT-ALLOWED`.

   Se a função principal fizer uma chamada para ela mesma, a seguinte mensagem de aviso deve ser emitida:
   - Tipo: Aviso.
   - Chave: `WAR-SEM-CALL-REC-FUNC-MAIN`.

Variáveis

1. **Identificador de Variáveis Locais e Globais:** O nome, tipo e escopo das variáveis devem ser armazenados na Tabela de Símbolos. Variáveis devem ser declaradas e inicializadas antes de serem utilizadas (leitura). Lembrando que uma variável pode ser declarada:
   - No escopo do procedimento (como expressão ou como parâmetro formal).
   - No escopo global.

   Se houver a tentativa de leitura ou escrita de qualquer variável não declarada, a seguinte mensagem deve ser gerada:
   - **Tipo:** Erro.
   - **Chave:** `ERR-SEM-VAR-NOT-DECL`.
2. **Variável Declarada e Não Inicializada:** Se uma variável `'a'` for apenas declarada e não inicializada (escrita), e houver a tentativa de leitura dessa variável, a seguinte mensagem de aviso deve ser gerada:
   - **Tipo:** Aviso.
   - **Chave:** `WAR-SEM-VAR-DECL-NOT-INIT`.
3. **Variável Declarada e Não Utilizada:** Se uma variável `'a'` for declarada, mas não utilizada (não lida), o analisador deve gerar a seguinte mensagem de aviso:
   - **Tipo:** Aviso.
   - **Chave:** `WAR-SEM-VAR-DECL-NOT-USED`.
4. **Variável Declarada Mais de uma Vez:** Warnings devem ser mostrados quando uma variável for declarada mais de uma vez. Se uma variável `'a'` for declarada duas vezes no mesmo escopo, a seguinte mensagem de aviso deve ser emitida:
   - **Tipo:** Aviso.

- **Chave:** `WAR-SEM-VAR-DECL-PREV`.

Atribuição de Tipos Distintos e Coerções Implícitas

1. **Atribuição de Tipos Distintos e Coerções Implícitas:** Na atribuição, devem ser verificados se os tipos são compatíveis. Warnings deverão ser mostrados quando ocorrer uma coerção implícita de tipos (inteiro ↔ flutuante).
2. Atribuição de variáveis, números, resultados de chamadas a funções ou resultados de expressões de tipos distintos devem gerar a seguinte mensagem:
   - **Tipo:** Aviso.
   - **Chave:** `WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-VAR`.
3. Atribuição de números de tipos distintos deve gerar a seguinte mensagem:
   - **Tipo:** Aviso.
   - **Chave:** `WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-NUM`.
4. Por exemplo, se uma variável `'a'` recebe uma expressão `'b + c'`, os tipos declarados para `'a'` e o tipo resultante da inferência do tipo da expressão `'b + c'` deverão ser compatíveis. Se `'b'` for inteiro e `'c'` for inteiro, o tipo resultante da expressão será também inteiro. Se `'b'` for do tipo inteiro e `'c'` for do tipo flutuante, o resultado pode ser flutuante (dependendo da especificação da linguagem), o que faria a atribuição `'a := b + c'` apresentar tipos diferentes e gerar a seguinte mensagem:
   - **Tipo:** Aviso.
   - **Chave:** `WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-EXP`.
5. O mesmo pode acontecer com a atribuição do retorno de uma função. Se os tipos forem incompatíveis, o usuário deve ser avisado com a seguinte mensagem:
   - **Tipo:** Aviso.
   - **Chave:** `WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-RET-VAL`.
6. Se algum argumento de uma chamada de função for de tipo diferente do tipo declarado, deve ser gerado um aviso com a seguinte mensagem:
   - **Tipo:** Aviso.
   - **Chave:** `WAR-SEM-ATR-DIFF-TYPES-IMP-COERC-OF-FUNC-ARG`.

Arranjos

1. **Declaração de Arranjos:** Na linguagem TPP, é possível declarar arranjos. Pela sintaxe da linguagem, o índice de um arranjo deve ser inteiro e isso deve ser verificado. Na Tabela de Símbolos, devemos armazenar se uma variável declarada tem um tipo e se é uma variável escalar, um vetor ou uma matriz. Podemos armazenar um campo `'dimensões'`, onde:
   - `'0'`: Escalar.
   - `'1'`: Arranjo unidimensional (vetor).
   - `'2'`: Arranjo bidimensional (matriz).
   - `'n'`: Arranjo `n`-dimensional.
2. **Verificação do Índice de Arranjos:** Ao encontrar a referência a um arranjo, o índice, seja ele um número, variável ou expressão, deve ser um inteiro. Caso contrário, a seguinte mensagem deve ser gerada:
   - **Tipo:** Erro.
   - **Chave:** `ERR-SEM-ARRAY-INDEX-NOT-INT`.
3. **Verificação de Acesso Fora do Intervalo:** Se o acesso ao elemento do arranjo estiver fora de sua definição, por exemplo, um vetor `'A'` é declarado como tendo 10 elementos (índices de 0 a 9) e houver um acesso a `A[10]`, a seguinte mensagem de erro deve ser apresentada:
   - **Tipo:** Erro.

- **Chave:** `ERR-SEM-ARRAY-INDEX-OUT-OF-RANGE`.

4.3 Geração da tabela semântica

Agora que sabemos o que deve ser verificado, podemos implementar a análise semântica. Iremos criar um algoritmo que analise a árvore sintática e construa uma Tabela de Símbolos semântica. Esta tabela conterá os seguintes dados para cada entrada:

- **Declaration:** O tipo de declaração, que pode ser uma declaração de variável, função, chamada de função, entre outros.
- **Type:** O tipo da variável, como `inteiro` ou `flutuante`.
- **ID:** O nome da variável ou função.
- **Scope:** O escopo da variável ou função, que pode ser `global` ou dentro de um procedimento.
- **Data:** Utilizado para armazenar parâmetros de funções ou variáveis usadas em uma atribuição.

Usando uma classe para facilitar o uso de palavras-chave, criamos uma função chamada *creatingSemanticTable*, que tem como parâmetro uma árvore da biblioteca Python *anytree*. Essa árvore será explorada para a criação da Tabela de Símbolos semântica. Começamos com um loop que percorre toda a árvore, procurando por palavras-chave como:

- `declaracao_funcao`
- `declaracao_variaveis`
- `atribuicao`
- `chamada_funcao`
- `FIM`
- `retorna`
- `fator`
- `parametro`
- `var`

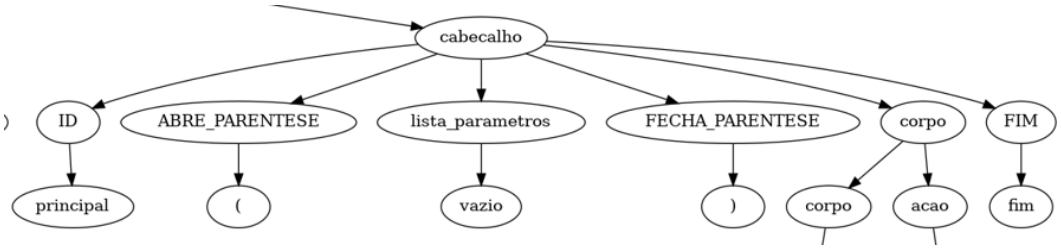

Figura 10: Pedaço da arvore semântica

Neste exemplo, ao encontrar `declaracao_funcao`, o código explora os nós seguintes para encontrar o seu tipo, o ID, e os parâmetros. O código fica da seguinte maneira:

Algoritmo 20: Trecho de código que le os dados importantes da declaração

```
1         # Declaração de Função
2         if hasattr(node, 'name') and node.name == PalavrasChaves.
              declaracao_funcao.value:
3
4             #Pega o tipo
5             nodeAux = node.children[0]
6             nodeAux = nodeAux.children[0]
7             type = nodeAux.name
8
9             #Pega o nome
10            nodeAux = node.children[1]
```

```
11              nodeAux = nodeAux.children[0]
12              nodeAux = nodeAux.children[0]
13              name = nodeAux.name
14
15              nodeAux = node.children[1]
16              nodeAux = nodeAux.children[2]
17              data = []
18
19              scopeAux = scope
20              scope = name
21              find_parameters(nodeAux, semTable, scope, data)
22
23              scope = scopeAux
24              semTable.append({"declaration": node.name, "type": type, "id": name,
                    "scope": scope, "data": data})
25
26              scope = name
```

Note que a variável *scope* fica salva com o nome da função, para que a leitura dos nós seguintes do corpo da função esteja com o escopo anotado.

O resultado desse trecho é aproximadamente assim:

```
{'data': [], 'declaration': 'declaracao_funcao',
'id': 'principal', 'scope': None, 'type': 'INTEIRO'},
```

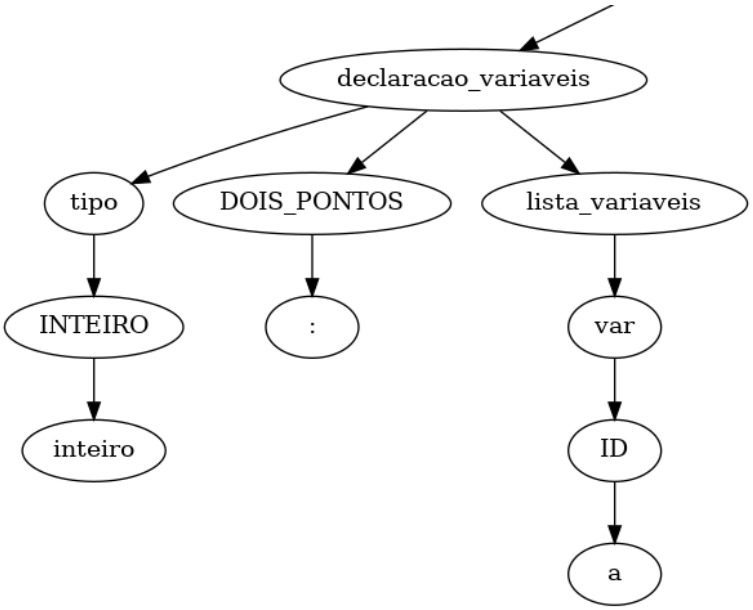

Figura 11: Pedaço da arvore semântica

Neste exemplo, ao encontrar o nó com o nome *"declaracao_variavel"*, o código percorre o padrão de nós para obter o tipo e o ID da variável. O código utilizado para a leitura desses nós é o seguinte:

Algoritmo 21: Trecho de código que lê os dados importantes da declaração

```
1  # Declaração de Variável
2  if hasattr(node, 'name') and node.name == PalavrasChaves.declaracao_variaveis.
       value:
3
4       nodeAux = node.children[0]
5       nodeAux = nodeAux.children[0]
6       type = nodeAux.name
```

```
7
8       nodeAux = node.children[2]
9       nodeAux = nodeAux.children[0]
10      nodeAux = nodeAux.children[0]
11      nodeAux = nodeAux.children[0]
12      name = nodeAux.name
13
14      semTable.append({"declaration": node.name, "type": type, "id": name, "scope"
            : scope})
```

O resultado desse código é aproximadamente assim:

```
{'declaration': 'declaracao_variaveis', 'id': 'x',
'scope': 'func', 'type': 'INTEIRO'},
{'declaration': 'declaracao_variaveis', 'id': 'y',
'scope': 'func', 'type': 'INTEIRO'}
```

Um outro exemplo interessante é o tratamento para a palavra-chave *FIM*. Embora ela não seja necessária para a tabela semântica, a busca dele é crucial para definir corretamente os escopos das variáveis. Esse trecho de código existe apenas para esse propósito.

Algoritmo 22: Trecho de código que define o escopo das variáveis com base na palavra-chave FIM

```
1 if hasattr(node, 'name') and node.name == PalavrasChaves.fim.value:
2     scope = None
```

Na tabela semântica, quando o valor de *scope* é `None`, isso significa que o escopo da variável é global.

Outro exemplo é o de atribuição, onde um *ID* é atribuído, e outros valores em uma possível expressão são utilizados para determinar o valor atribuído. O problema é que a atribuição deve ser bem documentada na tabela semântica, pois ela pode se encaixar em vários possíveis erros semânticos, como variáveis chamadas não inicializadas ou não declaradas, ou erros de atribuição de tipos distintos. A solução adotada é que, no campo *data*, sejam armazenadas as variáveis da expressão e seus tipos (caso a atribuição tenha um valor direto).

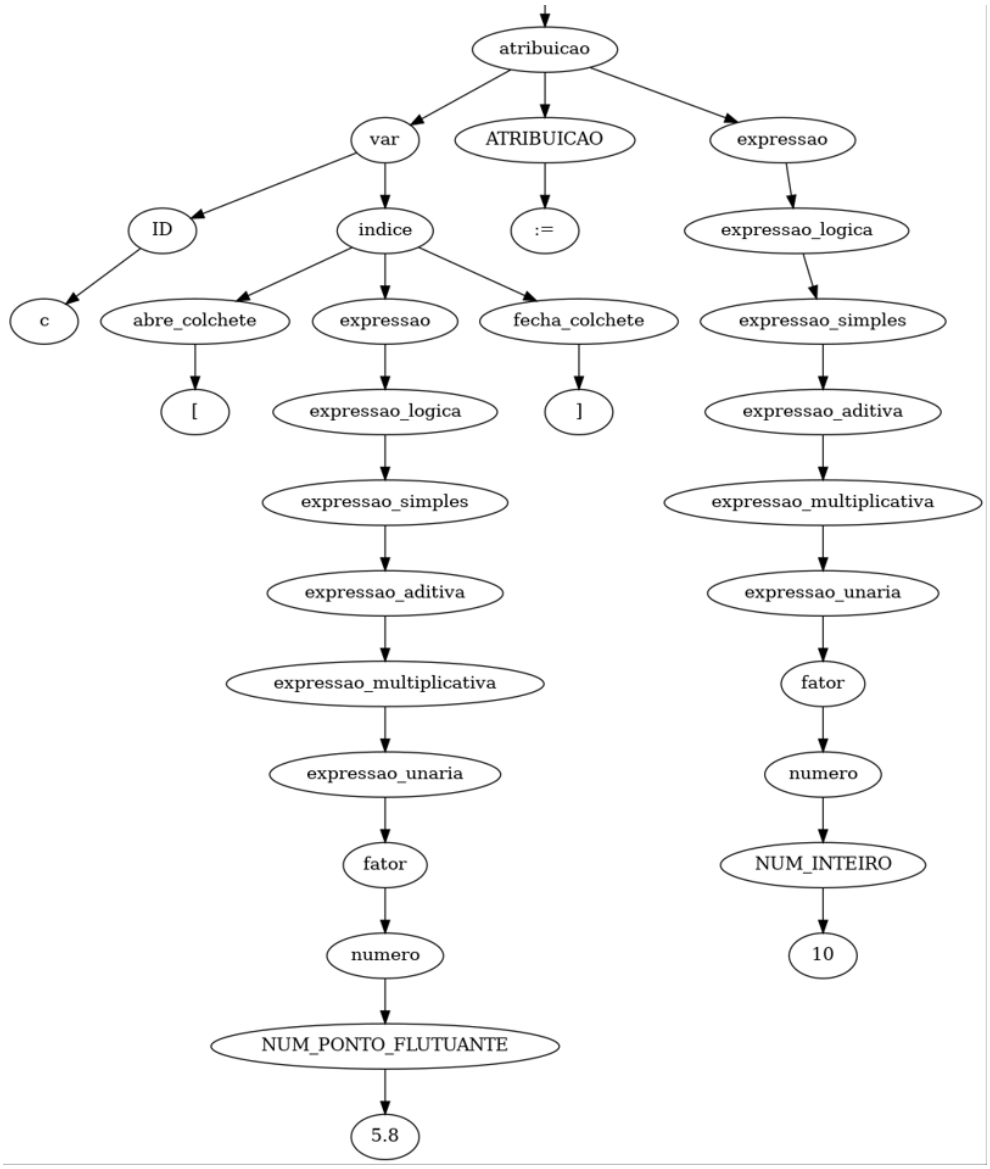

Figura 12: Pedaço da arvore sintática

O código responsável por documentar os dados de atribuições é o seguinte:

Algoritmo 23: Trecho de código que documenta atribuições na tabela semântica

```
1         # Atribuição
2         if hasattr(node, 'name') and node.name == PalavrasChaves.atribuicao.
              value:
3
4             nodeAux = node.children[0]
5             nodeAux = nodeAux.children[0]
6             nodeAux = nodeAux.children[0]
7             name = nodeAux.name
8
9             data = find_ID_and_factor(node)
10
11            #print(f"nodeAux = {nodeAux.name}")
12
13            semTable.append({"declaration": node.name, "type": '-', "id": name,
                  "scope": scope, "data": data})
```

O resultado é algo parecido com isso:

```
{'declaration': 'atribuicao', 'id': 'c',
```

```
'scope': 'principal', 'data': ['NUM_INTEIRO']},
```

Seguindo esses exemplos, o código é analisado e documentado em uma tabela semântica. Quando a tabela semântica estiver pronta, passamos para o próximo passo, onde verificamos o contexto, fazemos a gestão dos escopos e dos tipos das variáveis, para garantir que o código esteja semanticamente correto.

4.4 Finalizando análise semântica

No projeto, foi implementada a função *checkingTable(semTable)*, que executa todas as funções responsáveis por verificar a tabela semântica e realizar cada uma das especificações mencionadas anteriormente. A seguir, apresentamos um exemplo de uma função de verificação:

Algoritmo 24: Trecho de código que verifica a existência da função principal na tabela semântica

```
1  def verificarFuncaoPrincipal(semTable):
2      global arrError
3      global showKey
4      principal_existe = False
5      for table in semTable:
6          if table['declaration'] == 'declaracao_funcao' and table['id'] == '
               principal':
7              principal_existe = True
8              break
9      if not principal_existe:
10         arrError.append(error_handler.newError(showKey, 'ERR-SEM-MAIN-NOT-DECL')
               )
```

Essa função verifica na tabela semântica se existe uma declaração de função com um ID chamado *principal*. Caso contrário, um erro é adicionado a um array de erros, que servirá como uma possível saída do código.

Saída

Após todas as verificações, o algoritmo de análise verifica se o vetor responsável por armazenar a lista de erros contém algum conteúdo. Caso haja um erro ou aviso, ele será impresso ao final da execução. Abaixo estão alguns exemplos de códigos e suas respectivas saídas.

Algoritmo 25: Entrada

```
1  inteiro: a
2  flutuante: b
```

Este código simples como entrada resultará nas seguintes saídas:

```
{Erro: Função 'principal' não declarada}
{Aviso: Variável 'a' declarada e não utilizada}
{Aviso: Variável 'b' declarada e não utilizada}
```

Ao utilizar a flag ‘-k‘, como mencionado anteriormente, as saídas são transformadas em um formato de fácil entendimento para automatização, resultando em saídas como:

```
ERR-SEM-MAIN-NOT-DECL
WAR-SEM-VAR-DECL-NOT-INIT
WAR-SEM-VAR-DECL-NOT-INIT
```

Com a análise feita, temos a verificação que o código análisado está pronto para ser o próximo passo do compilador *TPP*, que é a geração de código.

5 Geração de Código

Na etapa de geração de código em um compilador, o objetivo é transformar a representação intermediária, obtida após as análises sintática e semântica, em um código de saída que possa ser executado diretamente pela máquina ou interpretado por um ambiente de execução. A saída esperada nesta fase pode ser um relatório de erros resultantes das análises anteriores ou um código LLVM, gerado por meio da biblioteca Python `llvmlite`.

A prioridade principal desta etapa é garantir que o código gerado seja funcional e correto. Em segundo plano, com menor prioridade, estão aspectos como a eficiência da tradução, o gerenciamento de memória e a otimização do código. A utilização da `llvmlite` facilita esse processo ao fornecer uma interface para a geração de código intermediário, permitindo que o compilador foque na geração de um código confiável e utilizável.

Código LLVM

Já que o resultado final desejado do nosso compilador é um código LLVM, é essencial entender o que ele representa. O *LLVM* (*Low-Level Virtual Machine*) é uma representação intermediária (IR), que atua como uma camada intermediária entre o código em alto nível e o código de máquina específico de cada plataforma. Essa representação pode ser utilizada em várias plataformas de hardware, tornando o processo de compilação altamente portátil.

Sua principal característica é a flexibilidade e a capacidade de otimização independente do hardware de destino. O código LLVM pode ser traduzido para o código de máquina específico em uma etapa posterior, adaptando-se facilmente a diferentes arquiteturas.

Por fim, a biblioteca `llvmlite` em Python permite a criação e manipulação dessa IR, facilitando significativamente o trabalho de quem está encarregado da geração de código, tornando o processo mais acessível e eficiente.

5.1 Implementação

Nesta etapa, a implementação básica, resumidamente, será uma tradução da árvore sintática para código LLVM. O professor GONÇALVES (2016) da disciplina recomendou podar a árvore sintática para eliminar ”galhos”desnecessários, como ilustrado na figura 13:

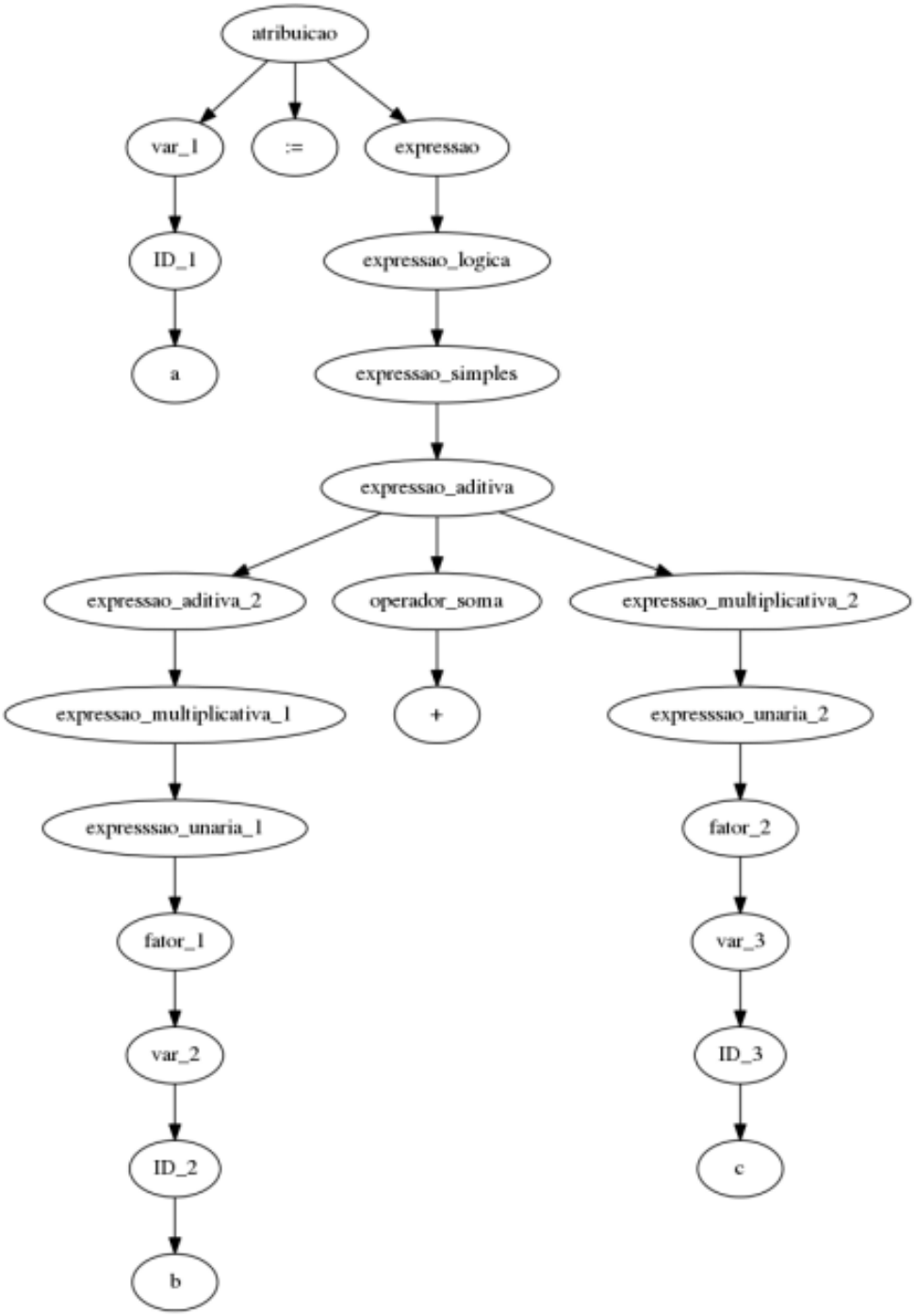

Figura 13: Pedaço da árvore sintática com atribuição não podada

A quantidade de galhos insignificativos é considerável, e o objetivo é que a árvore fique desta forma da figura 14:

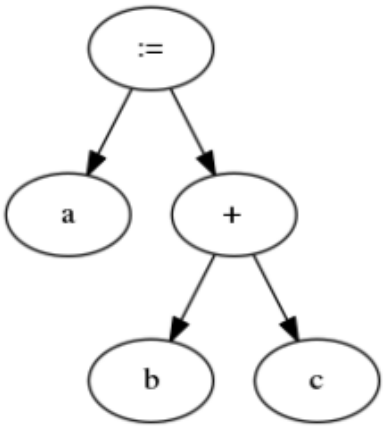

Figura 14: Árvore sintática com atribuição podada

Este seria o ideal. No entanto, durante os testes, a poda da árvore se mostrou um processo mais complicado do que o previsto, pois a biblioteca *anytree* não permite a atribuição direta dos filhos de cada nó. Quando tentávamos substituir os nós, a estrutura da árvore acabava se desorganizando.

Diante dessas dificuldades, optamos por explorar a árvore sem realizar a poda, para evitar desordens e manter a estrutura original durante o processo de geração de código.

5.1.1 Leitura da Arvore Sintática

Como mencionado anteriormente, a ideia é realizar uma tradução da árvore sintática. Ao ler os nós contendo palavras-chave específicas, será feita uma tradução apropriada de acordo com o que foi lido. Utilizando o repositório do professor GONÇALVES (2024), fica claro o que cada elemento da árvore representa em termos das funções da biblioteca *LLVM*.

A estrutura básica da leitura da árvore é mostrada no código 26.

Algoritmo 26: Estrutura base do algoritmo de leitura da árvore

```
1  for node in PreOrderIter(tree):
2
3          if node.name == "PALAVRA CHAVE":
4              # TRADUÇÃO DE ACORDO COM A PALAVRA CHAVE
```

Com essa estrutura base definida, o próximo passo é aumentar as capacidades da geração de código para que ele possa traduzir todas as construções suportadas pela linguagem *TPP*. Por exemplo, vamos tratar a palavra-chave *"declaracao_funcao"* no código 27.

Algoritmo 27: Tratamento para palavra chave "declaracao_funcao"

```
1  if(node.name == "declaracao_funcao"):
2
3      name = browseNode(node, [1,0,0]).name
4      #FUNÇÃO PRINCIPAL TEM QUE SE CHAMAR 'main'
5      #POR ISSO IREMOS TROCAR QUANDO NOME É principal OU main
6      if(name == 'principal'):
7          name = 'main'
8      elif(name == 'main'):
9          name = 'principal'
10
11     scope = name
12     type = browseNode(node, [0,0,0]).name
13
14     var = createTypeVar(type)
```

```
15
16      varNameList, typeList = verifyParams(node, name)
17
18      functInfo = ir.FunctionType(var, typeList)
19      func = ir.Function(module, functInfo, name=name)
20      funcList.append({"name": name, "func": func})
21
22      for i in range(len(varNameList)):
23          func.args[i].name = varNameList[i]
24          #print(func.args[i], " : ", func.args[i].name)
25      #print(functInfo, name, var, type)
26      entryBlock = func.append_basic_block('entry')
27
28
29              builder = ir.IRBuilder(entryBlock)
```

O código 27 tem a função de declarar uma função, armazená-la em uma lista para futuras chamadas e iniciar um bloco de código, conforme previsto na árvore sintática. Além disso, ele verifica se a árvore possui parâmetros e os adiciona no código assembly. Também garante que a função chamada "principal"seja referenciada como a função `main` no código LLVM.

No entanto, algumas funções implementadas exigem mais do que uma simples tradução. Por exemplo, as funções `Leia` e `Escreva` da linguagem TPP, que são equivalentes a `scanf` e `printf` em C, têm uma implementação mais complexa. Isso ocorre porque `Leia` e `Escreva` são implementadas como funções externas e precisam ser declaradas antes de qualquer chamada.

Isso forçou uma leitura prévia do código para verificar se `Leia` e `Escreva` são utilizadas no código de entrada. Além disso, as funções externas de leitura e escrita são divididas em duas versões cada: uma para inteiros e outra para flutuantes. Assim, foi necessário verificar o tipo da variável que estava sendo passada como argumento. O resultado dessa verificação está mostrado no Código 28.

Algoritmo 28: Trecho que verifica se Leia e Escreva foram utilizado na código de entrada

```
1  for node in (PreOrderIter(tree)):
2
3      #print(node.name)
4      if(node.name == "declaracao_funcao"):
5          scope = browseNode(node, [1,0,0]).name
6
7      if(node.name == "declaracao_variaveis"):
8          ADDAnotherVarInList(node,scope,list)
9
10     if(node.name == "leia" and len(node.children) > 1):
11         #por enquanto ta com nome da variavel
12         nameVar = browseNode(node,[2,0,0]).name
13         type = getTypeInList(nameVar, scope, list)
14
15         if(type in flutuante and not haveReadFloat):
16             _leiaF = ir.FunctionType(ir.FloatType(), [])
17             leiaF = ir.Function(module, _leiaF, "leiaFlutuante")
18             haveReadFloat = True
19
20         if(type in inteiro and not haveReadInt):
21             _leiaI = ir.FunctionType(ir.IntType(32), [])
22             leiaI = ir.Function(module, _leiaI, "leiaInteiro")
23             haveReadInt = True
24
25     if(node.name == "escreva" and len(node.children) > 1):
26
27         nodeAux = browseNode(node, [2])
```

```
28          type = findFirstTypeVar(nodeAux,list,scope)
29          if(type in flutuante and not havePrintFloat):
30              _escrevaF = ir.FunctionType(ir.VoidType(), [ir.FloatType()])
31              escrevaF = ir.Function(module, _escrevaF, "escrevaFlutuante")
32              havePrintFloat = True
33
34          if(type in inteiro and not havePrintInt):
35              _escrevaI = ir.FunctionType(ir.VoidType(), [ir.IntType(32)])
36              escrevaI = ir.Function(module, _escrevaI, "escrevaInteiro")
37              havePrintInt = True
```

Expressões

Durante as implementações, desenvolvemos uma função chamada `expression`. Esta função é crucial para o código, pois, ao ser implementada corretamente, simplifica significativamente a tradução, resolvendo-a em poucas linhas. Com a função `expressao`, a implementação de loops, condições e atribuições torna-se mais direta. O Código 29 mostra como foi realizado o tratamento de expressões.

Algoritmo 29: Tratamento de expressões do código de entrada

```
1       for node in (PreOrderIter(nodeE)):
2
3           # PARA NAO LER FATORES DAS CHAMADAS DE FUNÇÃO
4           if any(parent in ignore_nodes for parent in node.ancestors):
5               continue
6
7           # Se o nó atual for 'chamada_funcao' ou 'lista_argumentos', adiciona ao
                conjunto
8           if node.name in ["chamada_funcao", "lista_argumentos"]:
9               #print(f"Ignorando o nó e seus filhos: {node.name}")
10              ignore_nodes.add(node)
11              continue
12
13          #print(node.name)
14          if node.name == "fator":
15              #nodeAux = browseNode(node, [0,0])
16
17              if(browseNode(node, [0]).name == 'chamada_funcao'):
18
19                  nodeAux = browseNode(node, [0,0,0])
20                  func = getFuncInList(nodeAux.name)
21
22                  nodeAux = browseNode(node, [0,2])
23                  if(x_temp == None):
24                      x_temp = builder.call(func, giveArgsList(nodeAux, scope))
25                  else:
26                      y_temp = builder.call(func, giveArgsList(nodeAux, scope))
27                  #print(x_temp, y_temp, expression)
28
29              elif(browseNode(node, [0,0]).name == "ID"):
30                  #CASO SEJA UM ID
31                  nodeAux = browseNode(node, [0,0,0])
32                  if(x_temp == None):
33                      var = getVarInList(nodeAux.name, scope)
34                      # SE NAO FOR PARAMETRO
35                      if isinstance(var, ir.instructions.AllocaInstr) or
                            isinstance(var, ir.values.GlobalVariable):
36                          x_temp = builder.load(var, name='x_temp')
```

```
37                      else:
38                          x_temp = var
39
40                  else:
41                      var = getVarInList(nodeAux.name, scope)
42                      # SE NAO FOR PARAMETRO
43                      if isinstance(var, ir.instructions.AllocaInstr) or
                            isinstance(var, ir.values.GlobalVariable):
44                          y_temp = builder.load(var, name='y_temp')
45                      else:
46                          y_temp = var
47
48                      #print(x_temp, y_temp, expression)
49                      x_temp = expressionsAux(x_temp, y_temp, expression)
50                      y_temp = None
51                      expression = None
52              elif(browseNode(node, [0,0]).name in flutuante or browseNode(node,
                    [0,0]).name in inteiro):
53                  #CASO SEJA UMA CONSTANTE
54                  nodeAux = browseNode(node, [0,0])
55                  varType = createTypeVar(nodeAux.name)
56                  nodeAux = browseNode(nodeAux, [0])
57                  if(x_temp == None):
58                      x_temp = ir.Constant(varType,float(nodeAux.name))
59                  else:
60                      y_temp = ir.Constant(varType,float(nodeAux.name))
61                      x_temp = expressionsAux(x_temp, y_temp, expression)
62                      y_temp = None
63                      expression = None
64
65
66          if(node.name in sinais_aritmeticos or node.name in sinais_logicos):
67              expression = node.name
68              #print(node.name)
69              #TODO : TRATAR PARENTESES COM RECURSIVIDADE !!!!!!
70
71
72      return(x_temp)
```

Esta função, que lida com expressões de qualquer tamanho, é responsável por tratar variáveis globais, constantes e chamadas de função. Ela realiza essa tarefa ao ler a árvore onde a expressão se origina. Durante a execução, manipula registradores temporários, como `x_temp` e `y_temp`, para cálculos e manipulação da variável `operador`, que indica a operação aritmética ou lógica a ser realizada.

Durante os testes, descobrimos que a função estava detectando variáveis das expressões dos argumentos usados nas chamadas de função, resultando em um comportamento indesejado. Para resolver isso, fizemos uma adição antes da leitura da árvore, conforme mostrado no Código 30.

Algoritmo 30: Trecho de código que impede que as variaveis da expressão usadas como argumentos na chamada de função sejam lidas

```
1       # PARA NAO LER FATORES DAS CHAMADAS DE FUNÇÃO
2       if any(parent in ignore_nodes for parent in node.ancestors):
3           continue
4
5       # Se o nó atual for 'chamada_funcao' ou 'lista_argumentos', adiciona ao
            conjunto
6       if node.name in ["chamada_funcao", "lista_argumentos"]:
7           #print(f"Ignorando o nó e seus filhos: {node.name}")
8           ignore_nodes.add(node)
```

```
9           continue
```

Quando o algoritmo encontra um nó chamado `chamada_funcao` ou `lista_argumentos`, ele entra na lista negra, e o nó e seus descendentes são ignorados para evitar o problema mencionado anteriormente.

Outro desafio enfrentado na implementação foi relacionado à busca de variáveis usadas na expressão. Se a variável é um parâmetro da função, a função de busca não conseguia progredir adequadamente. Isso ocorreu porque uma variável declarada é tratada como um ponteiro, enquanto o parâmetro é considerado uma constante. Para resolver esse problema, adicionamos mais instruções `if` para verificar o tipo retornado pela função `getVarInList` (responsavel por entregar os ponteiros das variaveis declaradas).

Otimização do código de entrada

Infelizmente, as únicas otimizações realizadas no projeto são aquelas que resultam diretamente da biblioteca `llvmlite` em Python. Embora a `llvmlite` não execute otimizações diretamente, a estrutura do código gerado cria um ambiente onde não há código morto devido à presença de um retorno (`return`) no final de cada trecho de código.

Para aplicar otimizações adicionais ao código gerado, é possível usar o compilador LLVM `llc` e a ferramenta `opt`. O processo de otimização é realizado em duas etapas:

```
# Gere código IR usando llvmlite e salve em 'code.ll'
# Compile o código IR para um arquivo objeto
llc -filetype=obj code.ll -o code.o

# Aplique otimizações usando opt
opt -O2 code.ll -o optimized.ll

# Compile o código otimizado para o arquivo objeto
llc -filetype=obj optimized.ll -o optimized.o
```

5.2 Saida

O objetivo desta etapa é gerar um código LLVM ou retornar um erro das etapas anteriores. Para executar esta etapa e compilar um código `TPP`, é necessário utilizar apenas uma linha de comando, desde que as bibliotecas mencionadas na introdução do relatório estejam instaladas:

```
python3 tppgencode.py CAMINHO_DO_CODIGO_TPP
```

Caso deseje obter o código de saída ao invés de mensagens, adicione a flag `-k` no final da linha de comando.

Ao utilizar o comando, o código de entrada será lido e o código LLVM será gerado na pasta `tests` como `meu_modulo.ll`, pronto para ser compilado em um arquivo objeto. A seguir, apresentamos alguns exemplos de entrada e saída desta etapa.

Por exemplo, considere o código de entrada mostrado no Código 31:

Algoritmo 31: Código TPP de entrada

```
1     {Declaração de variáveis}
2  inteiro: a
3
4  inteiro principal()
5      inteiro: b
6
```

```
7       a := 10
8
9       b := a
10
11      retorna(b)
12  fim
```

Este código gera a árvore sintática mostrada na Figura 15:

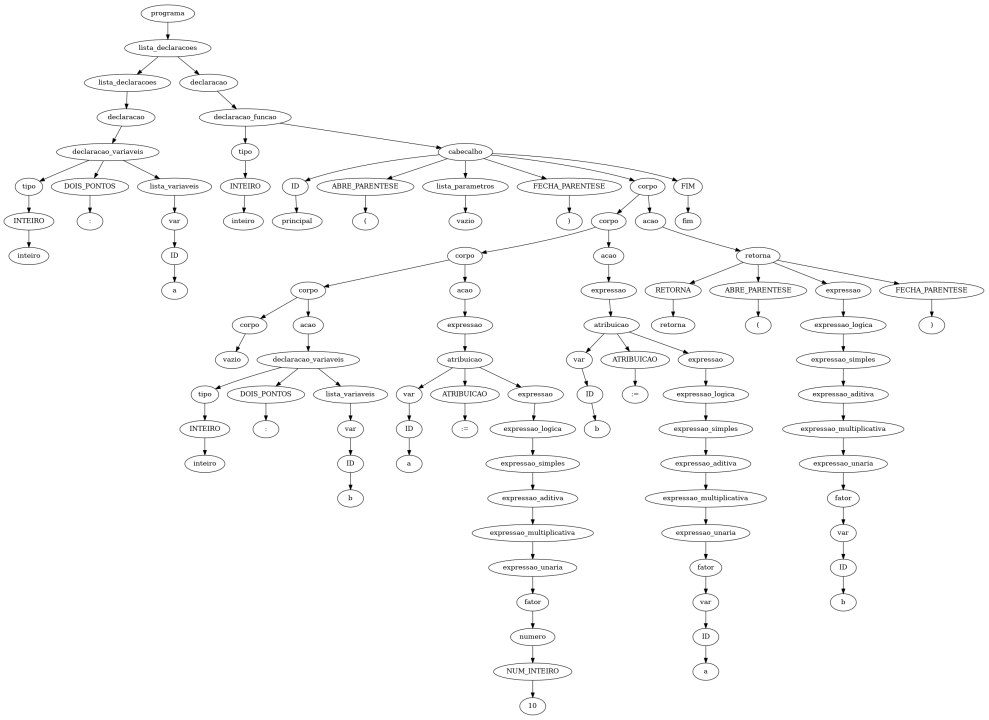

Figura 15: Árvore sintática da entrada

A árvore sintática resulta no seguinte código 32:

Algoritmo 32: meu_modulo.ll (saída)

```
1  ; ModuleID = "meu_modulo.bc"
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
       :16:32:64-S128"
4
5  @"b" = global i32 0, align 4
6  @"a" = global i32 0, align 4
7  define i32 @"main"()
8  {
9  entry:
10   %"c" = alloca i32, align 4
11   store i32 10, i32* @"a"
12   store i32 20, i32* @"b"
13   %"x_temp" = load i32, i32* @"a"
14   %"y_temp" = load i32, i32* @"b"
15   %"soma" = add i32 %"x_temp", %"y_temp"
16   store i32 %"soma", i32* %"c"
17   %"x_temp.1" = load i32, i32* %"c"
18   ret i32 %"x_temp.1"
19 }
```

O código 32 pode ser compilado em um arquivo objeto usando o compilador `llc` e, em seguida, o arquivo objeto pode ser compilado pelo `gcc` (usado para C e C++) ou pelo `clang`. Para compilar o código LLVM para um arquivo objeto e gerar o executável, utilize os seguintes comandos:

```
llc -filetype=obj meu_modulo.ll -o code.o
```

Para compilar o arquivo objeto e gerar o executável, utilize:

```
clang code.o -o meu_executavel
```

ou

```
gcc code.o -o meu_executavel
```

Finalmente, execute o programa gerado com:

```
./meu_executavel
```

6 Considerações Finais

Ao longo do projeto, compreendemos a importância dos compiladores e seu comportamento geral, com o objetivo de aprofundar nosso entendimento sobre programação e os algoritmos subjacentes nas linguagens de programação modernas. Esse conhecimento proporciona uma ”vantagem”no aprendizado de novas linguagens e aprimora a compreensão dos conceitos fundamentais da ciência da computação.

Para a criação do compilador, optamos pelo uso de *Python*, devido à sua facilidade de uso e à ampla disponibilidade de ferramentas poderosas. Utilizamos ferramentas léxicas para análise léxica, *yacc* para análise sintática e *llvmlite* para geração de código. Essas ferramentas oferecem suporte robusto e eficiente para as diversas etapas do processo de compilação, facilitando o desenvolvimento e a implementação de nosso compilador.

Além disso, a escolha do *Python* permite uma rápida prototipagem e desenvolvimento iterativo, o que é essencial para explorar e ajustar os aspectos complexos do compilador de forma eficiente. As bibliotecas e ferramentas disponíveis no ecossistema Python tornam o processo de construção de um compilador mais acessível e menos propenso a erros, proporcionando uma base sólida para a implementação e expansão de funcionalidades.

Referências

GONÇALVES, R. A. 2016. Projeto llvm-gencode-samples. https://github.com/rogerioag/llvm-gencode-samples.

GONÇALVES, R.A. 2024. Projeto de implementação de um compilador para a linguagem tpp: Análise léxica (trabalho – 1ª parte).