

Relatório de Compiladores

Luis Otavio Oliveira Capelari

¹Universidade Tecnológica Federal do Paraná (UTFPR)
Bacharelado em Ciência da Computação
Departamento Acadêmico de Computação (DACOM)
Campo Mourão – PR – Brasil

luiscapelari@alunos.utfpr.edu.br

Abstract. *In computing, a compiler is a program responsible for translating a program written in a high-level language into a machine language. To achieve this, the compiler divides this task into several steps, namely: lexical analysis, syntactic analysis and semantic analysis. Based on this information, this article presents the implementation of an algorithm to perform the T++ language compilation.*

Resumo. *Na área de computação, um compilador é um programa responsável por traduzir um programa escrito em uma linguagem de alto nível para uma linguagem de máquina. Para conseguir isso, o compilador divide essa tarefa em várias etapas, sendo elas: análise léxica, análise sintática e análise semântica. Com base nessas informações, esse artigo apresenta a implementação de um algoritmo para realizar a compilação da linguagem T++.*

1. Introdução

Este artigo é um relatório desenvolvido para a disciplina de Compiladores. O objetivo desse artigo é demonstrar como será feita cada etapa da compilação de um algoritmo escrito na linguagem T++.

A seção 2 apresenta a linguagem T++. Em seguida, na seção 3 será explicado o que é a análise léxica e as suas sub-partes. Na seção 4 será explicada a Análise Sintática. Chegando na seção 5 temos o detalhamento da análise semântica. Por fim, a seção 6 conta sobre a transformação do código em T++ para uma versão compilável para linguagem C.

2. Linguagem T++

A linguagem de programação T++ (ou tpp), é uma linguagem simples, possuindo poucas palavras reservadas, alguns operadores e suportando apenas alguns tipos de dados. Os lexemas da linguagem T++ são escritos em português brasileiro.

Os tipos de dados suportados são: números inteiros, números reais (com ponto flutuante) e arranjos uni e bidimensionais. Os operadores suportados são: +, -, *, /, (,), [,], ', ', : =, :, &&, ||, !, <>, <=, >=, <, >, e =. E as palavras reservadas são: se, então, senão, fim, repita, flutuante, inteiro, retorna, até, leia e escreva.

3. Análise Léxica

A análise léxica é uma fase da compilação de um código-fonte, ela consiste no reconhecimento de tokens através de expressões regulares.

3.1. Tokens

Para facilitar o entendimento, nesse artigo trataremos tokens e lexemas como sendo a mesma coisa. Os tokens representam a qual classe uma determinada palavra no código pertence, por exemplo, no trecho de código `i := i + 1` temos os seguintes tokens `<ID>` `<ATRIBUICAO>` `<ID>` `<numero inteiro>`.

Na análise léxica da linguagem T++ temos os seguintes tokens para serem reconhecidos:

- ID
- Ponto flutuante em notação científica
- Ponto flutuante
- Número inteiro
- Sinal de adição
- Sinal de subtração
- Sinal de multiplicação
- Sinal de divisão
- Símbolo de "E"lógico
- Símbolo de "OU"lógico
- Diferença
- Menor ou igual
- Maior ou igual
- Menor
- Maior
- Igual
- Negação
- Abrir parenteses
- Fechar parenteses
- Abrir colchetes
- Fechar colchetes
- Vírgula
- Dois pontos
- Atribuição

Além disso, reconhecer as seguintes palavras reservadas:

- se
- então
- senão
- fim
- repita
- flutuante
- retorna
- até
- leia
- escreva
- inteiro

Tabela 1. Tabela de expressões regulares.

3.3. Código fonte do algoritmo em Python

Para realizar a análise léxica de códigos em linguagem T++, um algoritmo em Python foi desenvolvido. Esse algoritmo utilizou a biblioteca PLY, em específico o módulo lex, que fornece suporte a análise léxica.

Nas figuras a seguir serão apresentados os trechos do algoritmo em Python.

```
# -*- coding: utf-8 -*-
from sys import argv, exit
import ply.lex as lex
from ply.lex import TOKEN

import logging
logging.basicConfig(
    level = logging.DEBUG,
    filename = "log.txt",
    filemode = "w",
    format = "%(filename)10s:%(lineno)4d:%(message)s"
)
log = logging.getLogger()
```

Figura 1. Importação da biblioteca e configuração de registro de log.

```
tokens = [
    "ID", # identificador
    # numerais
    "NUM_NOTACAO_CIENTIFICA", # ponto flutuante em notação científica
    "NUM_PONTO_FLUTUANTE", # ponto flutuante
    "NUM_INTEIRO", # inteiro
    # operadores binarios
    "MAIS", # +
    "MENOS", # -
    "MULTIPLICACAO", # *
    "DIVISAO", # /
    "E_LOGICO", # &&
    "OU_LOGICO", # ||
    "DIFERENCA", # <>
    "MENOR_IGUAL", # <=
    "MAIOR_IGUAL", # >=
    "MENOR", # <
    "MAIOR", # >
    "IGUAL", # =
    # operadores unarios
    "NEGACAO", # !
    # simbolos
    "ABRE_PARENTESE", # (
    "FECHA_PARENTESE", # )
    "ABRE_COLCHETE", # [
    "FECHA_COLCHETE", # ]
    "VIRGULA", # ,
    "DOIS_PONTOS", # :
    "ATRIBUICAO", # :=
    # 'COMENTARIO', # {***}
]
```

Figura 2. Definição dos tokens.

```

reserved_words = {
    "se": "SE",
    "então": "ENTÃO",
    "senão": "SENAO",
    "fim": "FIM",
    "repita": "REPITA",
    "flutuante": "FLUTUANTE",
    "retorna": "RETORNA",
    "até": "ATE",
    "leia": "LEIA",
    "escreva": "ESCREVA",
    "inteiro": "INTEIRO",
}

tokens = tokens + list(reserved_words.values())

```

Figura 3. Definição das palavras reservadas.

```

digito = r"([0-9])"
letra = r"([a-zA-Záâãäåæéíîóôõö])"
sinal = r"([\-\+\?]*)"

id = (
    r"(" + letra + r"(" + digito + r"+|_" + letra + r")*)"
)

inteiro = r"d+"

flutuante = (
    r'^(d+[eE][+-]?|d+(\.d+|\d+\.d*)([eE][+-]?|d+)?)'
)

notacao_cientifica = (
    r"(" + sinal + r"([1-9])\. + digito + r"+[eE]" + sinal + digito + r"+)"
)

```

Figura 4. Expressões regulares para identificadores, e tipos de números.

```

# Expressões Regulares para tokens simples.
# Símbolos.
t MAIS = r'\+'
t MENOS = r'\-'
t MULTIPLICACAO = r'\*'
t DIVISAO = 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':'

# Operadores Lógicos.
t E_LOGICO = r'&&'
t OU_LOGICO = r'\|\|'
t NEGACAO = r'!'

# Operadores Relacionais.
t DIFERENCA = r'<>'
t MENOR_IGUAL = r'<='
t MAIOR_IGUAL = r'>='
t MENOR = r'<'
t MAIOR = r'>'
t IGUAL = r'='

```

Figura 5. Expressões regulares para operadores.

```

@TOKEN(id)
def t_ID(token):
    token.type = reserved_words.get(
        token.value, "ID"
    )

    return token

@TOKEN(notacao_cientifica)
def t_NUM_NOTACAO_CIENTIFICA(token):
    return token

@TOKEN(flutuante)
def t_NUM_PONTO_FLUTUANTE(token):
    return token

@TOKEN(inteiro)
def t_NUM_INTEIRO(token):
    return token

t_ignore = " \t"

```

Figura 6. Reconhecimento de identificadores, e tipos de números.

```

def t_COMENTARIO(token):
    r"(\{((.|\\n)*?)\\})"
    token.lexer.lineno += token.value.count("\\n")

def t_newline(token):
    r"\\n+"
    token.lexer.lineno += len(token.value)

def define_column(input, lexpos):
    begin_line = input.rfind("\\n", 0, lexpos) + 1
    return (lexpos - begin_line) + 1

def t_error(token):

    line = token.lineno

    message = "Caracter inválido '%s'" % token.value[0]

    print(message)

    token.lexer.skip(1)

```

Figura 7. Reconhecimento de comentários, quebra de linha, coluna do carácter e erros.

```

def main():
    # argv[1] = 'Fibonacci.tpp'
    aux = argv[1].split('.')
    if aux[-1] != 'tpp':
        raise IOError("Not a .tpp file!")
    data = open(argv[1])

    source_file = data.read()
    lexer.input(source_file)

    # Tokenize
    while True:
        tok = lexer.token()
        if not tok:
            break # No more input
        # print(tok)
        print(tok.type)
        print(tok.value)

    # Build the lexer.
    # __file__ = "analizador_lexico.py"
    lexer = lex.lex(optimize=True, debug=True, debuglog=log)

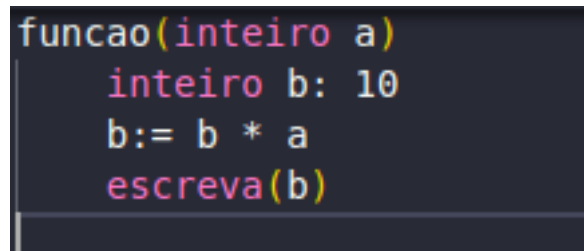
if __name__ == "__main__":
    main()

```

Figura 8. Função principal do algoritmo.

3.4. Execução do algoritmo de análise léxica

Para finalizar esse artigo e validar se o algoritmo desenvolvido está funcionando corretamente, foi executado um pequeno teste com um código em T++, para verificar se a saída do algoritmo em Python reconheceu corretamente os tokens. Na figura 9 é apresentado o código em T++ e a figura 10 mostra o resultado do teste, confirmando que o algoritmo em Python está correto.

A screenshot of a code editor showing T++ code. The code is: funcao(inteiro a) inteiro b: 10 b:= b * a escreva(b). The text is color-coded: 'funcao' is blue, 'inteiro' is red, 'a)' is green, 'b:' is red, '10' is green, 'b:= b * a' is red, and 'escreva(b)' is blue.

```
funcao(inteiro a)
    inteiro b: 10
    b:= b * a
    escreva(b)
```

Figura 9. Código em T++.

A terminal window showing the execution of a Python script. The prompt is 'luiscapelari@luiscapelari-Lenovo-ideapad-310-14ISK:~/Área de Trabalho/compiladores'. The command is '\$ python3 analisador_lexico.py teste_rapido.tpp'. The output lists tokens: ID, ABRE_PARENTESE, INTEIRO, ID, FECHA_PARENTESE, INTEIRO, ID, DOIS_PONTOS, NUM_INTEIRO, ID, ATRIBUICAO, ID, MULTIPLICACAO, ID, ESCRIVA, ABRE_PARENTESE, ID, and FECHA_PARENTESE.

```
luiscapelari@luiscapelari-Lenovo-ideapad-310-14ISK:~/Área de Trabalho/compiladores
$ python3 analisador_lexico.py teste_rapido.tpp
ID
ABRE_PARENTESE
INTEIRO
ID
FECHA_PARENTESE
INTEIRO
ID
DOIS_PONTOS
NUM_INTEIRO
ID
ATRIBUICAO
ID
MULTIPLICACAO
ID
ESCREVA
ABRE_PARENTESE
ID
FECHA_PARENTESE
```

Figura 10. Resultado da execução do algoritmo em Python para o arquivo em T++.

4. Análise Sintática

A Análise Sintática é responsável por determinar a sintaxe de um programa com base nos tokens gerados na Análise Léxica. Sua principal função é definir se a sequência de tokens contidos no programa é uma sequência válida. A sintaxe da linguagem é definida usando regras gramaticais de uma gramática livre de contexto, GLC.

Gramáticas livres de contexto são bem parecidas com Expressões Regulares, porém melhores, já que uma GLC pode ser recursiva. Uma GLC é uma especificação de uma linguagem de programação, ela é formada por:

- Um conjunto de terminais
- Um conjunto de não-terminais
- Um não terminal inicial

- Um conjunto de produções
- Uma linguagem denotada pela gramática

Sendo que uma produção, também chamada de regra, é composta por um par de um não-terminal e uma cadeia de terminais e não-terminais, além de que as regras gramaticais podem ser recursivas, tanto à direita quanto à esquerda

Um dos principais componentes da Análise Sintática é seu algoritmo, chamado de parser, que pode usar uma análise ascendente ou descendente. Ele é responsável por determinar a estrutura do programa e construir uma árvore sintática que represente essa estrutura.

4.1. Gramática da linguagem T++

As regras gramaticais da linguagem T++ foram desenvolvidas seguindo a forma de Backus-Naur, ou BNF, que é uma notação para representar regras gramaticais.

A seguir, da figura 11 até a figura 46, serão apresentadas as regras gramaticais que compõem a linguagem T++, representadas com a ajuda do software Railroad Diagram Generator.

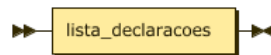


Figura 11. programa \rightarrow lista_declaracoes.

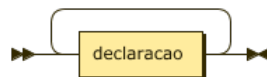


Figura 12. lista_declaracoes \rightarrow lista_declaracoes declaracao | declaracao.

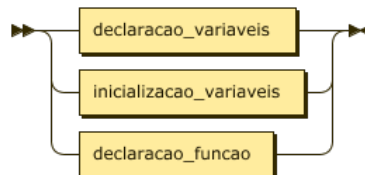


Figura 13. declaracao \rightarrow declaracao_variaveis | inicializacao_variaveis | declaracao_funcao.

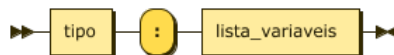


Figura 14. declaracao_variaveis \rightarrow tipo : lista_variaveis.

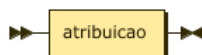


Figura 15. inicializacao_variaveis \rightarrow atribuicao.

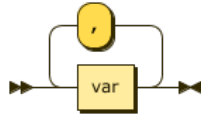


Figura 16. $\text{lista_variaveis} \rightarrow \text{lista_variaveis} , \text{var} \mid \text{var}.$

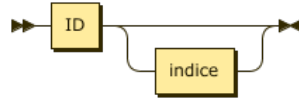


Figura 17. $\text{var} \rightarrow \text{ID} \mid \text{ID indice}.$

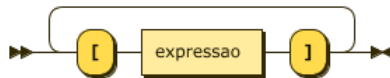


Figura 18. $\text{indice} \rightarrow \text{indice} [\text{expressao}] \mid [\text{expressao}].$

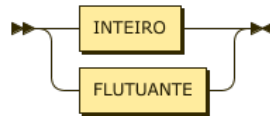


Figura 19. $\text{tipo} \rightarrow \text{INTEIRO} \mid \text{FLUTUANTE}.$

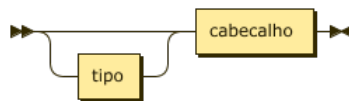


Figura 20. $\text{declaracao_funcao} \rightarrow \text{tipo cabecalho} \mid \text{cabecalho}.$



Figura 21. $\text{cabecalho} \rightarrow \text{ID} (\text{lista_parametros}) \text{ corpo FIM}.$

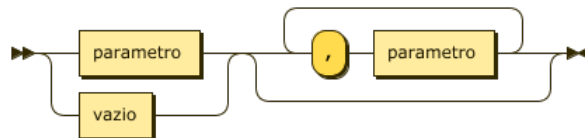


Figura 22. $\text{lista_parametros} \rightarrow \text{lista_parametros} , \text{parametro} \mid \text{parametro} \mid \text{vazio}.$

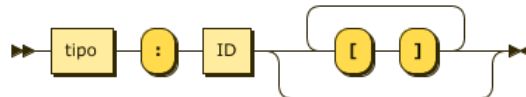


Figura 23. $\text{parametro} \rightarrow \text{tipo} : \text{ID} \mid \text{parametro} [].$

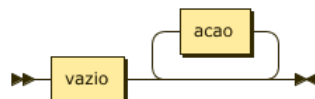


Figura 24. $\text{corpo} \rightarrow \text{corpo} \text{ acao} \mid \text{vazio}.$

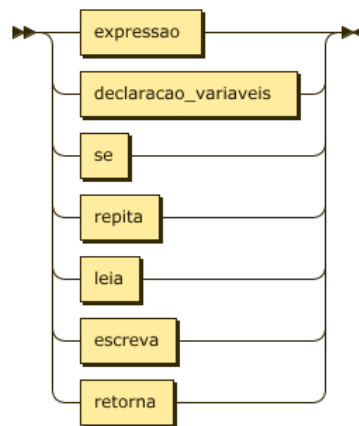


Figura 25. $acao \rightarrow expressao \mid declaracao_variaveis \mid se \mid repita \mid leia \mid escreva \mid retorna$.

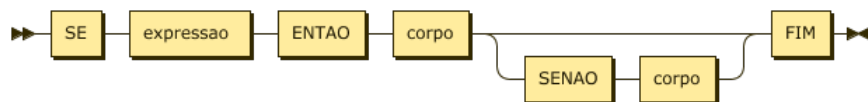


Figura 26. $se \rightarrow SE \text{ expressao } ENTAO \text{ corpo } FIM \mid SE \text{ expressao } ENTAO \text{ corpo } SENAO \text{ corpo } FIM$.



Figura 27. $repita \rightarrow REPITA \text{ corpo } ATE \text{ expressao}$.



Figura 28. $atribuicao \rightarrow var := expressao$.



Figura 29. $leia \rightarrow LEIA (var)$.



Figura 30. $escreva \rightarrow ESCREVA (expressao)$.



Figura 31. $retorna \rightarrow RETORNA (expressao)$.

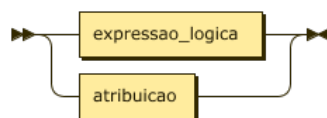


Figura 32. $expressao \rightarrow expressao_logica \mid atribuicao$.

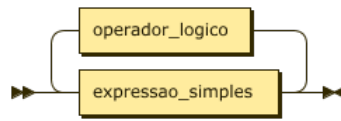


Figura 33. $\text{expressao_logica} \rightarrow \text{expressao_simples}$
 $\text{expressao_logica operador_logico expressao_simples}.$

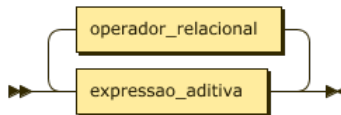


Figura 34. $\text{expressao_simples} \rightarrow \text{expressao_aditiva}$
 $\text{expressao_simples operador_relacional expressao_aditiva}.$

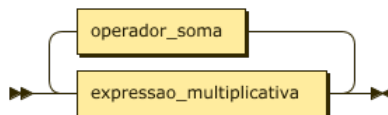


Figura 35. $\text{expressao_aditiva} \rightarrow \text{expressao_multiplicativa}$
 $\text{expressao_aditiva operador_soma expressao_multiplicativa}.$

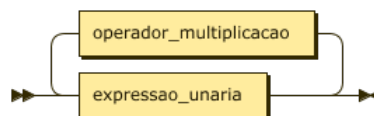


Figura 36. $\text{expressao_multiplicativa} \rightarrow \text{expressao_unaria}$
 $\text{expressao_multiplicativa operador_multiplicacao expressao_unaria}.$

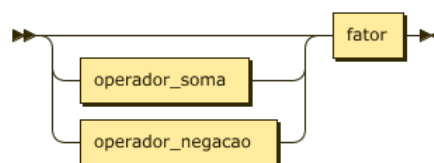


Figura 37. $\text{expressao_unaria} \rightarrow \text{fator}$ | $\text{operador_soma fator}$
 $\text{operador_negacao fator}.$

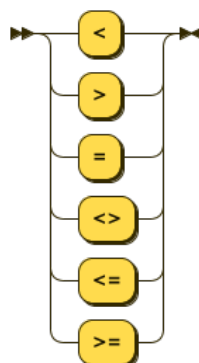


Figura 38. $\text{operador_relacional} \rightarrow < | > | = | <> | <= | >=.$

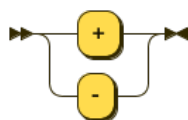


Figura 39. operador_soma $\rightarrow + \mid -$.

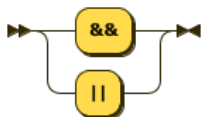


Figura 40. operador_logico $\rightarrow \&\& \mid ||$.



Figura 41. operador_negacao $\rightarrow !$.

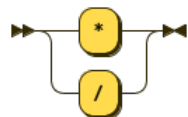


Figura 42. operador_multiplicacao $\rightarrow * \mid /$.

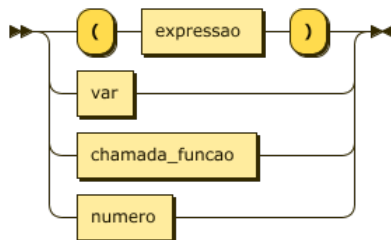


Figura 43. fator $\rightarrow (\text{expressao}) \mid \text{var} \mid \text{chamada_funcao} \mid \text{numero}$.

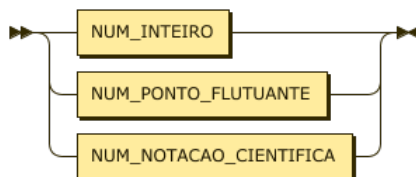


Figura 44. numero $\rightarrow \text{NUM_INTEIRO} \mid \text{NUM_PONTO_FLUTUANTE} \mid \text{NUM_NOTACAO_CIENTIFICA}$.



Figura 45. chamada_funcao $\rightarrow \text{ID} (\text{lista_argumentos})$.

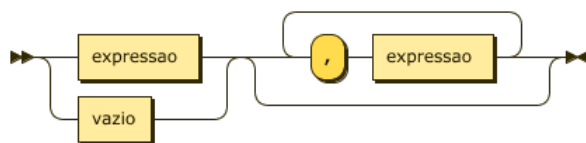


Figura 46. lista_argumentos $\rightarrow \text{lista_argumentos} , \text{expressao} \mid \text{expressao} \mid \text{vazio}$.

4.2. Análise Sintática LALR

Para fazer a análise sintática da linguagem T++ foi utilizado o método de análise ascendente LALR. A análise ascendente constrói a árvore a partir das folhas indo até a raiz (símbolo não-terminal inicial). As palavras são classificadas da esquerda para a direita, procurando uma derivação mais a direita. Por mais que o parser comece do símbolo final indo em direção ao símbolo alvo, as derivações começam do símbolo alvo indo ao símbolo final. Esse método usa as funções shift (inserir token na pilha) e reduce (trocar um símbolo por um não-terminal), que usam um automato de pilha, para fazer a análise. O analisador LALR usa uma tabela de estados menores do que outros analisadores, como LR(0) e LR(1), o que acelera o processo de análise.

4.3. Código Fonte

Para realizar a análise sintática, foi desenvolvido um algoritmo em Python, utilizando o modulo Yacc da biblioteca PLY, e um código gerador da árvore sintática, explicado na sub-seção 4.4.

A figura 47 demonstra a regra na forma BNF para a análise sintática da função ESCRVA. A frase escrita entre aspas é a regra gramatical, usada pelo Yacc para reconhecer a função ESCRVA. As marcas contidas nessa regra são detectadas em cada posição de p , como, por exemplo, a chamada da função ocupa a posição $p[1]$. Cada marca é representada em um nó na árvore sintática.

```
def p_escrava(p):
    """escrava : ESCRVA ABRE_PARENTESE expressao FECHA_PARENTESE"""

    pai = MyNode(name='escrava', type='ESCREVA')
    p[0] = pai

    filho1 = MyNode(name='ESCREVA', type='ESCREVA', parent=pai)
    filho_sym1 = MyNode(name=p[1], type='ESCREVA', parent=filho1)
    p[1] = filho1

    filho2 = MyNode(name='ABRE_PARENTESE', type='ABRE_PARENTESE', parent=pai)
    filho_sym2 = MyNode(name='(', type='SIMBOLO', parent=filho2)
    p[2] = filho2

    p[3].parent = pai # expressao.

    filho4 = MyNode(name='FECHA_PARENTESE', type='FECHA_PARENTESE', parent=pai)
    filho_sym4 = MyNode(name=')', type='SIMBOLO', parent=filho4)
    p[4] = filho4
```

Figura 47. Função para a regra da função ESCRVA.

Na figura 48 é demonstrada a função que trata dos erros gerados pela sintaxe da função ESCRVA.

A figura 49 apresenta a função contendo a regra gramatical para listas de argumentos. Esse exemplo faz uso de recursão (uma lista de argumentos pode produzir outra lista de argumento) e possui varias produções (separadas pelo carácter |).

A figura 50 mostra a função para tratar erros em geral.

```
def p_escrava_error(p):
    """escrava : ESCREVA ABRE_PARENTESE error FECHA_PARENTESE"""
    print("Erro na definicao da acao ESCREVA.")

    print("Erro:p[0]:{p0}, p[1]:{p1}, p[2]:{p2}, p[3]:{p3}, p[4]:{p4}".format(
        p0=p[0], p1=p[1], p2=p[2], p3=p[3], p4=p[4]))
    error_line = p.lineno(3)
    father = MyNode(name='ERROR:{}'.format(error_line), type='ERROR')
    logging.error(
        "Syntax error parsing index rule at line {}".format(error_line))
    parser.errok()
    p[0] = father
```

Figura 48. Função para a regra que captura erros na função ESCREVA.

```
def p_lista_argumentos(p):
    """lista_argumentos : lista_argumentos VIRGULA expressao
    | expressao
    | vazio
    """
    pai = MyNode(name='lista_argumentos', type='LISTA_ARGUMENTOS')
    p[0] = pai

    p[1].parent = pai
    if len(p) > 2:
        filho2 = MyNode(name='VIRGULA', type='VIRGULA', parent=pai)
        filho_sym = MyNode(name=p[2], type='SIMBOLO', parent=filho2)
        p[2] = filho2

    p[3].parent = pai
```

Figura 49. Função para a regra de lista de argumentos.

```
def p_error(p):
    if p:
        token = p
        print("Erro:[{line},{column}]: Erro próximo ao token '{token}'".format(
            line=token.lineno, column=token.lineno, token=token.value))
```

Figura 50. Função que trata erros em geral.

A figura 51 mostra a função principal do programa, que a partir de um código em T++, faz sua análise sintática, gera sua árvore sintática e a salva em uma imagem.

Na figura 52 vemos a inicialização do Yacc, usando método LALR, definindo como simbolo inicial "programa", habilitando debug e definindo nome do arquivo da tabela de parsing .

```

def main():
    aux = argv[1].split('.')
    if aux[-1] != 'tpp':
        raise IOError("Not a .tpp file!")
    data = open(argv[1])

    source_file = data.read()
    parser.parse(source_file)

    if root and root.children != ():
        print("Generating Syntax Tree Graph...")
        DotExporter(root).to_picture(argv[1] + ".ast.png")
        UniqueDotExporter(root).to_picture(argv[1] + ".unique.ast.png")
        DotExporter(root).to_dotfile(argv[1] + ".ast.dot")
        UniqueDotExporter(root).to_dotfile(argv[1] + ".unique.ast.dot")
        print(RenderTree(root, style=AsciiStyle()).by_attr())
        print("Graph was generated.\nOutput file: " + argv[1] + ".ast.png")

        DotExporter(root, graph="graph",
                     nodelnamefunc=MyNode.nodelnamefunc,
                     nodeattrfunc=lambda node: 'label=%s' % (node.type),
                     edgeattrfunc=MyNode.edgeattrfunc,
                     edgetypefunc=MyNode.edgetypefunc).to_picture(argv[1] + ".ast2.png")
    else:
        print("Unable to generate Syntax Tree.")
        print('\n\n')

```

Figura 51. Função principal.

```

parser = yacc.yacc(method="LALR", optimize=True, start='programa', debug=True,
                  debuglog=log, write_tables=False, tabmodule='tpp_parser_tab')

```

Figura 52. Inicialização do Yacc.

4.4. Árvore Sintática

Para montar a estrutura da Árvore Sintática, foi desenvolvido um algoritmo a parte, apresentado nas figuras 53 e 54.

Na figura 55 vemos um exemplo de uma árvore gerada, onde os nós folhas representam o código analisado. Com a figura 56 vemos uma árvore que detectou um erro na linha 1 do código apresentado na figura 57, causado pela inicialização de um vetor sem especificar seu tamanho. E na figura 58 é possível ver que o código de análise sintática também apresenta os erros detectados nas linhas do terminal.


```

from anytree import Node, RenderTree, AsciiStyle, PreOrderIter
from anytree.exporter import DotExporter
from anytree import NodeMixin, RenderTree

node_sequence = 0

class MyNode(NodeMixin): # Add Node feature

    def __init__(self, name, parent=None, id=None, type=None, label=None, children=None):
        super(MyNode, self).__init__()
        global node_sequence

        if (id):
            self.id = id
        else:
            self.id = str(node_sequence) + ': ' + str(name)

        self.label = name
        self.name = name
        node_sequence = node_sequence + 1
        self.type = type
        self.parent = parent
        if children:
            self.children = children

    def nodenamefunc(node):
        return '%s' % (node.name)

    def nodeattrfunc(node):
        return '%s' % (node.name)

```

Figura 53. Primeira parte do código gerador da árvore sintática.

```

def edgeattrfunc(node, child):
    return ''

def edgetypefunc(node, child):
    return '--'

```

Figura 54. Segunda parte do código gerador da árvore sintática.

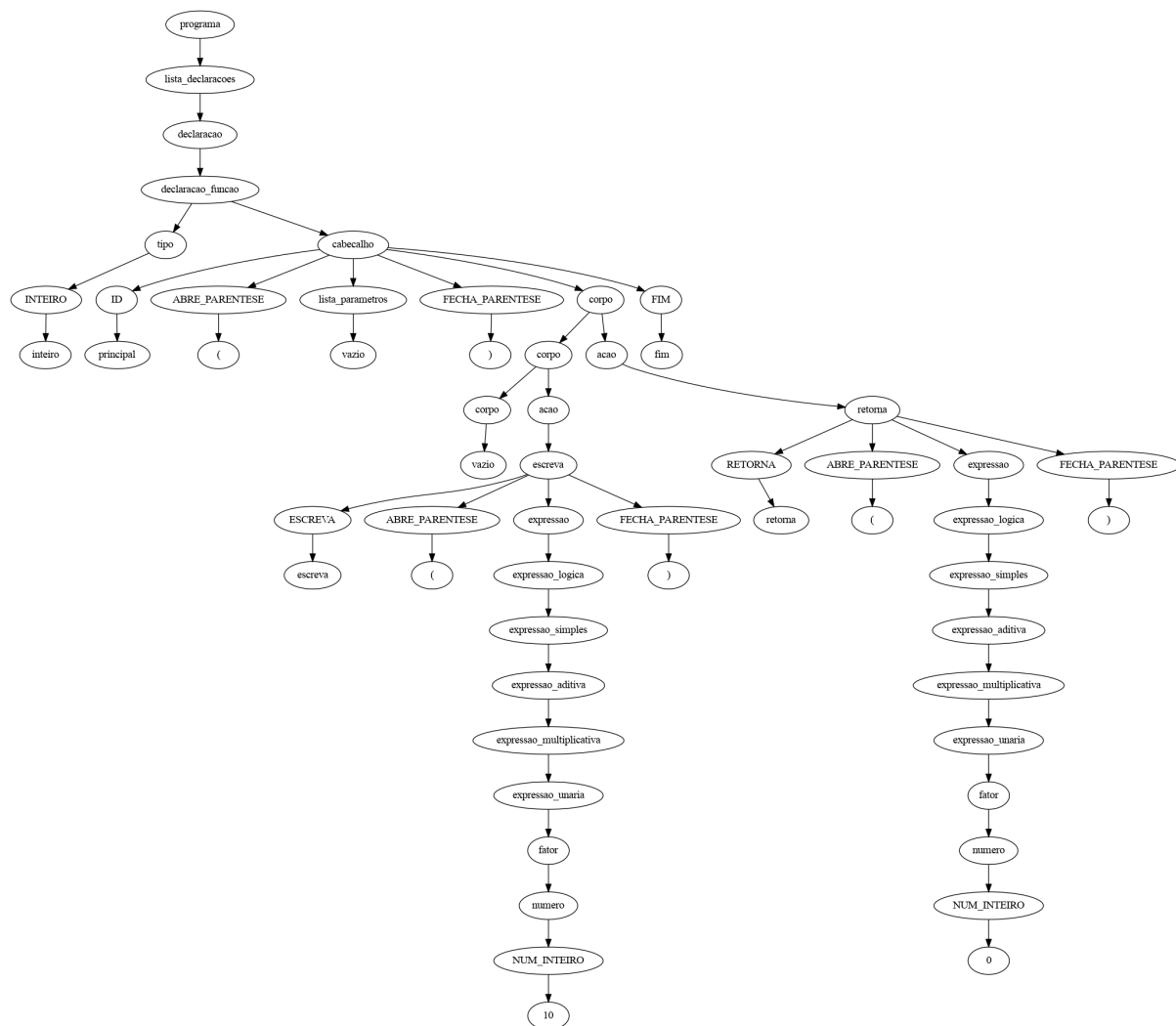


Figura 55. Exemplo de árvore sintática.

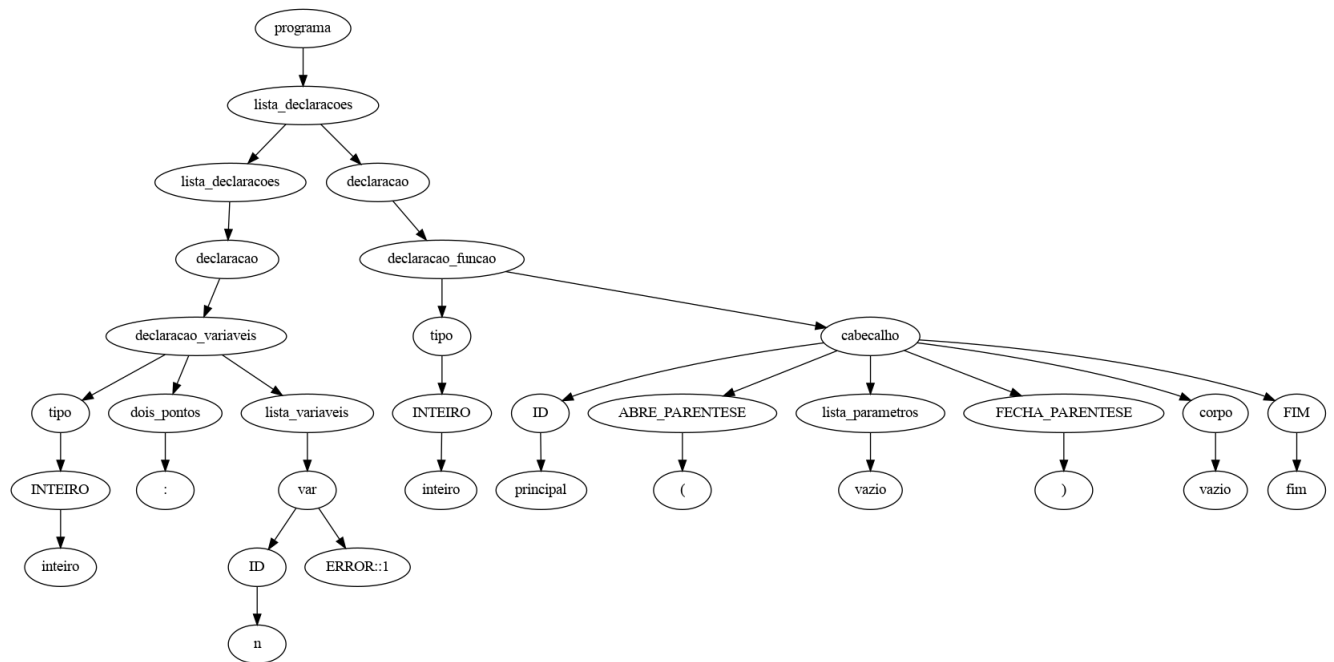


Figura 56. Nó com erro.

```
erro-004.tpp x
inteiro: n[
inteiro principal()
fim
```

Figura 57. Código de exemplo com erro.

```
python3 tppparser.py ../sintatica-testes/erro-006.tpp
Generating LALR tables
WARNING: 30 shift/reduce conflicts
Erro:[5,5]: Erro próximo ao token 'se'
Erro na definicao do parametro.
Erro:p[0]:None, p[1]:LexToken(ID,'x',6,45), p[2]:x
Erro:[7,7]: Erro próximo ao token 'senão'
Erro:[9,9]: Erro próximo ao token 'fim'
Erro:[12,12]: Erro próximo ao token 'se'
```

Figura 58. Erro apresentado no terminal.

5. Análise Semântica

Dando sequência na compilação de códigos na linguagem T++, passamos para a etapa de Análise Semântica, também chamada de análise sensível ao contexto. Nessa etapa são encontrados erros de tipo e de concordância, como, por exemplo: se uma variável utilizada foi inicializada; se uma função é declarada, mas não é utilizada; se os tipos presentes em uma expressão são compatíveis; etc. Essa análise possui uma capacidade maior de análise do que as regras definidas por GLCs.

Os objetivos da análise semântica são: garantir a corretude de um programa e melhorar a eficiência da execução de um programa. Para alcançar isso, a análise semântica faz a análise dos atributos/propriedades de uma linguagem. Um compilador tem interesse nos atributos denominados estáticos, determinados antes da execução do programa.

5.1. Estratégia para a análise sensível ao contexto

Na Análise Semântica o código desenvolvido em Python usou listas como variáveis para armazenar informações a respeito dos nós, dos casos encontrados e das regras semânticas, como é possível ver na figura 59.

```
nodeNewRoot = None
tempCabecalhoFunc = None
ParentTree = list()
list_parents = list()
list_terminal= list()
list_semantic= list()
n_used_func = list()
list_func_checked= list()
list_func_declared= set()
list_vars_declared= set()
node_list_parents = list()
node_list_terminal= list()
list_func_no_return = set()
list_var_inicializada = list()
ignorar = ["vazio", ",", "]
criarPai = ["var", "numero"]
```

Figura 59. Listas de controle e armazenamento de informações

A execução da Análise Semântica é invocada pela função *main*, visível na figura 60. A função começa recebendo a árvore gerada pela Análise Sintática, percorre a árvore separando nós terminais de não-terminais, executa as funções para tratar cada caso possível de erro, então gera a tabela de símbolos e faz a redução da árvore.

Foi feita uma função para percorrer a árvore sintática e armazenar informações sobre ela e, também foram feitas duas funções, que juntas tem a responsabilidade de armazenar informações a respeito do nó que as invocaram, e informações sobre seus descendentes. A figura 61 apresenta essas funções.

Para realizar de fato a análise, a função apresentada na figura 62 faz chamadas para as funções "grandes", ou seja, as funções mais importantes. Dentro de cada uma dessas funções são chamadas funções secundárias. Após executar as funções que analisam as regras e verificam os casos que podem gerar erros ou alertas, a função apresentada na figura 62 executa a montagem da tabela de símbolos.

```
def main():
    arvore = None
    try:
        arvore = tpparser.main()
        percorre(arvore)
        executar_tudo()

    except:
        print("ERROR: não foi possível capturar as regras semânticas")

    tree_builder(arvore)

if __name__ == "__main__":
    main()
```

Figura 60. Função main

```
def percorre(arvore):
    if len(arvore.children)>0: #se tiver filhos
        list_parents.append(arvore)
        for i in arvore.children:
            percorre(i) #percorrendo a arvores para os filhos de um nó
    else:
        list_terminal.append(arvore) #se for um simbolo terminal

def percorre_node(node):
    node_list_parents.clear() # para armazenarem apenas as informacoes de um nó passado, sem acumular de todos nós da arvore
    node_list_terminal.clear()
    walk_node(node)

def walk_node(node):
    if len(node.children)>0:
        node_list_parents.append(node)
        for i in node.children:
            walk_node(i)
    else:
        node_list_terminal.append(node)
```

Figura 61. Funções para percorrer arvore sintática

```
def executar_tudo():
    declaracao_variavel()
    main_rule()
    funcao_declarada()
    tem_atribuicao()
    funcoes_param_compare()
    check_escreva()
    print_tabela()
```

Figura 62. Função que executa as principais funções

Em seguida temos alguns exemplos de funções das análises de semântica, como, por exemplo, função para verificar se a função *principal* foi declarada, verificar se o índice de um array é válido, verificar se uma variável foi inicializada e uma função para verificar se uma variável não foi inicializada, apresentadas nas figuras 63, 64, 65 e 66 respectivamente.

```
def main_rule():
    for i in list_terminal:
        if i.name == "principal": #achar nó que referencia a funcao main
            if i.parent.parent.name != "chamada funcao":
                tipo = i.parent.parent.parent.children[0].children[0].children[0] #adicionando o tipo da funcao
                node = i # nó da principal
                escopo = i.name #escopo
                list_principal = [tipo,node,-1,"global",escopo,"funcao","global"]
                list_semantic.append(list_principal.copy()) #adicionando a funcao a lista das valores semanticas
                funcoes_rule(i.parent.parent,i.name)
                return
    print("ERROR: Função principal não declarada")
```

Figura 63. Função para verificar se a função *principal* foi declarada

```
def indice_valido(node):
    for i in list_semantic:
        if node.children[0].children[0].name == i[1].name: #acha o nó referente ao nome da variavel
            percorre_node(node)
            for j in node_list_parents:
                if j.name == "numero": #acha onde tem o numero do indice
                    if float(j.children[0].children[0].name) > float(i[2].name): #se o numero
                                                                #atual for maior do que o na declaracao
                        print('ERROR: Índice \033[1m{}\033[0m do array \033[1m{}\033[0m está fora do intervalo'.format(
                            j.children[0].children[0].name,i[1].name))
            return
```

Figura 64. Função para verificar se um índice é válido

```
def tem_atribuicao():
    for i in list_parents:
        if i.name == "atribuicao": #acha atribuicao de variavel
            percorre_node(i)
            cabecalho_funcao(i)

            temp = list()
            hasFunc = False
            for find in node_list_parents:
                if find.name == "chamada_funcao":
                    hasFunc = True
                if find.name == "indice":
                    hasFunc = True
                    indice_valido(find.parent)

            for j in node_list_parents:
                if j.name == "ID":
                    if tempCabecalhoFunc != None: #se a variavel estiver dentro de uma funcao
                        temp.append([j.children[0],tempCabecalhoFunc.children[0].children[0].name])
                    else:
                        temp.append([j.children[0],"global"])
                list_var_inicializada.append(temp[0]) #adiciona como uma variavel que foi inicializada
            elif j.name == "numero" and not hasFunc:
                if tempCabecalhoFunc != None:
                    temp.append([j.children[0],tempCabecalhoFunc.children[0].children[0].name])
                else:
                    temp.append([j.children[0],"global"])
            check_tipo_var(temp)

        if i.name == 'leia':
            cabecalho_funcao(i)
            for j in i.children:
                if j.name == 'var':
                    temp = list()
                    if tempCabecalhoFunc != None: #se a variavel estiver dentro de uma funcao
                        temp.append([j.children[0].children[0],tempCabecalhoFunc.children[0].children[0].name])
                    else:
                        temp.append([j.children[0].children[0],"global"])
                    list_var_inicializada.append(temp[0]) #adiciona como uma variavel que foi inicializada
                    check_tipo_var(temp)
```

Figura 65. Função para conferir a atribuição em uma variável

```

def variavel_not_inicializada(variable, scope):
    for i in list_var_inicializada:
        if i[0].name == variable.name:
            if i[1] == scope or i[1] == "global": #se a variavel foi inicializada em algum escopo
                return 0

    for i in list_semantic:
        if i[1].name == variable.name:
            if i[5] == "funcao":
                return 0

    print('WARN: Variável \033[1m{}\033[0m declarada mas não inicializada'.format(variable.name))
    return -1

```

Figura 66. Função para conferir se a variável foi inicializada

5.2. Estratégia para montagem da tabela de símbolos

Após todos os casos terem sido tratados é gerada a tabela de símbolos do programa analisado. Uma tabela de símbolos contém informações a respeito das variáveis e funções do programa, informações essas sobre os contextos de cada coisa, como nome, tipo, escopo, etc. Para realizar isso são apresentadas informações contidas em uma lista que é atualizada em boa parte das funções de análise. A função que monta a tabela de símbolos é apresentada na figura 67.

```
def print_tabela():
    print("\n===== TABELA =====\n")
    print('{:^6}| {:^10}| {:^8}| {:^20}| {:^20}| {:^9}'.format('ESCOPO', 'DECLARACAO', 'TAG', 'LOCAL',
                                                             'NOME', 'TIPO'))
    print("-----")

    for i in list_semantic:
        if i[0] == "vazio":
            print("{:^6}| {:^10}| {:^8}| {:^20}| {:^20}| {:^9}".format(i[3], i[6], i[5], i[4],
                                                                      i[1].name, i[0]))
        else:
            print("{:^6}| {:^10}| {:^8}| {:^20}| {:^20}| {:^9}".format(i[3], i[6], i[5], i[4],
                                                                      i[1].name, i[0].name))
    print("\n===== \n")
```

Figura 67. Função para montar a tabela de símbolos

5.3. Redução da Árvore Semântica

Com a análise realizada e a tabela gerada partimos para a redução da árvore, onde os nós intermediários (não-terminais) são "podados" da árvore. A função da figura 68 chama a função presente na figura 69. A função *arvore_reduce* percorre a partir da raiz da árvore, verificando quais nós podem ser podados. Ao podar ela faz a ligação do nó considerado "mais importante" com o próximo nó "importante" ou com um nó folha, deixando de lado os outros nós que existiam no caminho. Essa redução ajudará na etapa de geração de código intermediário.

```
def tree_builder(root):
    try:
        arvore_reduce(root, MainRoot())
    except:
        print("ERROR: não foi possível gerar a Árvore Semântica")

    try:
        UniqueDotExporter(nodeNewRoot).to_picture(argv[1] + ".resume.unique.ast.png")
    except:
        0
```

Figura 68. Função para gerar arvore reduzida


```

def arvore_reduce(node,parent):
    if len(node.children) > 1:
        count = 0
        values = node_new_parent(node.name)
        valor, mudar_parent = values[0], values[1]
        if valor == -1:
            valor = math.ceil(len(node.children)/2) - 1 #valor do meio

        if mudar_parent: #se der para reduzir
            new_node = MyNode(name=node.name,parent=parent)
            parent = new_node
            Tree_add_parent(new_node)
        else:
            final_children(node.children[valor]) #pega folha do filho do meio; casos de atribuicao e declaracao
            ParentTree[0].parent = parent #adiciona o parent como pai do nó folha
            parent = ParentTree[0] #nó folha é o novo parent

        for i in node.children:
            if not mudar_parent:
                if count != valor:
                    arvore_reduce(i,parent) #rodando para reduzir; vai mudando o filho mas o pai continua
            else:
                arvore_reduce(i,parent)
            count += 1

    elif len(node.children) == 1:
        arvore_reduce(node.children[0], parent)

    else: #terminal
        if parent != None and node != None:
            if node.name not in ignorar:
                if node.parent.parent.name in criarPai:
                    if node.parent.parent.name == "numero":
                        node.parent = MyNode(name=node.name, parent=parent)
                    else:
                        node.parent = MyNode(name=node.parent.name,parent=parent)
                else:
                    node.parent = parent

```

Figura 69. Função para realizar podas na árvore

5.4. Demonstração

Para testar o código de análise semântica, foi executada a análise para o programa apresentado no trecho de código 1. A saída da execução do código em Python de análise semântica está visível na figura 70. A árvore reduzida, também chamada de árvore abstrata, desse programa é representada na figura 71.

```

1  flutuante: a
2  inteiro: b
3
4  func()
5      a := 10.2
6      retorna(a)
7  fim
8
9  inteiro principal()
10      b := 5
11      func()
12  fim

```

Listing 1. Código exemplo

```
python3 semantica.py semantica-testes/sema-018.tpp
ERROR: Função principal do tipo inteiro não possui retorno
ERROR: Função func do tipo vazio retornando tipo flutuante
```

TABELA							
ESCOPO	DECLARACAO	TAG	LOCAL	NOME	TIPO		
global	expressao	variável	global	a	flutuante		
global	expressao	variável	global	b	inteiro		
global	global	funcao	principal	principal	inteiro		
global	global	funcao	func	func	vazio		

Figura 70. Execução do algoritmo de análise sintática do programa sema-018.tpp

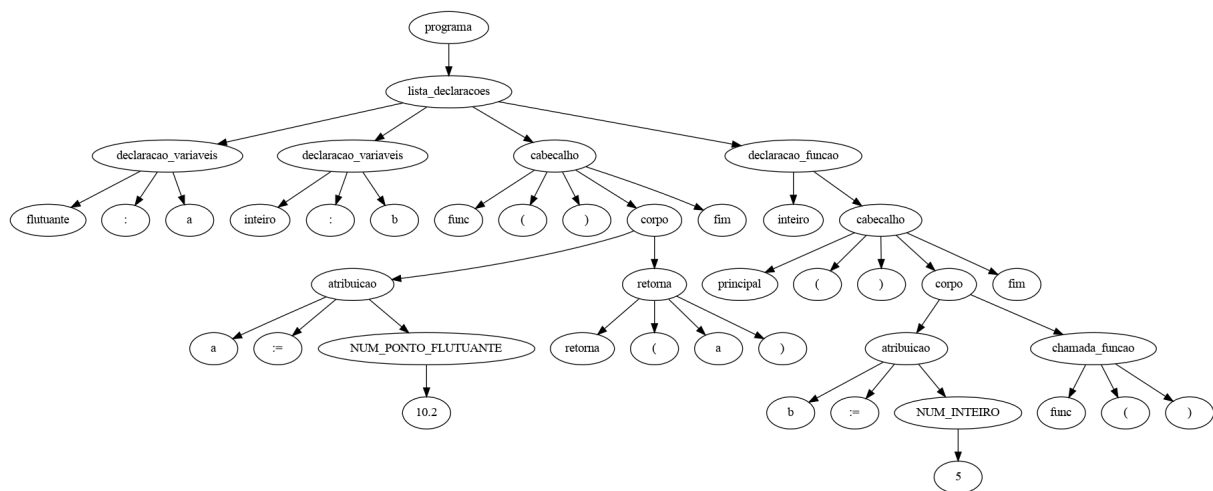


Figura 71. Árvore semântica reduzida do programa sema-018.tpp

6. Geração de código

Após feitas todas as análises no código em T++ é feita a conversão desse código para uma versão que pode ser compilada em C. O código gerado é voltado para uma máquina em específico. Em geral, essa etapa pode envolver otimização de código, mas nesse trabalho isso não foi realizado. Para realizar a geração desse código foi utilizado um código intermediário, criado utilizando a biblioteca LLVM Lite do Python.

6.1. LLVM Lite

O LLVM Lite é uma biblioteca em Python do projeto LLVM que permite gerar códigos intermediários que podem ser compilados para linguagem C utilizando a ferramenta clang. Um código intermediário é uma representação do código original que se assemelha do código-alvo. Ele é gerado com base na árvore sintática, que nesse trabalho é a árvore podada na fase de análise semântica.

6.2. Algoritmo para geração de código

Para fazer a geração do código intermediário foi criado um algoritmo em Python. Ele recebe a entrada das outras etapas para percorrer a árvore sintática, detectar as instruções e converte-las para o código intermediário.

Começando pelas configurações do arquivo, temos na figura 72 as importações de módulos da biblioteca LLVM Lite, importação do código de semântica que executará as outras etapas anteriores e retornará sua árvore podada, e importação do código de estrutura da árvore. Na figura 73 temos a inicialização do LLVM Lite e a criação de um módulo que irá conter o código-fonte que será gerado.

```
from llvmlite import ir
from llvmlite import binding as llvm
from llvmlite.binding import value
from llvmlite.ir.types import IntType
import semantica
from mytree import MyNode
```

Figura 72. Importação do LLVM Lite

```
# Código de Inicialização.
llvm.initialize()
llvm.initialize_all_targets()
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()

# Cria o módulo.
module = ir.Module('modulo_L0.bc')
module.triple = llvm.get_process_triple()
target = llvm.Target.from_triple(module.triple)
target_machine = target.create_target_machine()
module.data_layout = target_machine.target_data
```

Figura 73. Inicialização do LLVM Lite

Para começar a geração do código intermediário temos na figura 74 a função *main*, que recebe a árvore da etapa de análise semântica e começa a tratar os casos para converter as informações da árvore em instruções compiláveis. Após passar por todos casos, o código intermediário é escrito em um arquivo em um formato legível para humanos.

A parte de tratamento dos casos começa na função *montar*, visível na figura 75, onde quatro casos maiores são tratados, declaração e atribuição de variáveis globais, e declarações de funções.

Nas figuras 76 e 77 vemos que a declaração e a atribuição em variáveis globais tratam três casos: variáveis de tipo primitivo, e arrays uni e bi-dimensionais.

Percorrendo a árvore sintática, a cada declaração de função encontrada é chamada uma função específica para tratar esse caso, como apresentado na figura 78. A função *buildFuncao* tem a responsabilidade de criar a estrutura do bloco de entrada e saída de um função declarada no código em T++. Nessa função são usados métodos do LLVM Lite para especificar a função, dizendo seu nome, tipo de retorno e delimitando seu escopo (começo e fim).

Dentro de cada função há um corpo de código, contendo variadas outras instruções. A partir da função *buildFuncao* a função que trata cada caso específico é invocada, como

```

if __name__ == '__main__':
    # global module
    arvore = None
    try:
        arvore = semantica.main()
        percorre(arvore)
        montar(arvore.children[0]) #lista_declaracoes

    except:
        print("ERROR: não foi possível gerar o código")

arquivo = open('meu_modulo.ll', 'w')
arquivo.write(str(module))
arquivo.close()
# print(module)

```

Figura 74. Função main

```

def montar(node):
    for i in node.children:
        if i.name == 'declaracao funcao':
            gerarFuncao(i)
        elif i.name == 'cabecalho':
            gerarFuncaoVazia(i)
        elif i.name == 'declaracao variaveis':
            func_declaracao_varGlobal(i)
        elif i.name == 'atribuicao':
            func_atribuicaoGlobal(i) # se existir variavel global

```

Figura 75. Função montar

```

def func_declaracao_varGlobal(node):
    typeVar = ir.IntType(32) if node.children[0].name == 'inteiro' else ir.FloatType()
    if node.children[2].name == 'lista variaveis':
        for i in node.children[2].children:
            nameVar = i.name
            if nameVar == 'var':
                mountGlobalArray(i)
            else:
                criarVarGlobal(node, typeVar, nameVar)
    elif node.children[2].name == 'var': #variavel array
        mountGlobalArray(node.children[2])
    else: #variavel normal
        nameVar = node.children[2].name
        criarVarGlobal(node, typeVar, nameVar)

```

Figura 76. Declaração de variáveis globais

mostram as figuras 79 e 80.

Para exemplificar, na figura 81 temos a função de trata do caso de uma função leia do T++ ser executada. Nela é preciso tratar caso a variável que irá receber um valor seja uma variável primitiva ou um array, e caso seja um array é preciso obter o ponteiro para a posição que recebera o valor lido. Então é chamada uma função que realiza a leitura de uma variável de acordo com seu tipo (inteiro ou flutuante), como visto na figura 83.

```

def func_atribuicaoGlobal(node):
    if node.children[0].name == 'var':
        if node.children[0].children[1].name == 'indice':
            if len(node.children[0].children[1].children) > 3:
                atribuicaoArrayGlobal2D(node)
            else:
                atribuicaoArrayGlobal(node)
        else:
            atribuicaoNumGlobal(node)

```

Figura 77. Atribuição em variáveis globais

```

def buildFuncao(node,nome,tipo_retorno):
    global current_scope
    current_scope = nome
    nome = 'main' if nome == 'principal' else nome
    list_var_local[current_scope] = {'':None}
    list_func_params = list()

    for j in node.children:
        if j.name == 'lista_parametros':
            for k in j.children:
                param_tipo = ir.IntType(32) if k.children[0].name == 'inteiro' else ir.FloatType()
                param_name = k.children[2].name
                list_func_params.append(param_tipo)

        if j.name == 'corpo':
            tipo_func = ir.FunctionType(tipo_retorno,list_func_params.copy())
            func = ir.Function(module,tipo_func,nome)
            entryBlock = func.append_basic_block('entry')
            exitBasicBlock = func.append_basic_block('exit')
            builder = ir.IRBuilder(entryBlock)
            list_func_declarada.update({nome:[func,list_func_params,tipo_retorno]})
            for k in j.children:
                map_func_especial.get(k.name)(k,builder)
            builder.branch(exitBasicBlock)
            builder.position_at_end(exitBasicBlock)
            # builder.ret(builder.load(r))

def gerarFuncaoVazia(node):
    nome = node.children[0].name
    tipo_retorno = ir.VoidType()
    buildFuncao(node,nome,tipo_retorno)

def gerarFuncao(node):
    global module
    tipo_retorno = ir.IntType(32) if node.children[0].name == 'inteiro' else ir.FloatType()

    for i in node.children:
        if i.name == 'cabecalho':
            nome = i.children[0].name
            buildFuncao(i,nome,tipo_retorno)

```

Figura 78. Função buildFuncao para criar escopo de uma função

Depois o valor lido é salvo (*store*) na variável.

```
map_func_especial = {
    'se': func_se,
    'leia': func_leia,
    'escreva': func_escreva,
    'repita': func_repita,
    'retorna': func_retorna,
    'chamada_funcao': func_call_f,
    'atribuicao': func_atribuicao,
    'declaracao_variaveis': func_declaracao_var
}
```

Figura 79. Mapeamento para funções de casos específicos

```
def func_se(node,bloco): ...

#TODO
def func_leia(node,bloco): ...

def func_escreva(node,bloco): ...

def func_repita(node,bloco): ...

def func_retorna(node,bloco): ...

def func_call_f(node,bloco): ...

def func_calcular(operacao,retorno,exp1,exp2,bloco): ...

#TODO
def resolveExpressao(node,bloco): ...

#TODO
def func_atribuicao(node,bloco): ...
```

Figura 80. Definições dos TADs das funções específicas

```

def func_leia(node,bloco):
    global module
    global current_scope
    array = False

    if node.children[2].name == 'var':
        if node.children[2].children[1].name == 'indice':
            nome = node.children[2].children[0].name
            pos = int(node.children[2].children[1].children[1].children[0].name)
            array = True
        else:
            nome = node.children[2].name
    var = list_var_local[current_scope].get(nome)
    if var == None:
        var = list_vars_global.get(nome)
    funcLeia = leiaInteiro if var[0] == ir.IntType(32) else leiaFlutuante

    varRef = var[1]
    if array:
        varRef = bloco.gep(varRef,[ir.IntType(32)(0),ir.IntType(32)(pos)])

    resultado_leia = bloco.call(funcLeia, args=[])
    bloco.store(resultado_leia, varRef, align=4)

```

Figura 81. Função para conversão da função leia

```

leiaInteiro = ir.Function(module,ir.FunctionType(ir.IntType(32),[]),name="leiaInteiro")
leiaFlutuante = ir.Function(module,ir.FunctionType(ir.FloatType(),[]),name="leiaFlutuante")

```

Figura 82. Funções para ler cada tipo de variável

6.3. Exemplo de execução

Por último, será demonstrado um exemplo de geração de código. Será utilizado o código apresentado em 2.

```
1  inteiro principal()
2      inteiro: x
3      flutuante: y
4
5      x := 0
6      y := 0.0
7
8      leia(x)
9      leia(y)
10     escreva(x)
11     escreva(y)
12
13     retorna(0)
14 fim
```

Listing 2. Código exemplo

A figura mostra o código intermediário gerado, que pode ser compilado e executado usando o clang.

```
; ModuleID = "modulo_L0.bc"
target triple = "x86_64-unknown-linux-gnu"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

declare void @"escrevaInteiro"(i32 %.1)

declare void @"escrevaFlutuante"(float %.1)

declare i32 @"leiaInteiro"()

declare float @"leiaFlutuante"()

define i32 @main()
{
entry:
  %"x" = alloca i32, align 4
  %"y" = alloca float, align 4
  store i32 0, i32* %"x"
  store float 0x0, float* %"y"
  %".4" = call i32 @leiaInteiro()
  store i32 %".4", i32* %"x"
  %".6" = call float @leiaFlutuante()
  store float %".6", float* %"y"
  %".8" = load i32, i32* %"x"
  call void @escrevaInteiro(i32 %".8")
  %".10" = load float, float* %"y"
  call void @escrevaFlutuante(float %".10")
  br label %"exit"
exit:
  ret i32 0
}
```

Figura 83. Código intermediário resultante