UNIVERSIDADE TUIUTI DO PARANÁ

GABRIEL PINTO RIBEIRO DA FONSECA

INTERPRETADOR DA LINGUAGEM D+ INTEGRADO A UMA INTERFACE

CURITIBA

2019

GABRIEL PINTO RIBEIRO DA FONSECA

INTERPRETADOR DA LINGUAGEM D+ INTEGRADO A UMA INTERFACE

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Faculdade de Ciências Exatas e de Tecnologia da Universidade Tuiuti do Paraná, como requisito à obtenção ao grau de Bacharel.

Orientador: Prof. Diógenes Cogo Furlan

CURITIBA

2019

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Dentro do curso de bacharelado em ciências da computação. A disciplina de compiladores é responsável por apresentar ao aluno a forma e o significado contidos na construção de um compilador. Um compilador, de forma convencional, apresenta fases de código fonte, e fases de síntese, oras quais o código fonte de alto nível e transformado em código máquina. A disciplina de compiladores apresenta uma considerável dificuldade no seu aprendizado, uma vez que é uma disciplina muito abrangente e muito profunda. Ele faz uma síntese de todo curso de bacharelado em ciências da computação, uma vez que exige conhecimentos em algoritmos, programação estruturado e orientado a objetos, estrutura de dados, métodos de autômatos, gramáticas e ainda conhecimento de assemble.

O presente projeto tem como objetivo o desenvolvimento de um interpretador da linguagem D+, para auxiliar os alunos na absorção do conteúdo, ajudando-os a fixar melhor os conceitos. O interpretador compila linha por linha e mostra visualmente em um terminal o programa já compilado. Neste software vai aplicar conteúdos visto em sala de aula na prática, proporcionando ao aluno entender como eles são elaborados dentro dos processos de analise léxica e sintática.

Para desenvolver este software, é utilizado um compilador D+, linguagem elaborada pelo mestre Diógenes Cogo Furlan. Este compilador terá modificações para se tornar um interpretador. Para o desenvolvimento da interfase, é utilizado o editor QT Creator com a linguagem C++, pelo fato de este editor junto desta linguagem, disponibilizam ferramentas que facilitariam sua criação.

Com este projeto, os alunos da disciplina de compiladores, terão ao seu alcance uma ferramenta (software) que o auxiliaram no aprendizado da matéria. Podendo criar códigos, e ver a transformação que passa pelo compilador, passando pela análise léxica, análise sintática e análise semântica. Os alunos que utilizarem este software, terão exemplos de árvores ascendentes e árvores descendentes no código escrito na interface em tempo real. Com toda esta possibilidade, o aprendizado dos alunos da disciplina compiladores terão um aprendizado melhor.

2 TEORIA DA COMPILAÇÃO

Este capítulo tem como objetivo apresentar os conceitos envolvendo a compilação, técnicas encontradas nela, razão pelo qual foi criado, seu funcionamento e sua estrutura, como análise léxica, análise sintática e analise semântica, também explicando o funcionamento de um compilador e um interpretador, dando ênfase em suas diferenças.

Um processo de compilação designa o conjunto de tarefas que o compilador deve realizar para poder gerar uma descrição em uma linguagem a partir de outra (SANTOS, 2018). No começo da compilação, o compilador deve garantir que a tradução efetuada do código inserido seja correta, para que a execução dos comandos seja feita sem nenhum erro.

O motivo que influenciou o desenvolvimento dos compiladores se deve ao fato de se obter uma maior eficiência de programação. No princípio os computadores possuíam arquiteturas distintas entre eles, tornando assim a mesma instrução diferente para todas. Este mesmo problema era encontrado na representação de dados (RICARTE, 2008).

Na figura 1 é ilustrado uma situação em que se utiliza uma instrução de soma, nesta imagem pode se observar a complexidade que a comunicação atingia ao transmitir para outra máquina a instrução. Devido a este empecilho, era necessária uma intervenção humana, ocasionando em contratar um especialista para resolver este problema, para cada plataforma. Uma situação pouco eficiente e de alto custo para a empresas (RICARTE, 2008).

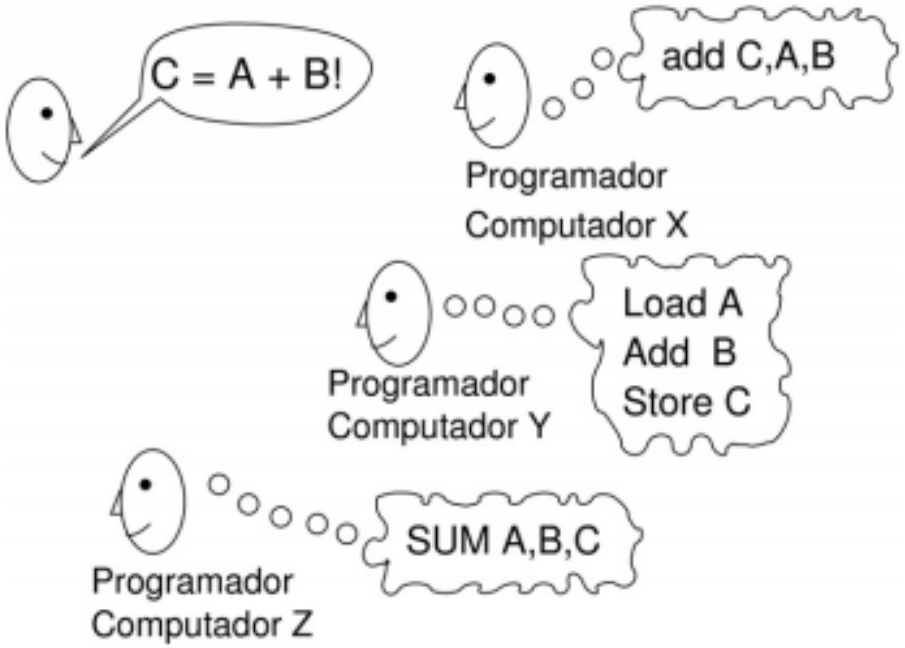
Para solucionar este problema, em busca de uma solução automática, procurando traduzir as especificações genéricas (RICARTE, 2008). Com este objetivo desenvolveu os compiladores.

A figura 2, ilustra o processo já incluindo os compiladores. Com essa automação na tradução do código, se obteve economia pelo fato de não ser necessário a contratação de um especialista para esta tarefa, e aumento de produtividade por ter retirado a intervenção humana em uma parte do processo.

2.1 COMPILADORES

Para se conceituar um compilador de forma simples, o compilador é um programa responsável por traduzir outro programa de uma linguagem fonte para outra linguagem alvo (AHO RAVI SETHI, 1995). Na figura 3 se demostrando a forma do funcionamento dos compiladores como descrito acima.

FIGURA 1 – A programação antes dos compiladores.



FONTE: (RICARTE, 2008).

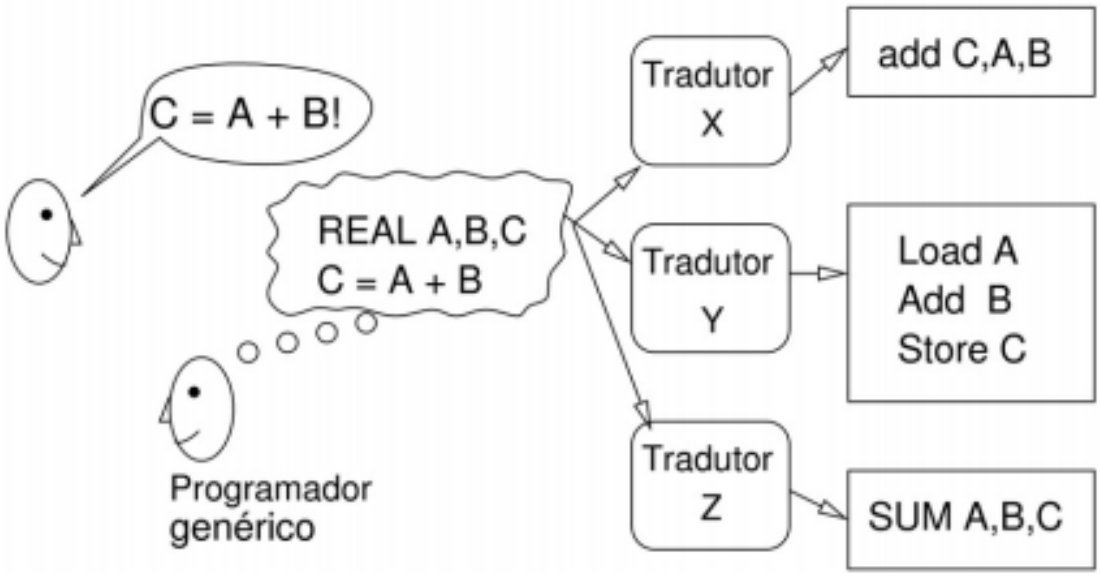
O compilador segue passos para a tradução correta do código fonte para o código de saída. Os passos de análises são: análise léxica (AL), análise sintática (AS) e análise semântica (ASE). Os passos de tradução, são a Geração de código intermediário, a otimização de código, a geração de código final e a otimização de código final. Na figura 4 é ilustrado os passos seguidos dentro de um compilador, quando um código entra em seu processo, e é finalizado como o código de máquina alvo.

2.2 INTERPRETADORES

O interpretador é programa que converte a linguagem atual para uma especifica, mas ao contrário do compilador, ele não converte o código todo para linguagem de máquina de uma vez. Ele executa diretamente cada instrução, passo a passo. MATLAB, LISP, Perl e PHP são apontadas como interpretadas (VICTORIA, 2019). O interpretador efetua a tradução em operações especificadas, dependendo de como foi construído, podendo percorrer o código e ao decorrer desta análise, efetuar as operações necessárias.

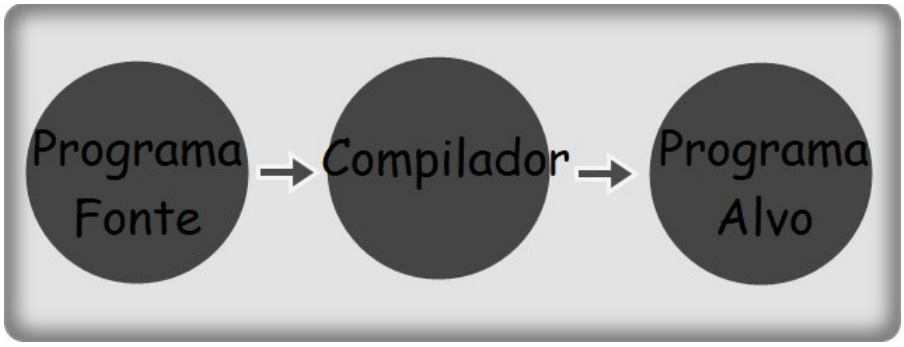
Na figura 4 é ilustrado um fluxo geral de como é convertido o código, comparando o compilador ao interpretador. O que difere entre ambos é em qual momento ocorre a inserção de dados, tempo de execução e a diferença na saída de arquivos após os processos, o compilador gera um arquivo com a linguagem alvo (neste caso linguagem

FIGURA 2 – A programação com compiladores.



FONTE: (RICARTE, 2008).

FIGURA 3 – Exemplo do Funcionamento de um Compilador.



FONTE: O próprio autor.

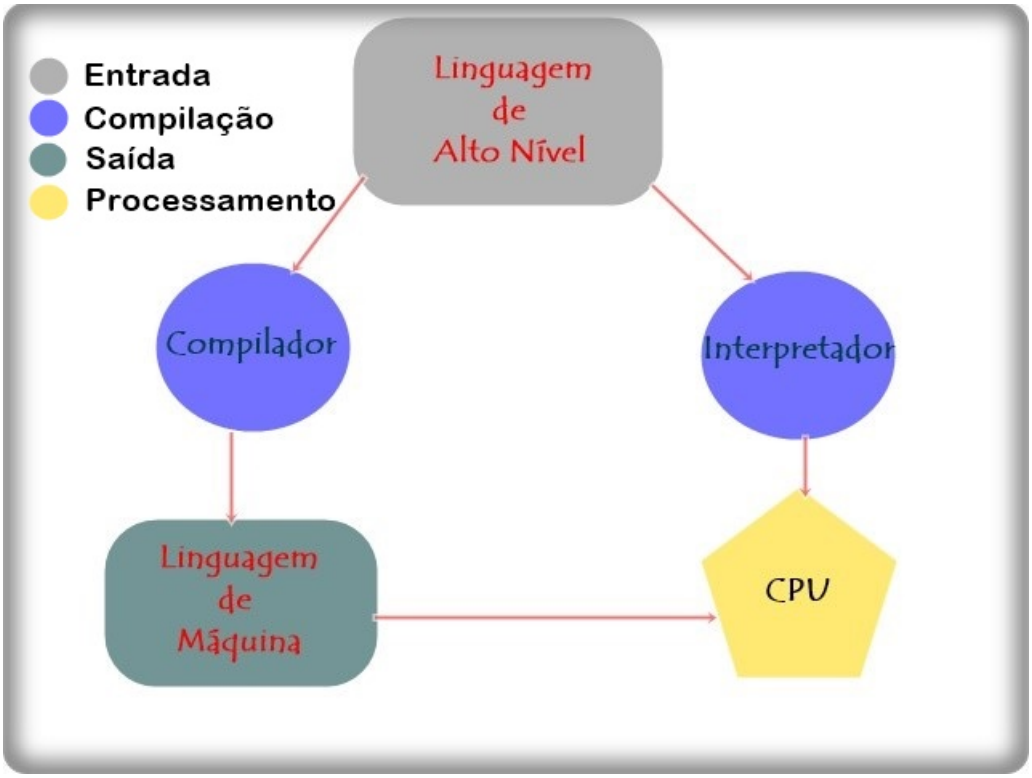
de máquina), e o interpretador executa diretamente a instrução, sem criar um arquivo fonte.

Caso o código seja executado uma segunda vez, ocorrerá uma nova tradução, pois a tradução ocorrida anteriormente não fica armazenada para futuras execuções. Compiladores e interpretadores utilizam nas análises léxica, análise sintática e análise semântica.

2.3 ANÁLISE LÉXICA

A análise Léxica, também chamada de análise scanning, é responsável por analisar os caracteres no programa da esquerda para a direita, e agrupa-los para a formação de tokens. Tokens são sequencias de caracteres que possuem um significado

FIGURA 4 – Comparação Compilador x Interpretador.



FONTE: O próprio autor..

coletivo (AHO RAVI SETHI, 1995).

Um exemplo que pode ser dado, é analisando um programa na linguagem C como ilustrado na figura 5.

A figura 5 expõe os passos internos de um compilador, e o que é gerado em cada passo, como os tokens na análise léxica, árvore sintática na análise sintática e a tabela de símbolos.

Efetuando uma análise léxica no código na linguagem C++ presente na figura 5, é possível apontar os tokens criados e os erros encontrados neste passo.

Começamos descrevendo as linhas corretas e os tokens retirados dela.

a) Identificador X

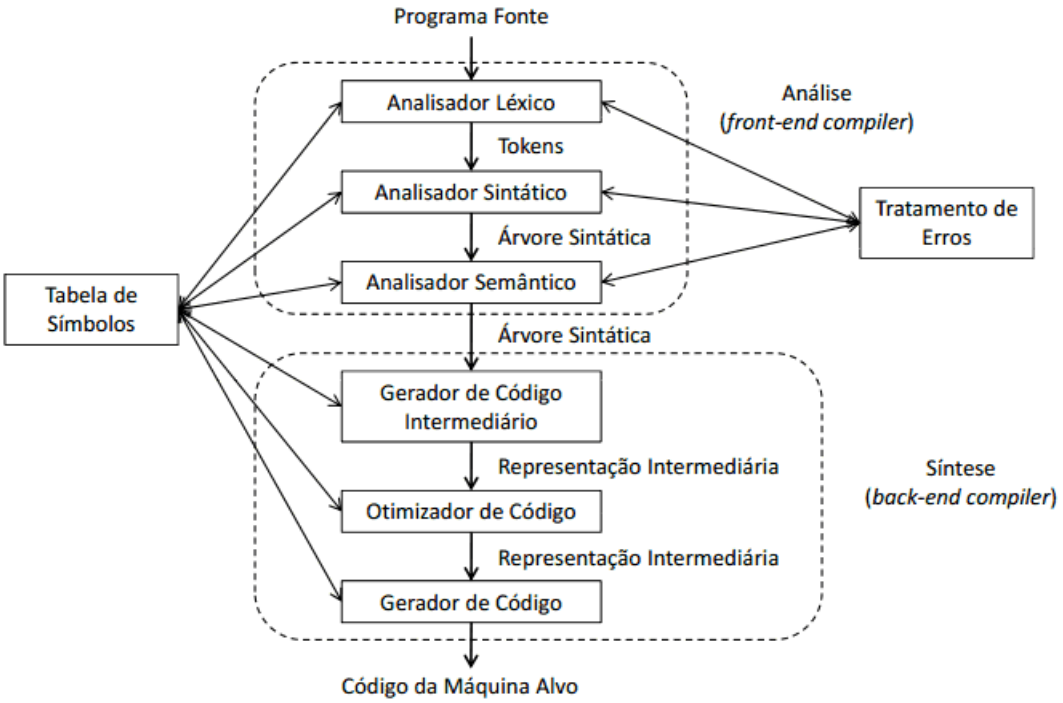
b) Atribuição =

c) Número 6

d) Identificador Y

Estes tokens citados acima, são os gerados na análise léxica, atribuídos e guardados para a próxima análise. Porém o código da figura 5 contém erros, impedindo de avançar para o próximo etapa.

FIGURA 5 – Exemplo dos passos presentes em um compilador.



FONTE: (MARANGON, 2015).

Os erros encontrados na análise léxica são:

a) Linha 8, o analisador léxico não conhece o caractere () como algo valido.

b) Linha 8, o analisador léxico não conhece o caractere) como algo valido.

c) Linha 10, a variável é declarada do tipo string (texto), mas na atribuição não é fechada as (“) para delimitar o fim do texto.

d) Linha 11, é iniciado um comando de bloco de comentário (/*), mas não é fechado com (*/).

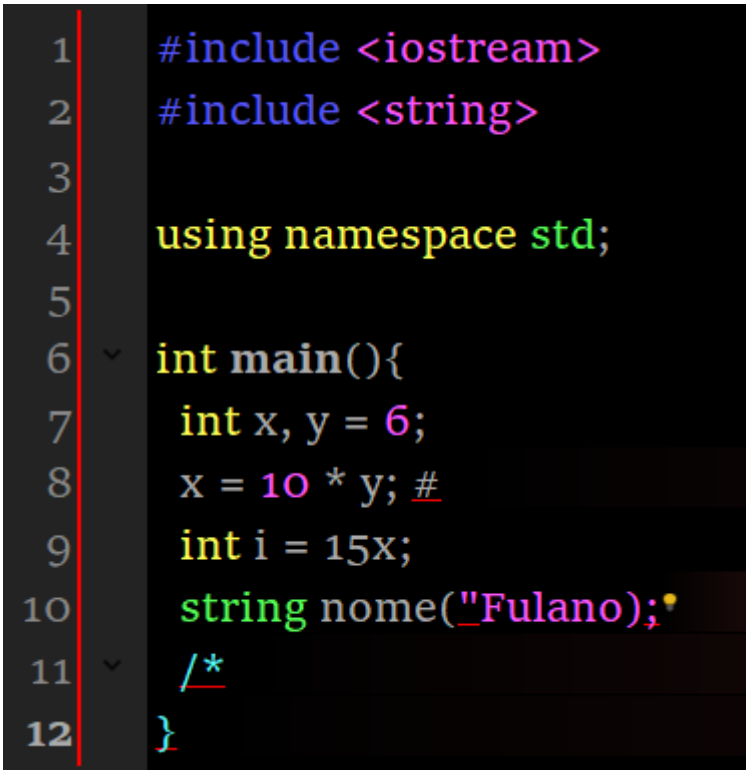
Na figura 5 ocorrem erros de análise sintática, que é o passo que abordaremos em seguida.

2.4 ANÁLISE SÍNTATICA

A análise Sintática é chamada também de análise hierárquica ou análise gramatical. Este passo utiliza dos tokens criados pela análise léxica, para criar um significado coletivo, obtendo uma ordem sequencial (AHO RAVI SETHI, 1995).

O analisador sintático, é responsável por avaliar se os tokens obtidos pelo passo anterior são válidos para a linguagem de programação em que ele e empregado,

FIGURA 6 – Código de exemplo para Análise Léxica.



FONTE: O próprio autor.

validando expressões, funções e métodos. A análise sintática geralmente utiliza de gramatica livre de contexto para especificar a sintaxe de uma linguagem de programação (MARANGON, 2015).

Na figura 7, é apresentado um código na linguagem C++ com alguns erros encontrados na análise sintática, estes erros estão descritos abaixo da figura.

a) Na linha 7, função main não fecha o “(“.

b) Na linha 8, não foi acrescentado o ; no final da linha, com isso o analisador não consegue distinguir o final da instrução e o começo de outra.

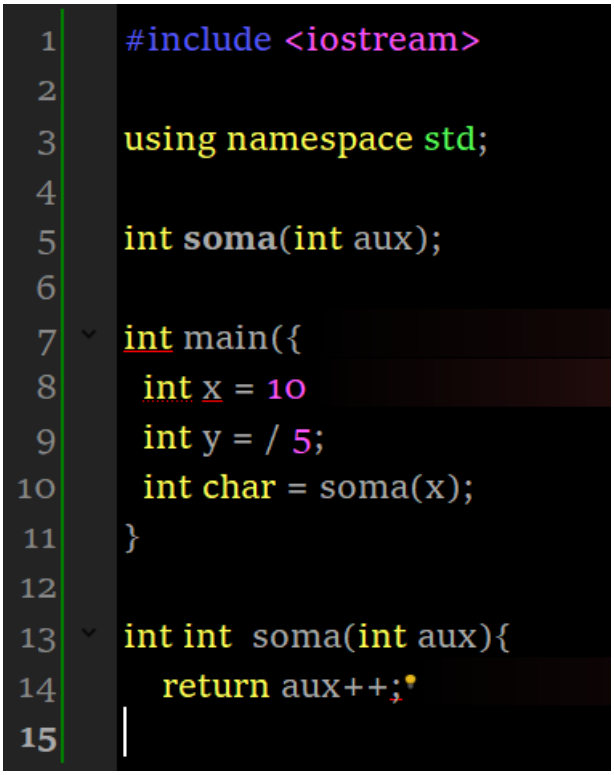
c) Na linha 9, o operador de divisão /, não consegue montar uma expressão de divisão por faltar o operador da esquerda.

d) Na linha 13, o tipo da função int int, confunde o analisador por espera o nome do método após declara o seu tipo inteiro(int).

e) Na linha 15, a função soma, não fecha o .

Após passar pela analise sintática, e verificado se o código faz sentido para linguagem que foi escrita, este passo é trato a seguir.

FIGURA 7 – Código de exemplo para Análise Sintática.



FONTE: O próprio autor.

2.5 ANÁLISE SEMÂNTICA

Até o momento vimos as etapas de análise léxica, que quebra o programa fonte em tokens e a analise sintática, que valida as regras a sintaxe da linguagem de programação. Não é possível representar com expressões regulares ou com uma gramática livre de contexto regras como: todo identificador deve ser declarado antes de ser usado. Muitas verificações devem ser realizadas com meta-informações e com elementos que estão presentes em vários pontos do código fonte, distantes uns dos outros. O analisador semântico utiliza a árvore sintática e a tabela de símbolos para fazer as analise semântica.

A análise semântica é responsável por verificar aspectos relacionados ao significado das instruções, essa é a terceira etapa do processo de compilação e nesse momento ocorre a validação de uma série de regras que não podem ser verificadas nas etapas anteriores (MARANGON, 2015).

Na figura 8, é ilustrado um código na linguagem C++ com alguns erros detectados na análise semântica.

Lista dos erros:

a) Linha 8, variável j sendo atribuída igual ao valor da variável y, porem a

variável j não foi declarada.

b) Linha 5 e 11, duas funções main declaradas.

c) Linha 19, uma classe não pode herdar dela mesma.

FIGURA 8 – Código de exemplo para Análise Semântica.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6    int y = 10;
7    cout << "Primeiro main" << endl;
8    j = y;
9  }
10
11 int main(){
12   cout << "Segundo main" << endl;
13 }
14
15 class veiculo {
16   bool rodas = true;
17 };
18
19 class carro : public  carro{
20 };
```

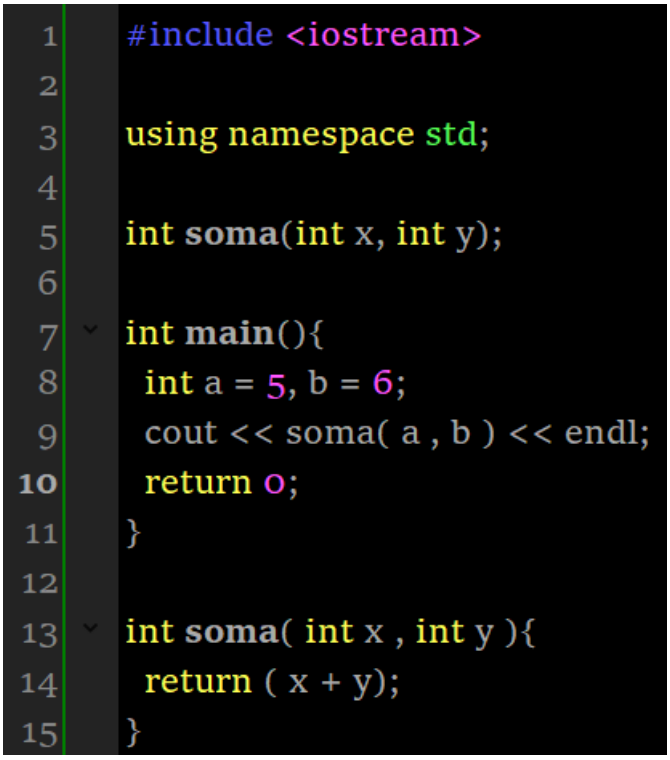
FONTE: O próprio autor.

2.6 TABELA DE SÍMBOLOS

A tabela de símbolos é uma estrutura auxiliar que tem como função apoiar a analise semânticas nas atividades do código (RICARTE, 2008). Este recurso é responsável por armazenar informações de identificadores, como variáveis, tipos de dados, funções e constantes. A estrutura da tabela de símbolos pode ser de uma árvore ou tabela hash.

A seguir temos um código como exemplo para a explicação da tabela de símbolos.

FIGURA 9 – Código de exemplo para Tabela de Símbolos.



```
1  #include <iostream>
2
3  using namespace std;
4
5  int soma(int x, int y);
6
7  int main(){
8   int a = 5, b = 6;
9   cout << soma( a , b ) << endl;
10  return 0;
11 }
12
13 int soma( int x , int y ){
14  return ( x + y);
15 }
```

FONTE: O próprio autor.

2.7 GERAÇÃO DE CODIGO

2.8 LINGUAGEM D+

A linguagem a ser utilizada neste trabalho é a D++, criada pelo professor Diógenes Furlan, para que os alunos que integram a disciplina de compiladores, por ele ministrada, utilizem como base para a criação de um compilador. Algumas regras presentes nesta linguagem são apresentadas na figura 10.

FIGURA 10 – Regras de gramáticas D+.

Declarações

1. programa → lista-decl
2. lista-decl → decl lista-decl | decl
3. decl → decl-var | decl-main
4. decl-var → VAR espec-tipo var ;
5. decl-main → MAIN ( ) bloco END
6. espec-tipo → INT | REAL | CHAR

Comandos

7. bloco → lista-com
8. lista-com → comando lista-com | ε
9. comando → decl-var | com-atrib | com-selecao | com-repeticao | com-leitura | com-escrita
10. com-atrib → var = exp ;
11. com-leitura → SCAN (var) ; | SCANLN (var) ;
12. com-escrita → PRINT (exp) ; | PRINTLN (exp) ;
13. com-selecao → IF exp THEN bloco END-IF | IF exp THEN bloco ELSE bloco END-IF
14. com-repeticao → WHILE exp DO bloco LOOP

Expressões

15. exp → exp-mult op-soma exp | exp-mult
16. op-soma → + | -
17. exp-mult → exp-simples op-mult exp-mult | exp-simples
18. op-mult → * | / | DIV | MOD
19. exp-simples → (exp) | var | literal
20. literal → NUMINT | NUMREAL | CARACTERE | STRING
21. var → ID

FONTE: (FURLAN, 2018).

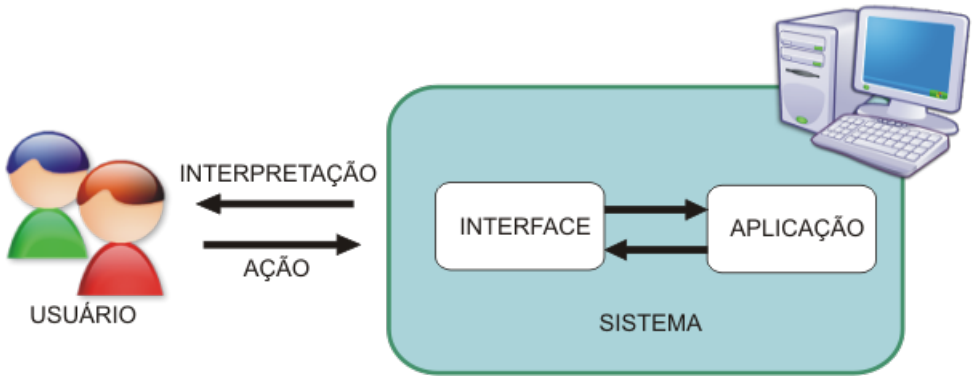
3 INTERAÇÃO HUMANO-COMPUTADOR(IHC)

Este capitulo tem como objetivo apresentar os conceitos primordiais da interface, os objetivos dela, história e os ganhos ofertados ao elaborá-la.

O surgimento do conceito da interface no princípio, era compreendido como o hardware e o software com que o homem poderia se comunicar (ROCHA, 2003). Toda a comunicação que teria entre um ser humano e uma máquina, abrangia este conceito, desde uma atividade mais simples como uma leitura em uma tela, a atividades mais complexas como desenvolvimento de um software. A evolução deste conceito levou a inclusão dos aspectos cognitivos e emocionais do usuário durante a comunicação (ROCHA, 2003).

Na figura 11, é ilustrado a comunicação entre uma pessoa e a máquina. O humano é representado pelo usuário, e a ação de comunicação são representadas pelas setas de interpretação e ação, a interface recebe as comunicações dos usuários e se comunica com a parte logica do software, aplicação.

FIGURA 11 – Interação Humano Computador..



FONTE: (SANTOS., 2012).

A interface pode ser visualizada como um lugar onde ocorre o contato entre duas entidades, homem e máquina, um exemplo, a tela de um computador (ROCHA, 2003). Com este exemplo, pode-se estender a varias situações, como, maçanetas de porta, botões de elevadores.

Uma definição que englobaria estes casos, seria, que a interface é uma superfície de contato que possui propriedade que alteram o que é visto, ou sentido, e alterando o controle da interação (Laurel, 1993).

IHC não possui uma definição estabelecida, mas a que mais a representa é, uma disciplina preocupada com o design, avaliação e implementação de sistemas computacionais interativos para uso humano e com o estudo dos principais fenômenos ao redor deles (ROCHA, 2003). Na figura 12, é ilustrado esta definição.

FIGURA 12 – Interação Humano-computador Adaptada da Descrição do comitê SIG-CHI 1992.

FONTE: (ROCHA, 2003).

4 TRABALHOS RELACIONADOS

4.1 AMBIENTE DE PROGRAMAÇÃO VISUAL BASEADO EM COMPONENTES

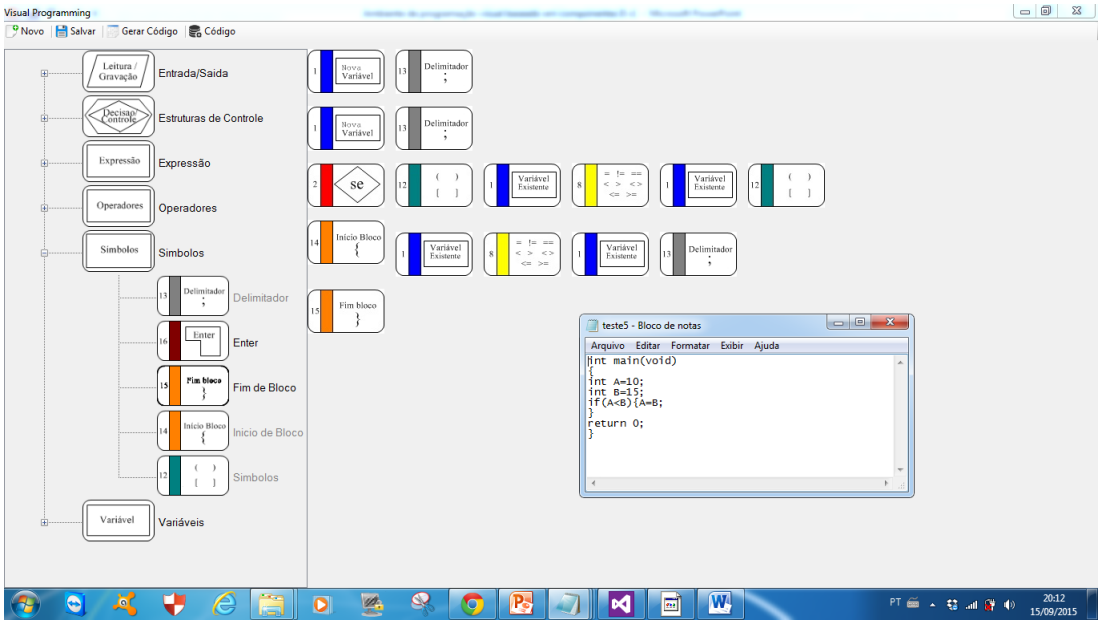
(BURIGO, 2015) apresenta um trabalho sobre a criação de um ambiente de desenvolvimento utilizando componentes vindos de fluxogramas, e transformando-os em código da linguagem C, para que auxilie pessoas que estão iniciando na área de programação, e também pessoas que já tem familiaridade na área, mas que não possuem conhecimento da linguagem ao qual o fluxo é convertido.

Neste trabalho foi utilizada a linguagem C#, fluxogramas e o banco de dados SQL Server. A linguagem C#, foi escolhida devido a sua facilidade em criar um ambiente gráfico e de sua administração com imagens como fluxograma.

A utilização de componentes baseados em fluxogramas auxilia no desenvolvimento de um ambiente de programação visual, possibilitando o usuário criar código através destes componentes.

A figura 13 ilustra a interface criada pelo trabalho, tendo o menu esquerdo as funções possíveis de utilização, e a aba da direita sendo o resultado dos conjuntos escolhidos.

FIGURA 13 – Exemplo de fluxo com o código gerado.



FONTE: (BURIGO, 2015).

SQL Server é um gerenciador de dados. Nele foram armazenados os fluxogramas para facilitar o processo de alocação, controle e manipulação, armazenando no final o código de saída do fluxograma montado. (BURIGO, 2015).

Foram realizados experimentos com 50 alunos, porém apenas 20% responderam

o questionário proposto, e suas respostas informavam que o software chama a atenção para seu uso, porém não era mais simples o seu entendimento, obrigando o usuário a ter um conhecimento de lógica de programação maior do que o desejado, assim não atingindo seu objetivo.

4.2 SCC: UM COMPILADOR C COMO FERRAMENTA DE ENSINO DE COMPILADORES

(FOLEISS GUILHERME P. ASSUNÇÃO, 2009) apresenta um trabalho do desenvolvimento de um compilador que permite criar programas na linguagem C e serem executados com supervisão em tempo real. Estas supervisões funcionam com uma execução detalhada, passo a passo. Com este recurso tem se um auxílio no aprendizado de compiladores, e suas etapas da geração do código.

Foram utilizadas a linguagem C e Assembly juntamente com a ferramenta SASM, que é um software que gera códigos objetos compatíveis com a arquitetura IA-32.

A maior parte do trabalho foi desenvolvido na linguagem C, porém algumas das rotinas básicas foram desenvolvidas em Assembly para melhorar o desempenho. Utilizando também o SASM, para testar a compatibilidade do compilador com esta ferramenta.

A figura 14 ilustra uma árvore de símbolos como saída de um código escrito para este compilador.

4.3 COMPILER BASIC DESIGN AND CONSTRUCTION

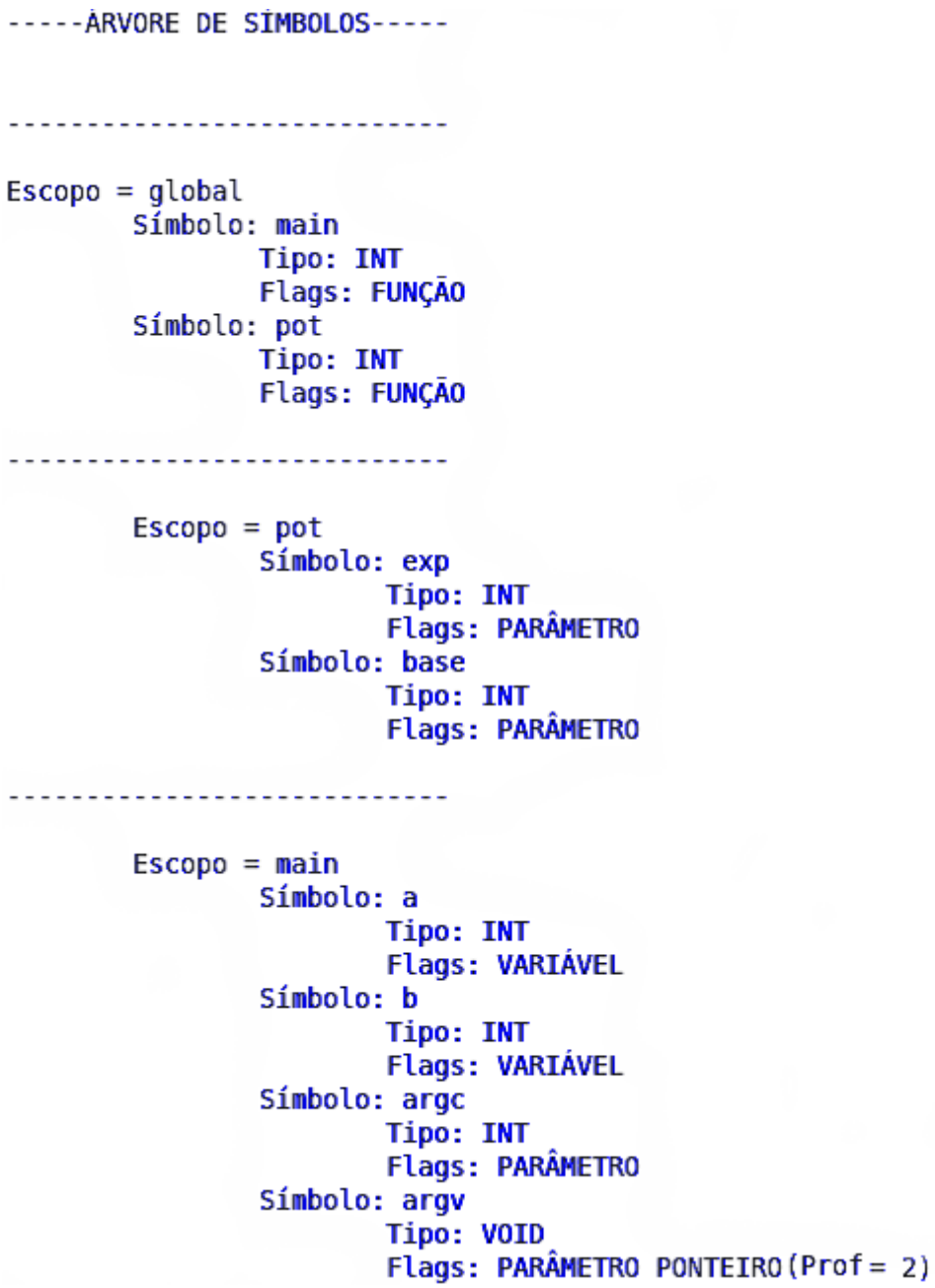
(JAIN NIDHI SEHRAWAT, 2014) apresenta um trabalho com sistema de compilação adaptativa, com o objetivo de fornecer uma documentação sobre o projeto e desenvolvimento do compilador, para auxiliar na compreensão do tema e criar técnicas eficazes para desenvolver.

Neste trabalho foi utilizada a linguagem Scheme para a implementação do compilador, e código de montagem (Assembly) como linguagem alvo. As técnicas implementadas foram análise léxica e análise sintática. A análise léxica tem como objetivo verificar a parte gramática de acordo com as regras da linguagem criada ou utilizada, validando o que está correto ou não. A análise sintática, é responsável por verificar a ordem dos símbolos e sentido.

4.4 INTERPRETADOR/COMPILADOR PYTHON

(BASTOS, 2010) apresenta um trabalho sobre o funcionamento da arquitetura do Python, analisando os processos de análise léxica, sintática e a geração de código.

FIGURA 14 – Árvore de símbolos.



FONTE: (FOLEISS GUILHERME P. ASSUNÇÃO, 2009).

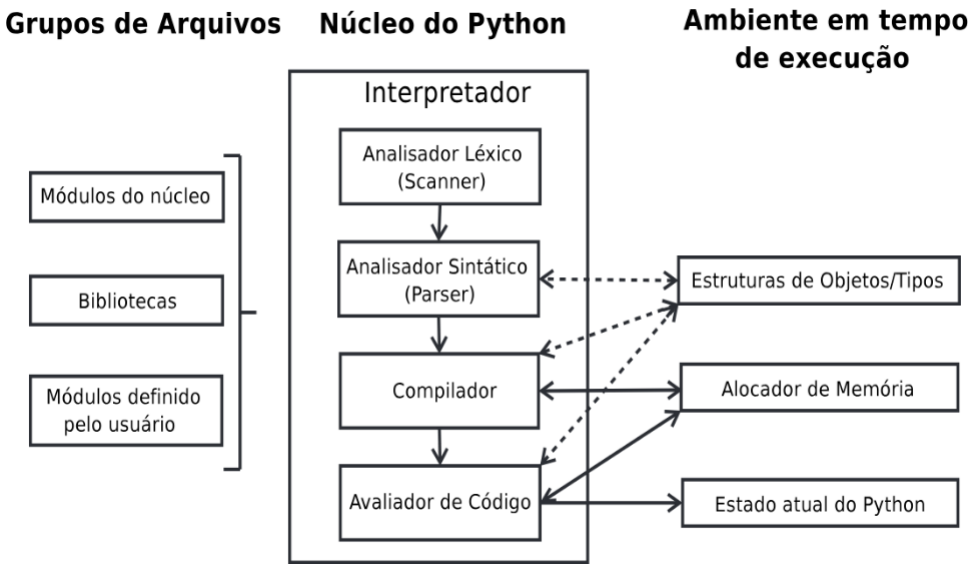
Este artigo realiza uma abordagem sobre a estrutura da arquitetura, do interpretador, a gramática da linguagem e as características.

Neste artigo foi utilizada a linguagem Python, uma linguagem de programação interpretada interativa e orientada a objetos, junto com o seu interpretador.

Após os estudos foi verificado que o interpretador obteve uma implementação diferenciada dos demais nos processos de análise léxica e análise sintática. Foi observado também que o Python utiliza máquina virtual para executar os códigos intermediários(bytecodes).

A figura 15 ilustra a arquitetura Python, tendo no centro ó interpretador, e nele os passos seguidos para a compilação do código.

FIGURA 15 – Interpretação da Arquitetura Python.



FONTE: (BASTOS, 2010).

4.5 COMPILER CONSTRUCTION

(SINGH SONAM SINHA, 2013) apresenta um artigo informando técnicas e exemplos, que facilite a implementação de um compilador.

Neste artigo é apresentado exemplos das técnicas de analise léxica e analise semântica, também mostrando trechos de códigos para melhor compreensão.

Foi utilizado a linguagem Scheme para a construção do compilador, e a linguagem de montagem, código de máquina (Uma linguagem composta apenas de números na base binaria), como linguagem alvo. Schema é uma linguagem que suporta programação funcional e procedural, facilitando assim a elaboração do compilador. O compilador criado utilizou de um gerenciamento de armazenamento do tipo pilha, essa estrutura tem a característica o maneja mento como, o último elemento a entrar nela, é o primeiro a sair.

QUADRO 1 – Comparação dos trabalhos relacionados.

Trabalho	**Mètodo**	**Gera uma saída**	**Ferramentas utilizaas**	**Linguagens utilizadas**
Trabalho 1	Componentes de fluxogramas; Administração de Imagens.	Código na linguagem C.	SQL Server; Fluxogramas.	C#; SQL
Trabalho 2	Análise sintática recursiva descendente	Árvore sintática; Árvore de Símbolos.	SASM.	C; Assembly
Trabalho 3	Análise Léxica; Análise Sintática.	Código de Montagem (Assembly).	-	Scheme; Assembly.
Trabalho 4	Análise Léxica; Análise Sintática.	-	Interpretador Python.	Python.
Trabalho 5	Análise Léxica; Análise Sintática.	Código de Máquina.	-	Scheme.

FONTE: próprio autor.

5 METODOLOGIA

Neste capitula é informado quais os métodos e ferramentas foram utilizados para desenvolver o interpretador e a interface, a forma de abordagem para a coleta de dados, o cenário e os indivíduos participantes.

A metodologia seguida para se obter dados sobre a utilização e a funcionalidades para os alunos e para o professor da matéria de compiladores, é a utilização de um questionário elaborado para ser respondido após o uso do software, este questionário está no apêndice. As tecnologias utilizadas para o desenvolvimento do interpretador, são: linguagem de programação C++, QT Creator, JFLAP, e as metodologias de criação de um compilador, Análise Léxica, Análise Sintática e tabela de símbolos.

5.1 LINGUAGEM C++

A linguagem utilizada para o desenvolvimento deste software é C++, que é a predominante no programa QT Creator, e por se diferenciar pouco da linguagem C, a base no ensino no curso de ciências da computação. Esta linguagem também possui bibliotecas que auxiliam no desenvolvimento, como o regex, que é utilizado neste trabalho para tratar expressões regulares e assim tornar o código melhor escrito sem necessidade de repetição de regra.

Na figura 16 é ilustrado a função descrita acima, nela é chamada a função regex_match, está função verifica se a letra passada, a variável line[aux], faz parte da gramatica enviada junto, regex("[a-zA-Z_]").

FIGURA 16 – Função do codigo de validação de caractere.

```
625  bool caracterValidationFirst() {
626      return (regex_match(string(1, line[aux]), regex("[a-zA-Z_]")));
627  }
```

FONTE: O próprio autor.

5.2 QT CREATOR

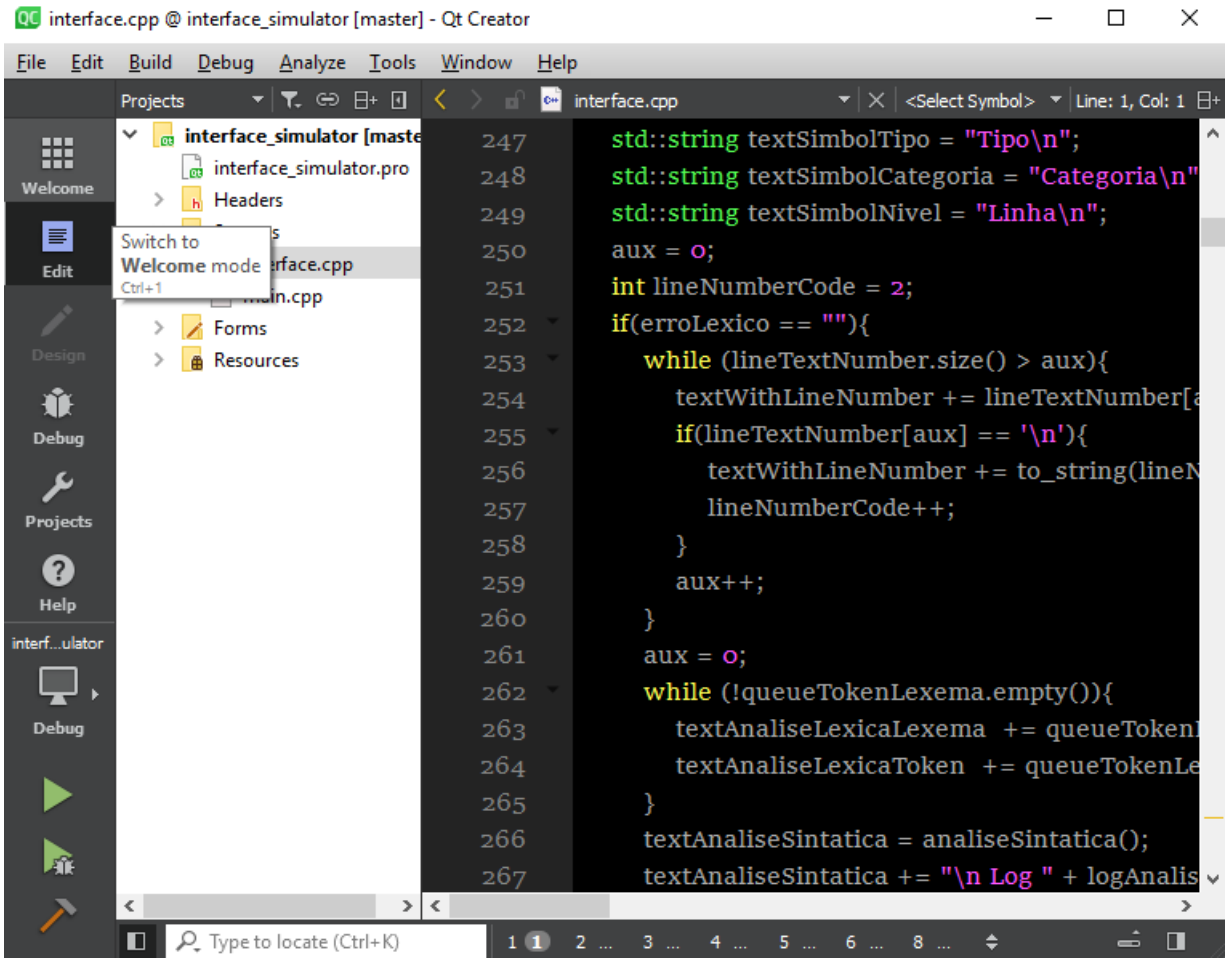
QT Creator é um ambiente de desenvolvimento integrado (IDE), criado por uma empresa norueguesa Trolltech, que visa facilitar e ajudar no desenvolvimento de softwares, permitindo criar sistemas para plataformas múltiplas. A versão utilizada é a 5.13.1, com o compilador MinGW 7.3.0 32 bits.

A principal razão da escolha desta IDE foi a facilidade de desenvolvimento, devido a possuir um compilador integrado, uma depuração excelente que possibilita

que o desenvolvedor encontre o erro no código produzido, e o corrija rapidamente, além de ter familiaridade com está framework por já utilizá-la em trabalhos da universidade.

Na figura 17 é ilustrado a interface do Qt Creator, e as opções de funcionalidades no lado esquerdo, sendo o ícone de triangulo deitado verde a função de compilação e o ícone de triangulo verde junto a um inseto o de depuração.

FIGURA 17 – Interface de código do Qt Creator.



FONTE: O próprio autor.

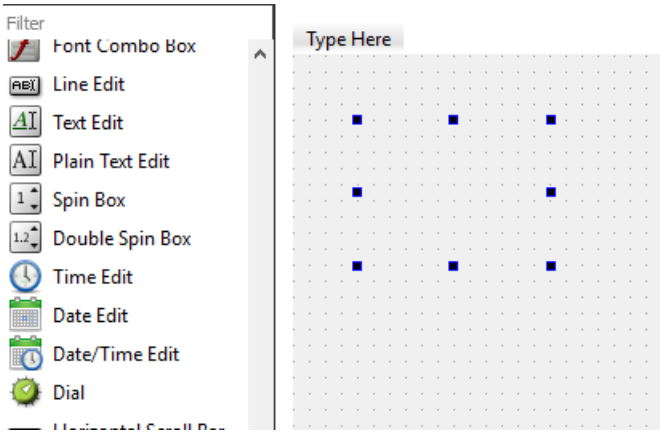
Este software da a possibilidade de elaborar a interface, com componentes básicos prontos necessitando apenas configura-los, assim centralizando tanto a Back-End (parte logica do software) como o Front-End (interface).

Na figura 18 é ilustrado a interface de do Qt Creator para a criação de interfaces, no lado esquerdo se encontra componentes já criados, sendo necessários apenas selecionalos e configura-los.

5.3 JFLAP

JFLAP é um software gratuito educacional desenvolvido na linguagem JAVA por Susan H. Rodger, o principal uso deste framework é na criação de autômatos finitos

FIGURA 18 – Interface de design do Qt Creator.



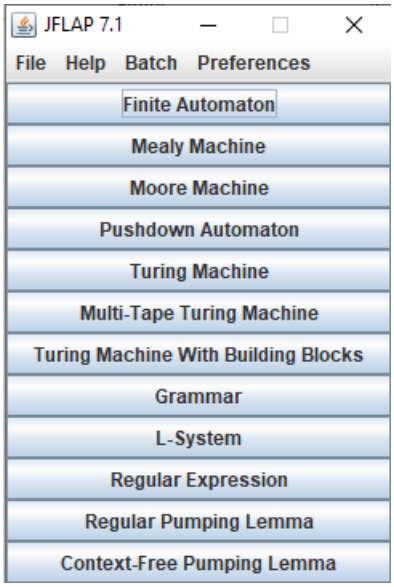
FONTE: O próprio autor.

não determinísticos, maquinas de Turing e vários tipos de gramatica (RODGER, 2005).

Este software foi utilizado para criar todos os autômatos da linguagem D+, utilizados para ilustrar na interface do interpretador por onde o código passou. Na figura 16 é ilustrado um autômato criado no JFLAP, ele é apresentado na interface desenvolvida quando aquele caminho de autômato é acessado pela analise léxica. Na figura 17 é ilustrado um autômato que não foi acessado na análise léxica).

Na figura 19 é ilustrado as opções que o software oferta para o usuário, desde autômato finito, gramáticas, máquina de Turing entre outros. Neste trabalho foi utilizado apenas a funcionalidade de autômato finito.

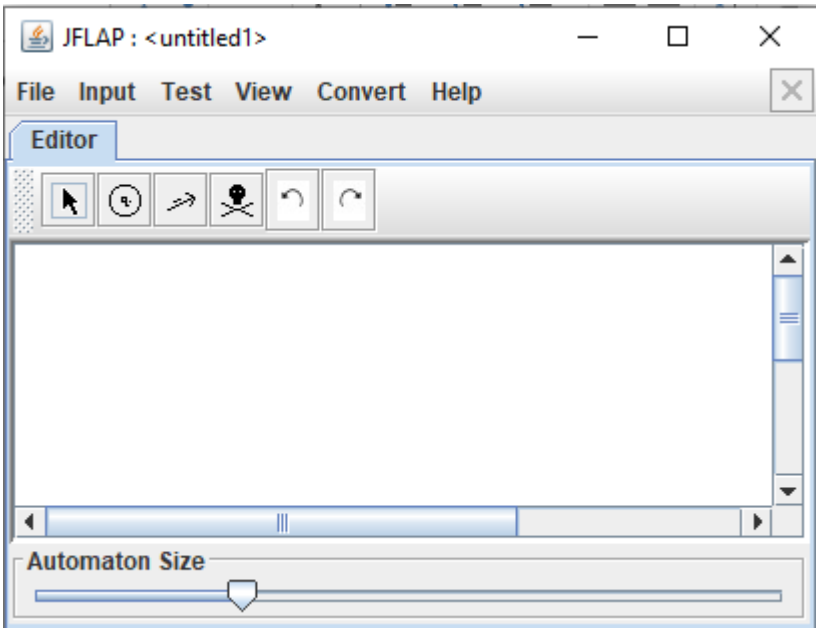
FIGURA 19 – Menu JFLAP.



FONTE: O próprio autor.

Na figura 20 é ilustrado o quadro que a framework abre para a criação dos autômatos.

FIGURA 20 – Janela de criação de autômatos.



FONTE: O próprio autor.

5.4 AUTÔMATOS

A criação dos autômatos da linguagem d+ foram criados para facilitar no desenvolvimento da análise léxica e posteriormente para a ilustração do percurso do código escrito. Foi desenvolvido uma função para cada estado do autômato.

Foram criado um total de 20 autômatos, exportados como png para utilizá-los na interface, e assim ilustrar para quem estiver utilizando o interpretador.

Na figura 21 é ilustrado um autômato criado na ferramenta JFLAP. Ele demonstra o caminho de validação de um identificador ou palavra reservada, onde sua gramática permite letras de A à Z tanto minúsculas ou maiúscula.

5.5 ESTRUTURAS ADOTADAS NO DESENVOLVIMENTO

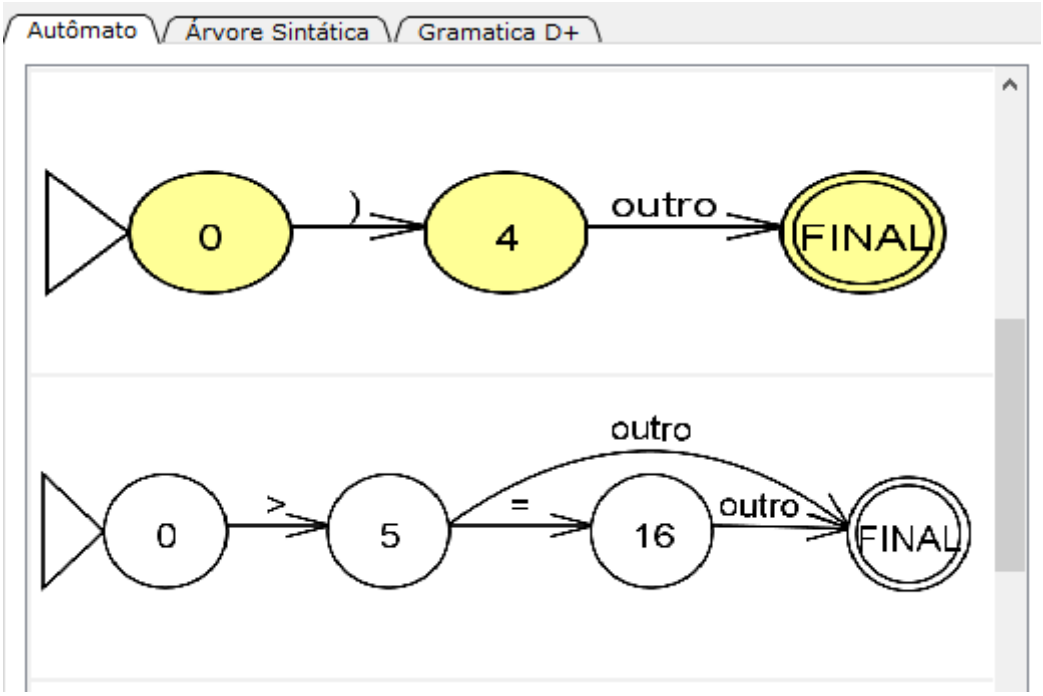
Neste capitulo são tratadas as estruturas utilizadas para o desenvolvimento da lógica da compilação, programação estruturada, Análise léxica, Análise Sintática, e tabela de símbolos.

5.5.1 Programação Estruturada

A estrutura de programação estruturada foi a escolhida no desenvolvimento do software, por conta de ser a base ensinada no curso de ciências da computação, e com isso ter maior familiaridade com esta estrutura.

A programação estruturada tem como base três mecanismos em que os blocos

FIGURA 21 – Autômotos na interface.

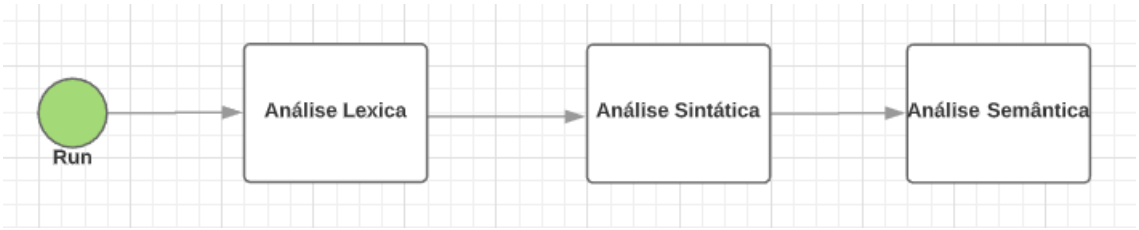


FONTE: O próprio autor.

de códigos se interligam, os mecanismos são: sequência, seleção e iteração(RICARTE, 2003).

A sequência é o fluxo que o código toma de acordo com a ação(RICARTE, 2003), um exemplo é o fluxo que o interpretador desenvolvido toma quando é clicado no botão de Run, ele carrega o código inserido, efetua a análise léxica em seguida a análise sintática e por fim a analise semântica. Na figura 22 é ilustrado um exemplo de sequência.

FIGURA 22 – Fluxo de sequência.

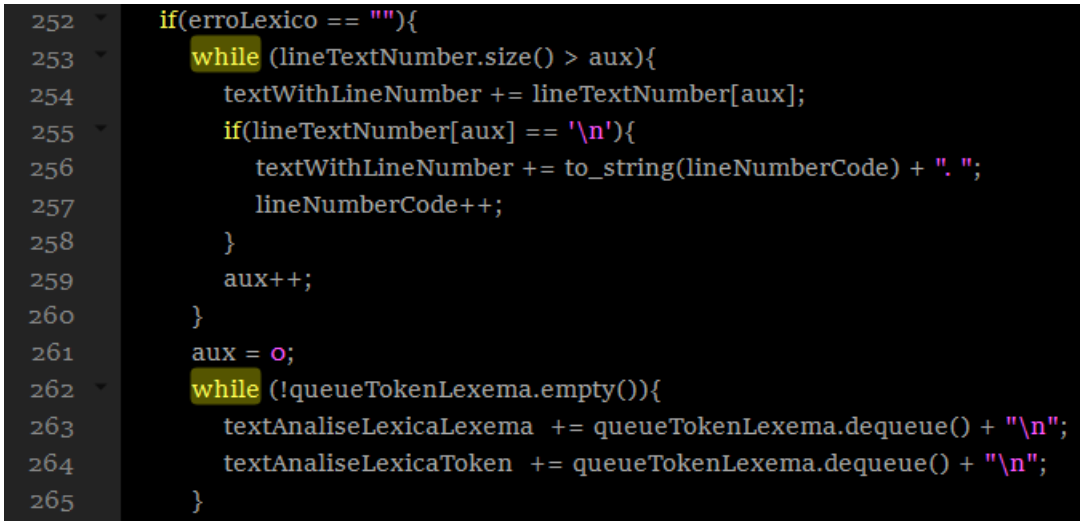


FONTE: O próprio autor.

A seleção é a verificação do caminho que será tomado com uma verificação do comando IF(RICARTE, 2003), um exemplo encontrado no código, é uma verificação que se faz andes de iniciar a análise sintática, caso haja um erro na analise léxica, ele não prossegue para a análise sintática. Na figura 23 é ilustrado o exemplo de seleção.

A iteração é o mecanismo de repetição, enquanto uma condição for atendida ela é executa(RICARTE, 2003), um exemplo encontrado no código é na função state01,

FIGURA 23 – Exemplo de iteração usando o comando while.

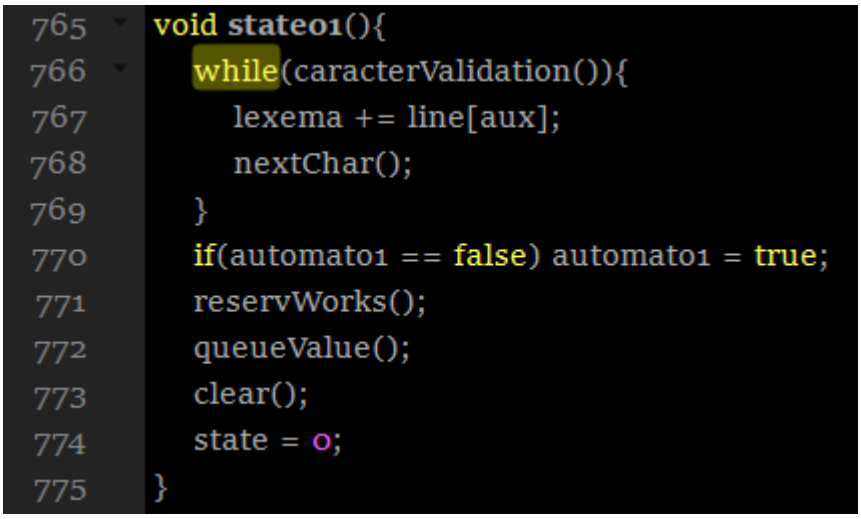


```
252  if(erroLexico == ""){
253     while (lineTextNumber.size() > aux){
254        textWithLineNumber += lineTextNumber[aux];
255        if(lineTextNumber[aux] == '\n'){
256           textWithLineNumber += to_string(lineNumberCode) + ". ";
257           lineNumberCode++;
258        }
259        aux++;
260     }
261     aux = 0;
262     while (!queueTokenLexema.empty()){
263        textAnaliseLexicaLexema  += queueTokenLexema.dequeue() + "\n";
264        textAnaliseLexicaToken  += queueTokenLexema.dequeue() + "\n";
265     }
```

FONTE: O próprio autor.

nesta função existe uma estrutura de repetição, while, em que e verificado se a letra de uma variável satisfaz a uma regra, caso sim, refaz os passos contidos dentro desta estrutura, ao contrário segue o código abaixo dela. Na figura 24 é ilustrado um exemplo de iteração.

FIGURA 24 – Exemplo de seleção.



```
765  void stateo1(){
766     while(caracterValidation()){
767        lexema += line[aux];
768        nextChar();
769     }
770     if(automato1 == false) automato1 = true;
771     reservWorks();
772     queueValue();
773     clear();
774     state = 0;
775  }
```

FONTE: O próprio autor.

O projeto esta organizado de acordo com 4 pastas, headers, sources, forms e resources.

A pasta headers contem todas os arquivos .h produzidos no desenvolvimento, neles está as declarações de variáveis, contantes e declarações de funções, estes arquivos estão organizados de acordo com sua respectivas analises, o arquivo analex.h contem as funções da analise léxica, o aquivo anasin.h contem as funções da analise sintática, anasem.h contem as funções de analise semântica e o arquivo interface.h

contem as funções da interface.

A pasta sources contem os arquivos .cpp, nela são encontrado toda a implementação da logica da framework, o aquivo main.cpp é o que inicializa o software e o arquivo interface.cpp é onde está a implementação das analises.

A pasta forms contem o arquivo interface.ui, este arquivo abrange todo o código da interface criada para o interpretador.

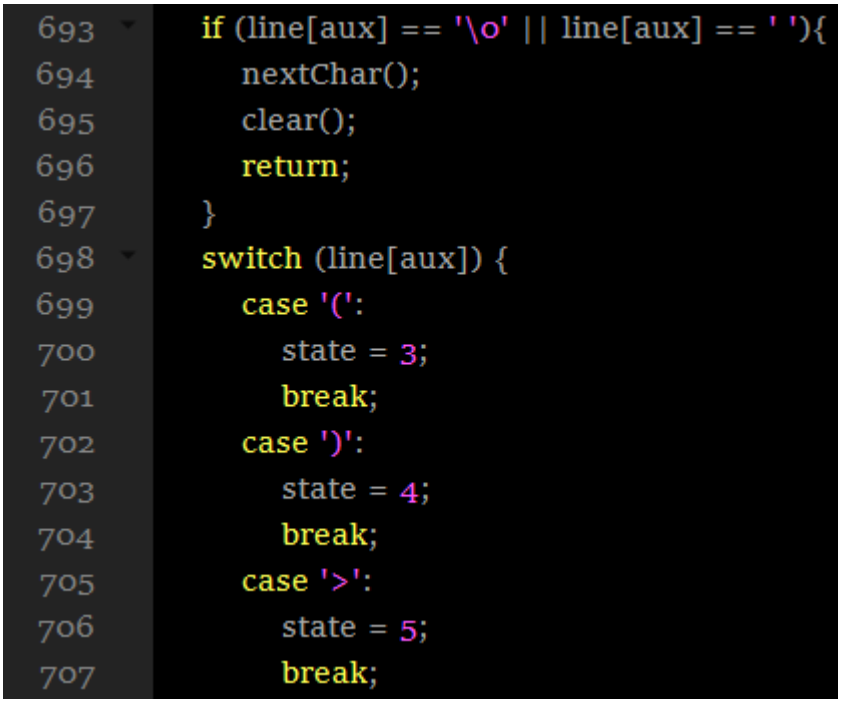
A pasta resources guarda todas as imagens utilizadas no softwa're, sendo elas os ícones e os automatos.

5.5.2 Análise léxica

Para o desenvolvimento da analise léxica, foram utilizados os autômatos criados, em apêndice, como guias, o autômato do estado 0 é o que seleciona o caminho que aquele caractere ira percorrer, o mesmo é feito na função state00, ao encontrar o caminho é atribuído um valor para a variável state sendo o valor o número do próximo estado, e assim prossegue para o estado subsequente.

Na figura 25 é ilustrado um trexo da função state00, nela é verificado a letra que o programa está analisando, caso faça parte da gramatica, é direcionada para o caminho correto, do contrário é retornado erro.

FIGURA 25 – Função state00.



FONTE: O próprio autor.

A função reservWorks contém a lógica para verificar se aquele lexema formado

é uma palavra reservada, a função possui um laço de repetição, for, em que percorre uma matriz criada que contém todas as palavras reservadas da linguagem D+, casso seja igual a alguma delas, é atribuído o token da palavra reservada, do contrário o token é atribuído como identificador. Na figura 26 é ilustrado a função reservWorks.

FIGURA 26 – Função reservWorks.

```
607  void reservWorks () {
608      lexemaMaiusculo();
609      int i = 0;
610      for (i = 0; i < 45; i++ ){
611          if (lexema == tableReservWorks[i][0]){
612              token = tableReservWorks[i][1];
613              return;
614          }
615      }
616     token = "IDENTIFICADOR";
617  }
```

FONTE: O próprio autor.

Após a análise obter os valores de lexema e token, é chamada a função queueValue, que atribui os valores do lexema e token em uma estrutura de fila, está fila é utilizada para passar estes valores para uma string que é utilizada para ilustrar em um campo text na interface do software, na figura 27 é ilustrada a saída deste string na aba Análise léxica.

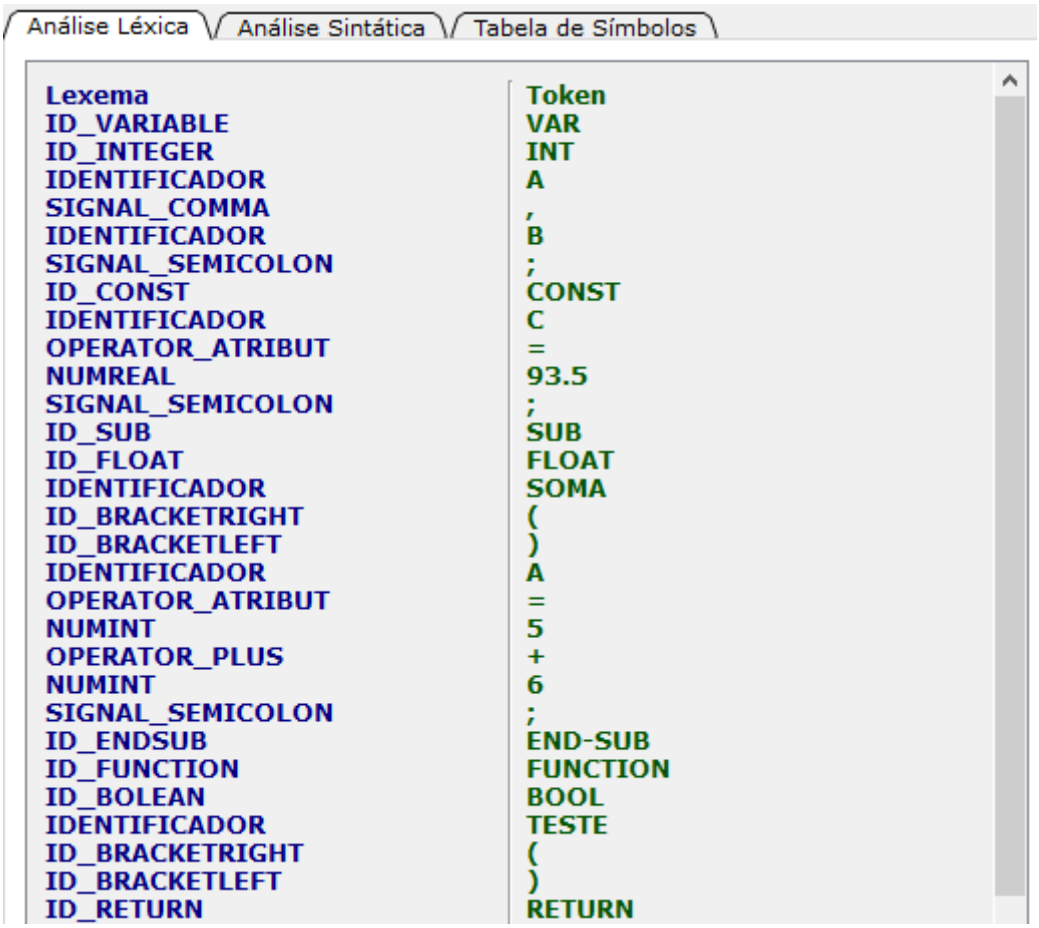
Para identificar por qual autômato o código passou é utilizando de variáveis globais booleanas, em que ao acessar um estado essas funções criadas como state01, state02, são atribuídos true para a variável correspondente aquele estado, como a variável automato1. No fim de toda compilação é selecionada a imagem de todos os autômatos para as labels da interface, caso aquela variável do autômato for true ele troca a imagem preto e branco para a colorida.

Na figura 28 é ilustrado o trecho do código que valida se o que sera carregado é o automato colorido ou o preto e branco.

Após validar quais variáveis dos autômatos são true, e atribuir a imagem correta, a interface é carregada.

Na figura 29 é ilustrado os dois casos, um autômato em preto e branco, quando não é acessado e outro colorido, quando é acessado.

FIGURA 27 – Saída da análise léxica.



FONTE: O próprio autor.

FIGURA 28 – Verificação do automato a ser carregado.

```
308     if(automato1){
309         QPixmap automatoState1(":/automatos/state1.png");
310         ui->automato1->setPixmap(automatoState1.scaled(480,150));
311     }else{
312         QPixmap automatoState1(":/automatos/stateDisabled1.png");
313         ui->automato1->setPixmap(automatoState1.scaled(480,150));
314     }
```

FONTE: O próprio autor.

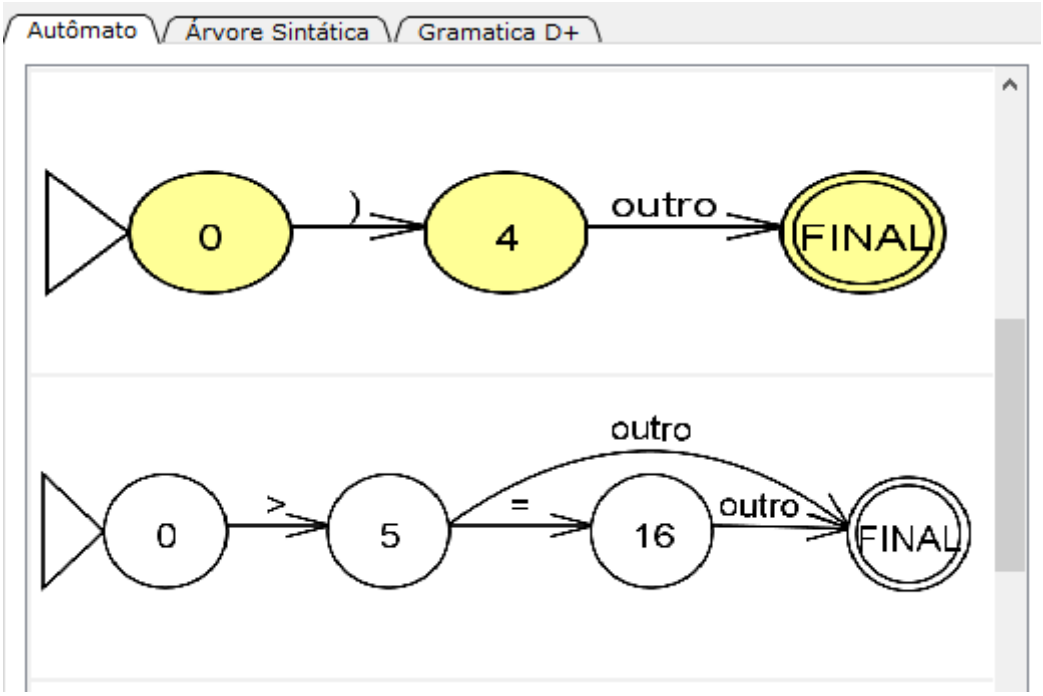
5.5.3 Análise sintática

Para o desenvolvimento da análise sintática, foi utilizado a criação de uma matriz que armazena todos os tokens e lexemas encontrados após a análise léxica, esta matriz é utilizada para verificar a sequencia de tokens, e validá-los se estão na ordem correta.

Após finalizar a análise léxica o software chama em sequência a rotina para a analise sintática, está inicia a verificação na posição 0 da matriz tokensLexemasTable, e percorre-a validando junto das regras da gramatica.

Nos quadros 2, 3 e 4 são ilustrados as funções criadas referentes a cada regra,

FIGURA 29 – Interface com os autômatos.



FONTE: O próprio autor.

seguindo uma logica para verificar se o próximo elemento da matriz corresponde com o elemento esperado.

Todas as função criada para satisfazer as regras da gramática possuem uma operação de incremento na variável chamada tamanho, esta variável global tem como função ser o index da matriz que contem todos os lexemas e tokens, tokensLexemasTable, com isto obtém a localização que se encontra o percurso tomado dentro da matriz, o valor daquela posição é comparado com o token esperado na regra que a análise se encontra, caso seja igual é seguido o caminho até finalizar a regra, do contrário é retornado erro.

Na figura 30 ilustra a função DV que corresponde a regra decl-var, nela é efetuada uma chamada para função ET e em seguida a função LV, após estas chamadas ocorrerem e não retornarem erro, ele valida se possui o lexema de (;), casso possua, e retornado uma mensagem de sucesso ao log, ao contrário retorna a mensagem de erro, a variável keySintatico funciona como uma chave, sempre que é encontrado um erro no código, é atribuído o valor 1, assim o software para de efetuar a analise sintática, por já ter encontrado erro.

No processo de desenvolvimento desta análise, foi identificado um problema na gramática, o comando “ DO B WHILE exp ; ”, da forma que foi desenvolvido a lógica impossibilitava este comando, a implementação seguia o conceito de recursividade, onde uma função é chamada por ela mesma, seguindo este conceito caso existisse

QUADRO 2 – Declaraçoes.

Regra	**Função**
programa -> lista-decl	P()
lista-decl -> lista-decl decl | decll	LD()
decl -> decl-const | decl-var | decl-proc | decl-func | decl-main	D()
decl-const -> CONST ID = literal ;	DC()
decl-var -> VAR espec-tipo lista-var ;	DV()
espec-tipo -> INT | FLOAT | CHAR | BOOL | STRING	ET()
decl-proc -> SUB espec-tipo ID (params) bloco END-SUB	DP()
decl-func -> FUNCTION espec-tipo ID (params) bloco END-FUNCTION	DF()
decl-main -> MAIN ( ) bloco END	DM()
params -> lista-param | Épsilon	PR()
lista-param -> lista-param , param | param	LP()
param -> VAR espec-tipo lista-var BY mode	PM()
mode -> VALUE | REF	M()

FONTE: próprio autor.

no código um comando de bloco, o software chamava novamente a função LV e assim abria opções para chamar novamente comandos de repetição, porem todos as regras que possui o bloco também possuem um token de saída do bloco, como “WHILE exp B LOOP”, o token loop informa que o bloco acabou, mas no comando “ DO B WHILE exp ; ” era confundido como um novo comando de repetição do WHILE, por conta de não conseguir sair do bloco, e não distinguir se o token atual era um token de saída, a palavra desta regra foi alterada para AS, resolvendo a questão.

Após finalizar a análise sintática é exibido na interface um log do percurso tomado, apresentando uma mensagem de sucesso ou erro, e em seguida mensagens de quais regras foram validadas e em qual regra o interpretador encontrou o erro. Na mensagem de erro é apresentado qual foi o erro encontrado, como a falta de um (;) ou de uma palavra reservada como END, e o numero da linha que ocorreu o erro.

Na figura 31 é ilustrado a saída da interpretação de um código, é apresentado uma mensagem de sucesso na regra de VAR na linha 1, porém em seguida informado um erro na linha dois e a causa do erro.

QUADRO 3 – Comandos.

Regra	**Função**
bloco -> lista-com	B()
lista-com -> comando lista-com | Épsilon	LC()
comando -> cham-proc | com-atrib | com-selecao | com-repeticao | com-desvio | com-leitura | com-escrita | decl-var | decl-const	C()
com-atrib -> var = exp ;	CA()
com-selecao -> IF exp THEN bloco END-IF | IF exp THEN bloco ELSE bloco END-IF	CS()
com-repeticao -> WHILE exp DO bloco LOOP | DO bloco AS exp ; | REPEAT bloco UNTIL exp ; | FOR ID = exp-soma TO exp-soma DO bloco NEXT	CR()
com-desvio -> RETURN exp ; | BREAK ; | CONTINUE ;	CD()
com-leitura -> SCAN (lista-var) ; | SCANLN (lista-var) ;	CL()
com-escrita -> PRINT (lista-exp) ; | PRINTLN (lista-exp) ;	CE()
cham-proc -> ID (args) ;	CP()

FONTE: próprio autor.

FIGURA 30 – Função DV.

```
1325  void DV(){
1326      gramatica5 = true;
1327      ET();
1328      if(keySintatico == 1) return;
1329      LV();
1330      if(testValue("SIGNAL_SEMICOLON")){
1331          treeSintatico();
1332          logAnaliseSintatica += "\n Sucesso no VAR(S) :" + tokensLexemasTable[tamanho-1][0];
1333      }else{
1334          logAnaliseSintatica += "\nErro: Erro na declaração de VAR, faltando ( ; )!";
1335          keySintatico = 1;
1336      }
1337  }
```

FONTE: O próprio autor.

5.5.3.1 Árvore Sintática

Para a criação da arvore sintática é utilizado o caractere “\t” que implementa um espaçamento fixo, este espaçamento é utilizado para alinhar os terminais que fazem parte daquele nível da árvore, sabendo a quantidade de espaçamento que deve possuir cara impressão na árvore para se obter o efeito de ramo é utilizado uma variável global

QUADRO 4 – Expressões.

Regra	**Função**
lista-exp -> exp , lista-exp | exp	LE()
exp -> exp-soma op-relac exp-soma | exp-soma	EXP()
op-relac -> <= | < | > | >= | == | <>	OR()
exp-soma -> exp-mult op-soma exp-soma | exp-mult	EXPS()
op-soma -> + | - | OR	OS()
exp-mult -> exp-mult op-mult exp-simples | exp-simples	EXPM()
op-mult -> * | / | DIV | MOD | AND	OM()
exp-simples -> (exp) | var | cham-func | literal | op-unario exp	EXPSP()
literal -> NUMINT | NUMREAL | CARACTERE | STRING | valor-verdade	L()
valor-verdade -> TRUE | FALSE	VV()
cham-func -> ID (args)	CF()
args -> lista-exp | Épsilon	AR()
var -> ID | ID [exp-soma]	VAR()
lista-var -> var , lista-var | var	LV()
op-unario -> + | - | NOT	OU()

FONTE: próprio autor.

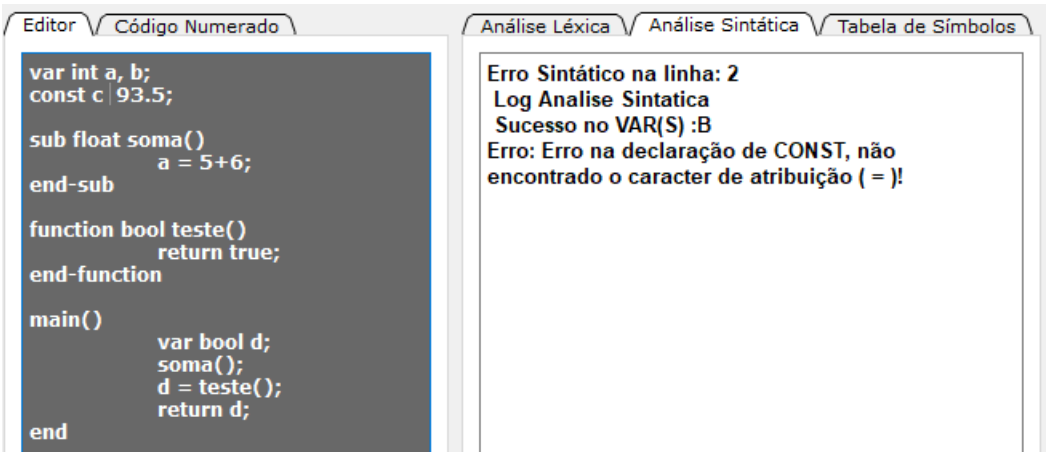
chamada NIVEL.

Todas as funções implementadas na análise sintática incrementam a variável NIVEL, e adicionam uma string da regra que a função corresponde na variável global treeTerminal, após estes processos é chamada uma função treeSintatica, este método forma árvore.

Na figura 32 é ilustrada a função treeSintatica, esta função possui uma estrutura de repetição while, que incrementa o caractere “\t” o número de vezes que a variável NIVEL possui como valor, esta parte da função implementa a estrutura da arvore, após este passo, é atribuído o caractere “|” para servir como referência a qual a palavra pertence ao ramo pai, por fim é atribuído o token a arvore, caso a variável treeTerminal for diferente de vazia, é adicionado aquilo que esta variável possui, do contrário é adicionado o lexema daquela posição.

Na figura 33 é ilustrado a saída da árvore de um código que efetua uma declaração de var, nela é possível ver por quais regras foram utilizadas na montagem.

FIGURA 31 – Log análise sintática.



FONTE: O próprio autor.

FIGURA 32 – Função treeSintatico.

```
1159  void treeSintatico(){
1160     int i =0;
1161     while(i < nivel){
1162        tree += "\t";
1163        i++;
1164        if(i == nivel){
1165           tree += "|        ";
1166        }
1167     }
1168     if(treeTerminal == ""){
1169        tree += tokensLexemasTable[tamanho][0] + "\n";
1170     }else{
1171        tree += treeTerminal + "\n";
1172        treeTerminal = "";
1173     }
1174  }
```

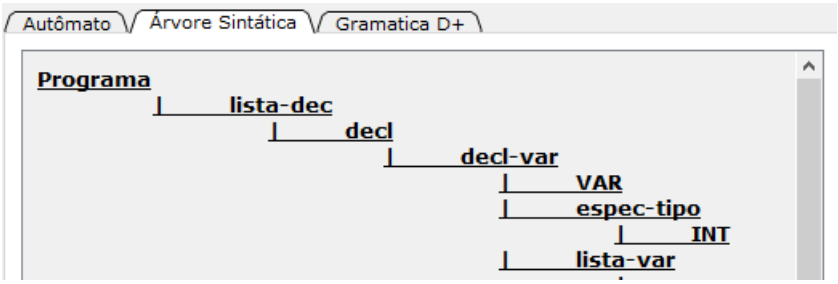
FONTE: O próprio autor.

5.5.3.2 Gramática

A gramática completa é apresentada na interface para que facilite no entendimento da linguagem, assim pode ser usado como consulta ou para retirar duvidas de uma regra especifica.

Após compilar o código inserido na interface é adicionado na aba da gramática as regras que abrange o código escrito, para desenvolver este comportamento é utilizado de uma variável global, similar ao dos autômatos, descrita no capítulo de análise léxica. Quando na análise sintática aquela regra específica é acessada, a variável global correspondente a regra, atribuída como false, é modificada para true. Após finalizar o processo da análise sintática, é verificado quais variáveis de cada regra

FIGURA 33 – Árvore Sintática montada.

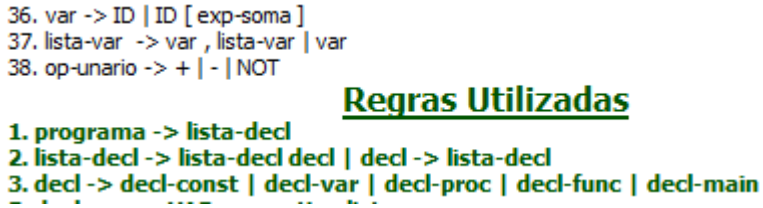


FONTE: O próprio autor.

possuem o valor true, os que possuem, é adicionado a regra na string que é passada para a interface da aba da gramática.

Na figura 34 é ilustrada a saída da aba gramática, as de cor verde são as regras utilizadas no código inserido no interpretador.

FIGURA 34 – Gramática utilizada.



FONTE: O próprio autor.

5.5.4 Tabela de Símbolos

O desenvolvimento da análise semântica se dá na criação da tabela de símbolos, está tabela tem como objetivo mostrar para o usuário do programa, os identificadores localizados pelo interpretador e trazer informações que auxiliem na compreensão de declaração de variáveis, constantes, procedures e funções.

O da criação da tabela de símbolos é feita na análise léxica, nela a função de atribuição queueValue, possui uma validação em que, se o token obtido for um IDENTIFICADOR e antes dele ter um token de CONST, VAR, SUB ou FUNCTION, este token e lexema é incrementado em uma matriz chamada simbolTable. Após finalizar o processo de análise léxica e análise sintática, é retirada os valores nela obtidos para ilustrar a tabela de símbolos.

Na figura 35 e ilustrado a tabela de símbolos presente na interface após efetuar todo o processo.

FIGURA 35 – Tabela de Símbolos.

FONTE: O próprio autor.

5.6 INTERFACE

Na criação da interface gráfica foi utilizando como base o editor de texto básico Notepad, e editores de desenvolvimentos simples como Code Blocks e VIsual Studio Code.

Na barra superior do programa foi incrementando funcionalidades simples mas práticas como, novo, que cria um arquivo novo, abrir, que abre uma janela para que possa procurar algum arquivo do formato d+ e seja carregado no software, salvar, run, que efetua os passos das análises no código escrito na framework, desfazer e refazer.

Na figura 36 é ilustrado a barra onde se encontra as funcionalidades descritas acima. Todos os ícones utilizados nesta interface são encontrados no site (https://www.flaticon.com).

FIGURA 36 – Barra de funcionalidade.

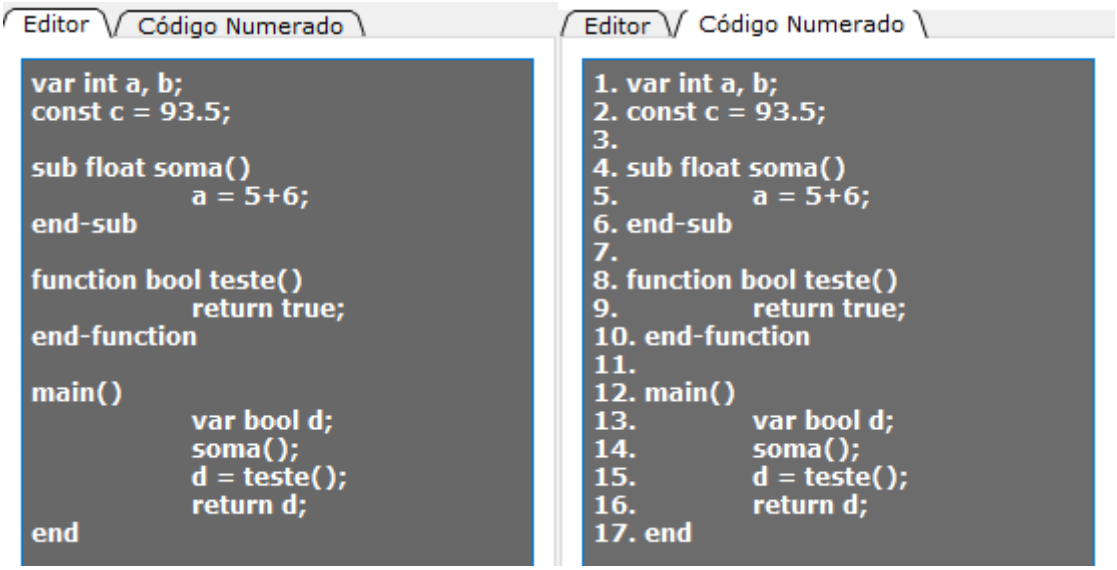
FONTE: O próprio autor.

A interface do software abrange todos os passos e resultados das análises implementadas, para melhor aproveitamento de espaço de tela e organizar de forma funcional, é utilizando de abas, similar ao dos navegadores como Google Chrome e Firefox. O usuário do programa tem a liberdade de selecionar por estas abas a informação que deseja que seja apresentada.

Na aba “editor” é o local em que insere o código d+, foi deixado um fundo acinzentado para auxiliar e destacar o local de escrita do código, na aba “Código Numerado”, é uma cópia do código inserido, porém numerado, para que assim seja mais fácil encontrar o erro acusado pelo log da análise sintática, esta funcionalidade foi implementada pois no decorrer do desenvolvimento, foi reparado que havia dificuldade para verificar qual linha havia um problema, mesmo passando o número desta.

Na figura 37 é ilustrada as duas abas descritas acima, no canto esquerdo a aba do “Editor” e no canto direito a aba do “Código Numerado”.

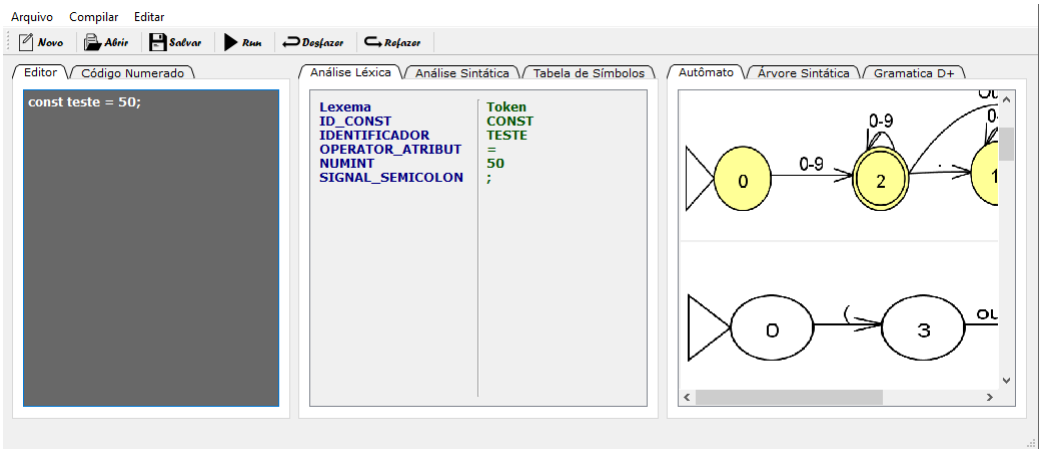
FIGURA 37 – Interface do editor.



FONTE: O próprio autor.

Na figura 38 é ilustrado uma visão geral da interface implementada.

FIGURA 38 – Interface do interpretador.



FONTE: O próprio autor.

REFERÊNCIAS

AHO RAVI SETHI, J. D. U. A. V. *Compiladores Princìpios, Tècnicas e Ferramentas*. 1. ed. Rio de Janeiro: Santuário, 1995. Acesso em: 10 mar 2019.

BASTOS, J. F. E. *Interpretador/Compilador Python*. 1. ed. Rio Grande do Sul: Universidade Catolica de Pelotas, 2010. Acesso em: 21 mar 2018.

BURIGO, J. M. *Ambiente de programação visual baseado em componentes*. 1. ed. Curitiba: UTP - Universidade Tuiuti do Paraná, 2015. Acesso em: 10 mar 2018.

FOLEISS GUILHERME P. ASSUNÇÃO, E. H. M. d. C. J. H. *SCC: Um Compilador C como Ferramenta de Ensino de Compiladores*. 1. ed. Maringá: Universidade Estadual de Maringá, 2009. Acesso em: 5 mar 2018.

FURLAN, D. C. *Regras de gramática D+*. [S.l.]: UTP, 2018.

JAIN NIDHI SEHRAWAT, N. M. M. *Compiler Basic Design and Construction*. 1. ed. India: Maharshi Dayanand University, 2014. Acesso em: 20 mar 2018.

MARANGON, J. D. *Compiladores para Humanos*. 2015. GitBook. Disponível em: <https://johnidm.gitbooks.io/compiladores-para-humanos/content/>. Acesso em: 14 jul 2019.

RICARTE, I. *Introdução a Compilação*. 1. ed. Rio de Janeiro: Elsevier - Campus, 2008. Acesso em: 20 mar 2019.

RICARTE, I. L. M. *Programação estruturada*. 2003. Faculdade de Engenharia Elátrica e de Computação FEEC - UNICAMP. Disponível em: <http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node7.html>. Acesso em: 10 nov 2019.

ROCHA, M. C. B. Heloísa Vieira da. *Design e Avaliação de Interfaces Humano-Computador*. 1. ed. Rio de Janeiro: Unicamp, 2003. Acesso em: 9 ago 2019.

RODGER, S. H. *JFLAP Version 7.1*. 2005. Jflap. Disponível em: <http://www.jflap.org/>. Acesso em: 30 oct 2019.

SANTOS., A. P. *A Importância da Interação Humano-Computador*. 2012. TIQx. Disponível em: <http://tiqx.blogspot.com/2012/02/compreenda-importancia-da-interacao.html>. Acesso em: 8 ago 2019.

SANTOS, T. L. P. R. *Compiladores da teoria a pratica*. 1. ed. Rio de Janeiro: LTC, 2018. Acesso em: 26 jul 2019.

SINGH SONAM SINHA, A. P. A. *Compiler Construction*. 1. ed. Gorakhpur: Institute of Technology e Management,GIDA Gorakhpur, 2013. Acesso em: 28 mar 2018.

VICTORIA, P. *Métodos de tradução: interpretador x compilador*. 2019. Imasters. Disponível em: <https://imasters.com.br/desenvolvimento/metodos-de-traducao-interpretador-x-compilador>. Acesso em: 20 mar 2019.

QUESTIONÁRIO DE UTILIZAÇÃO DO INTERPRETADOR INTEGRADO A UMA INTERFACE DA LINGUAGEM D+

DADOS PESSOAIS:

1. Sexo: ( ) Feminino ( ) Masculino 2.Idade (anos): _____

3. Nome: _____

4. Você possui experiência profissional em programação?
( ) Sim ( ) Não
Obs.: Se respondeu afirmativo para a "pergunta 8" responder a pergunta da sequência, do contrário siga para a pergunta nº 5.

5. Quanto tempo?
( ) menos de 6 meses ( ) de 6 meses a 1 ano ( ) de 1 a 2 anos ( ) mais de 2 anos

MATÉRIA COMPILADORES:

6. Qual nível de dificuldade você daria para a disciplina de compiladores?
( ) Baixo ( ) Regular ( ) Alto

7. Você consegue compreender a matéria sem grandes problemas?
( ) Sim ( ) Não

8. Um framework que ilustra os passos das análises, facilitaria no aprendizado?
( ) Sim ( ) Não

UTILIZANDO O INTERPRETADOR:

9. O Interpretador ajudou a compreender a matéria?
( ) Sim ( ) Não
Obs.: Se respondeu afirmativo para a "pergunta 8" responder a sequência de perguntas, do contrário siga para a pergunta nº 20.

10. Os tokens e lexemas deixaram claro a diferença entre ambos?
( ) Sim ( ) Não ( )Talvez

11. Foi possível verificar quais caracteres não fazem parte da linguagem?
( ) Sim ( ) Não ( )Talvez

( ) Sim ( ) Não ( )Talvez

13. Os Autômatos ilustrados auxiliaram na compreensão dos passos da Análise Léxica?
( ) Sim ( ) Não ( )Talvez

14. Ficou claro com os autômatos coloridos por onde o código percorreu?
( ) Sim ( ) Não ( )Talvez

15. A saída da Analise Sintática ficou clara?
( ) Sim ( ) Não ( )Talvez

16. Está claro por onde o código passou na Análise Sintática, com o auxílio do log?
( ) Sim ( ) Não ( )Talvez

17. Os erros encontrados na Análise Sintática são indicados de forma clara? com a ajuda da numeração das linhas?
( ) Sim ( ) Não ( )Talvez

18. A gramática da linguagem D+ auxilia na compreensão?
( ) Sim ( ) Não ( )Talvez

19. As regras da gramática adicionadas com a cor verde, ajuda a compreender quais regras foram utilizadas no código?
( ) Sim ( ) Não ( )Talvez

20. A árvore sintática gerada do código, facilita na ilustração da análise?
( ) Sim ( ) Não ( )Talvez

21. A árvore sintática é fácil de compreender?
( ) Sim ( ) Não ( )Talvez

22. A tabela de símbolos ficou clara para compreender os identificadores?
( ) Sim ( ) Não ( )Talvez

23. A tabela de símbolos auxiliou na compreensão?
( ) Sim ( ) Não ( )Talvez

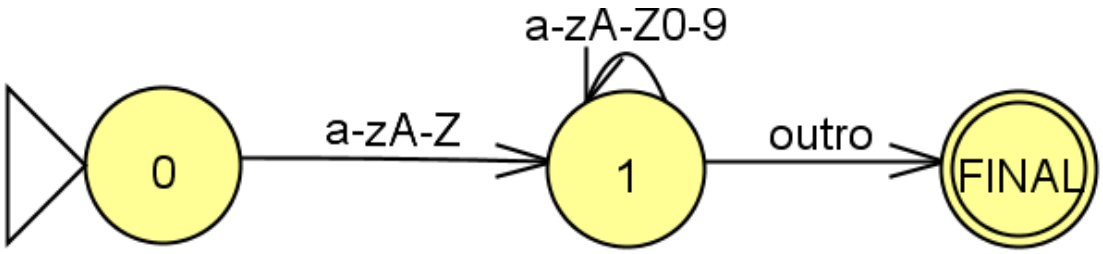
24. Qual parte do framework você achou mais interessante?
_____

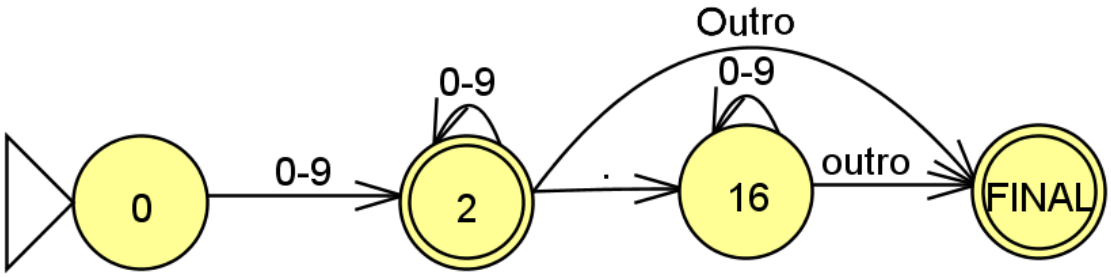
( ) Sim ( ) Não ( )Talvez

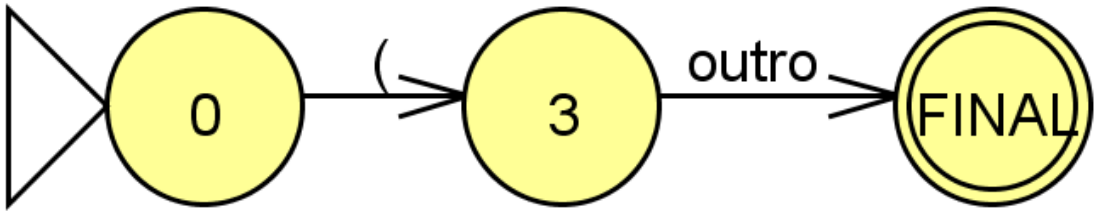
25. Se você tivesse acesso a esta ferramenta no início da matéria, te ajudaria na compreensão?
( ) Sim ( ) Não ( )Talvez

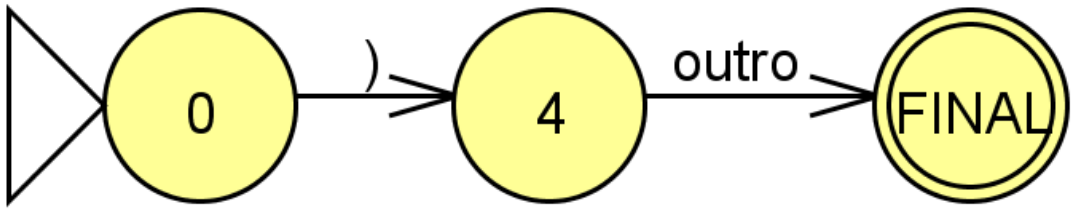
26. Você indicaria para um colega com dificuldade na matéria?
( ) Sim ( ) Não ( )Talvez

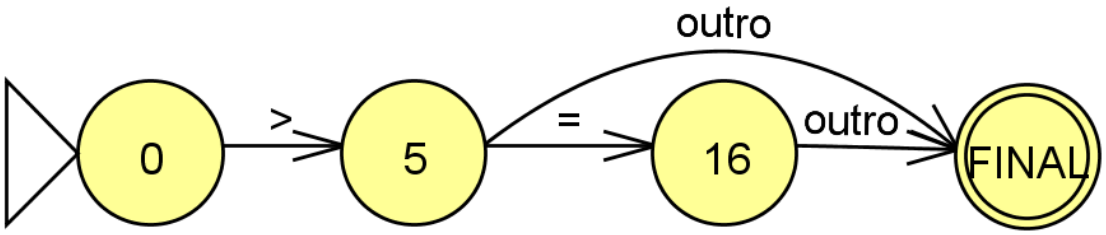
APÊNDICE B – AUTÔMATOS ATIVOS

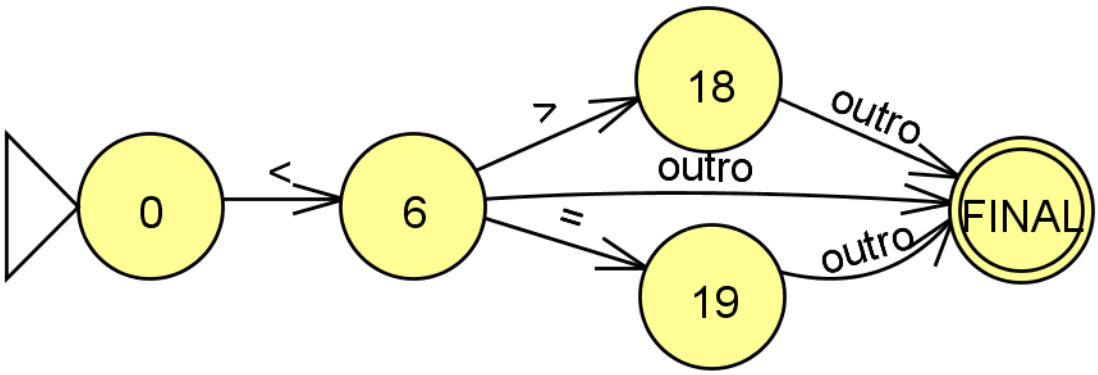


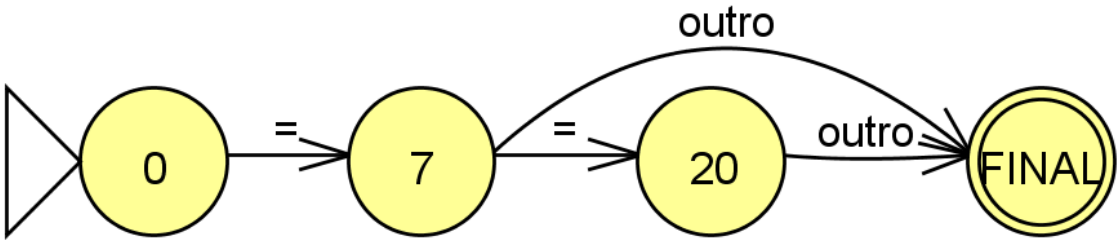


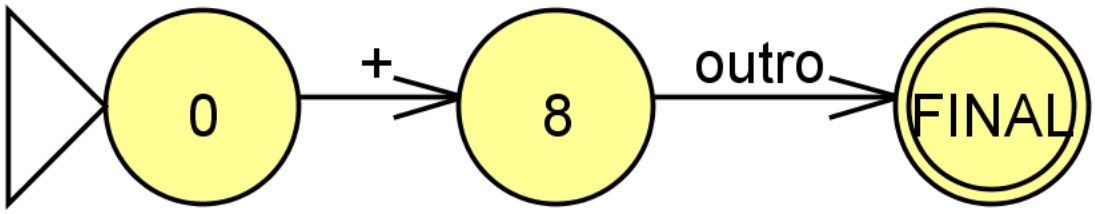


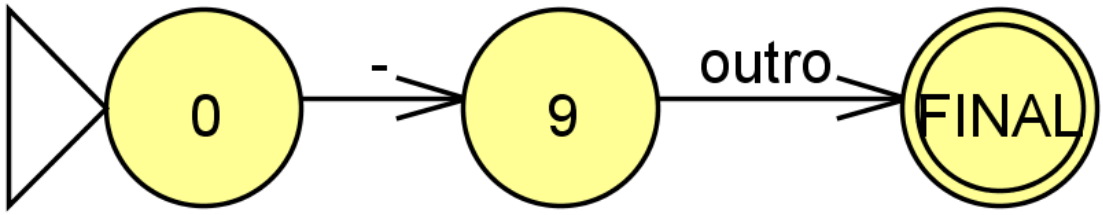


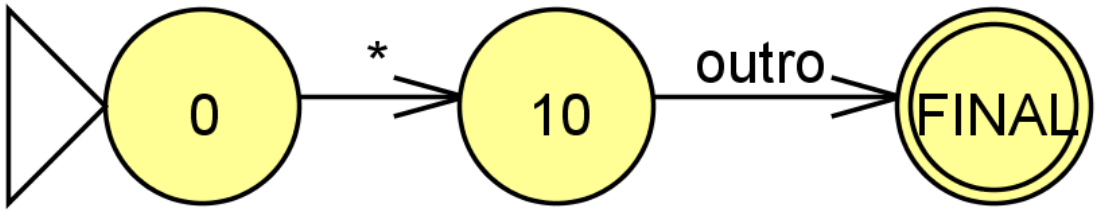

outro
0
>
5
=
16
outro
FINAL

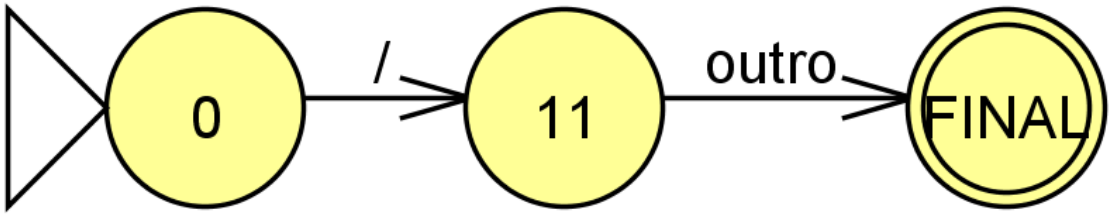

18
>
outro
0
<
6
outro
FINAL
=
outro
19

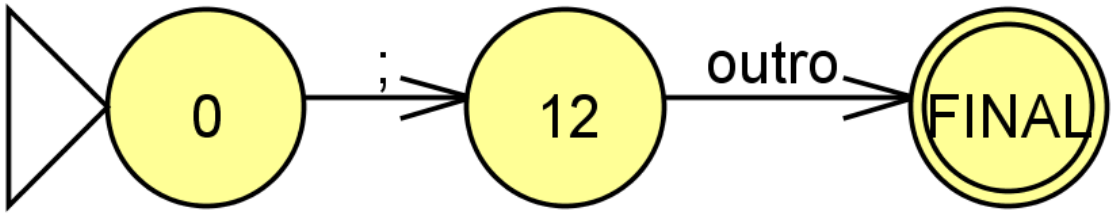

outro
0
=
7
=
20
outro
FINAL

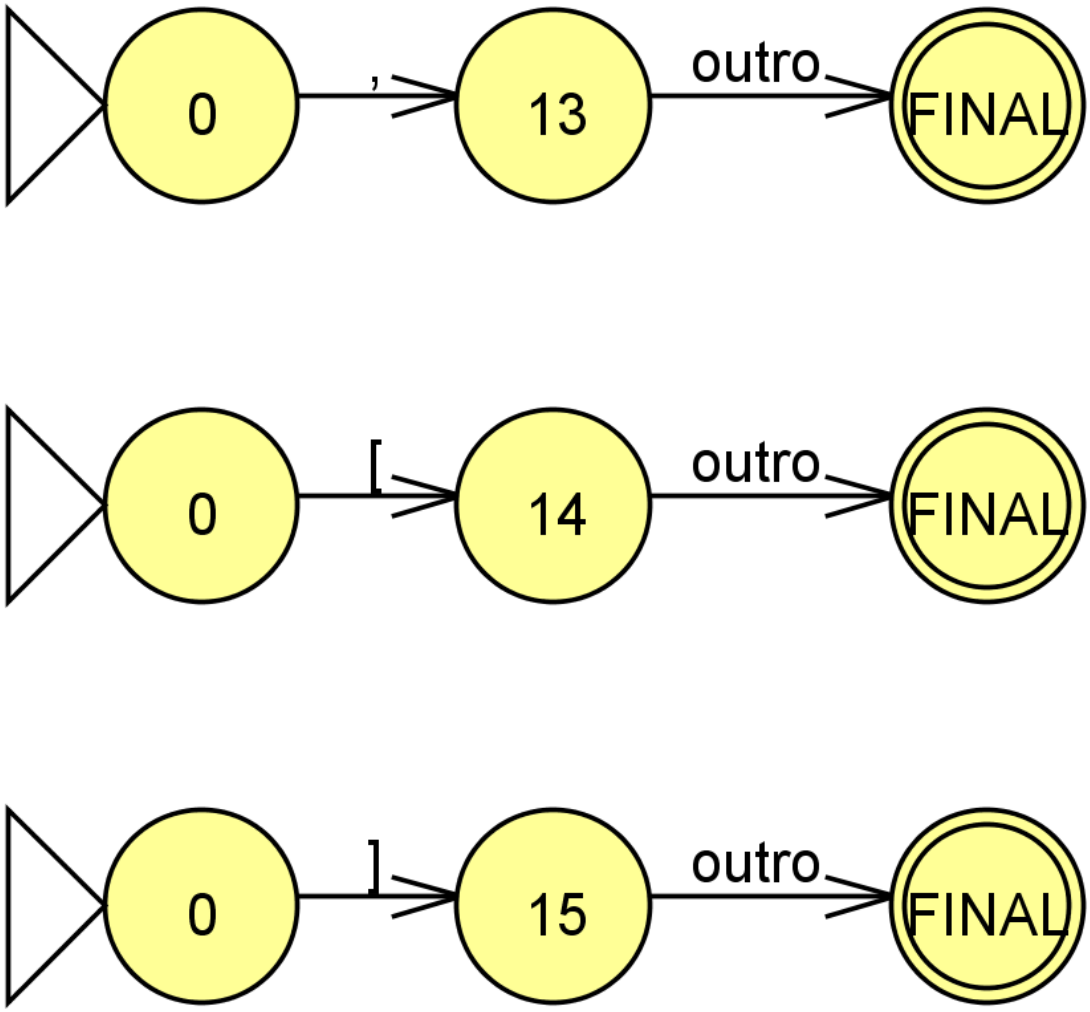

0
+
8
outro
FINAL


0
-
9
outro
FINAL

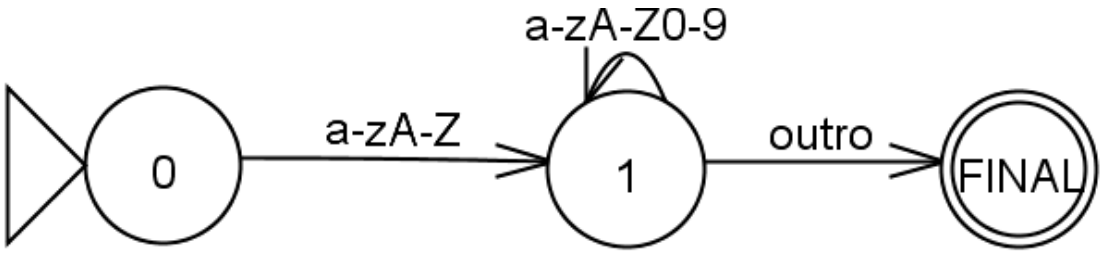

0
*
10
outro
FINAL

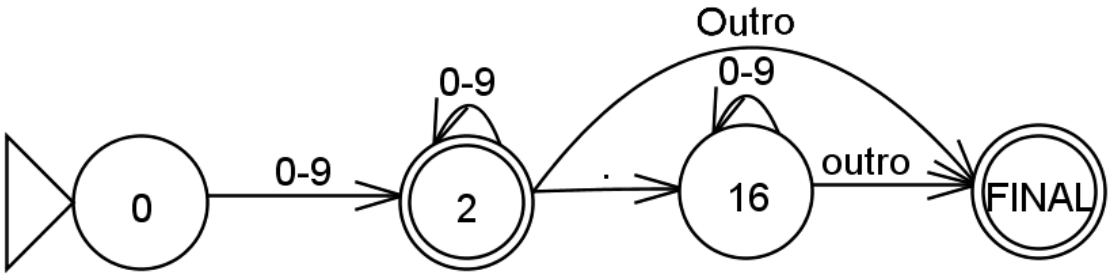

0
/
11
outro
FINAL

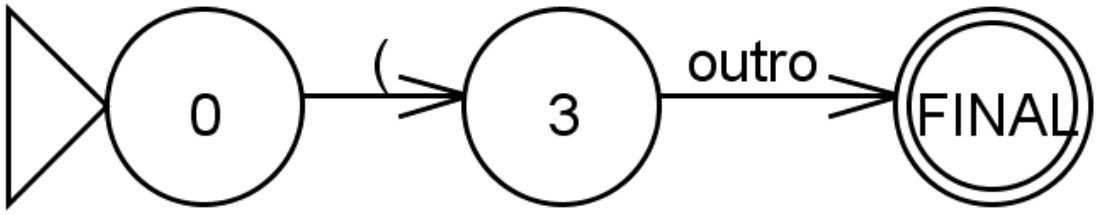

0
;
12
outro
FINAL

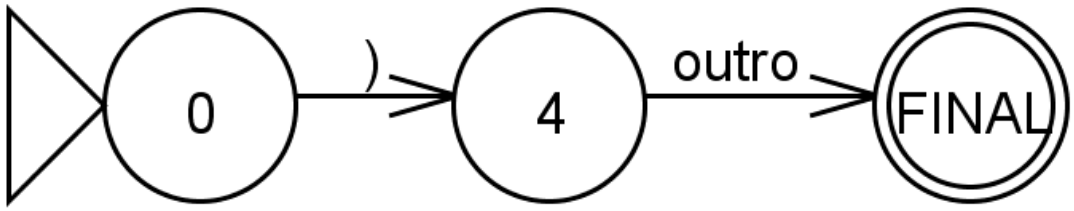

0
,
13
outro
FINAL
0
[
14
outro
FINAL
0
]
15
outro
FINAL

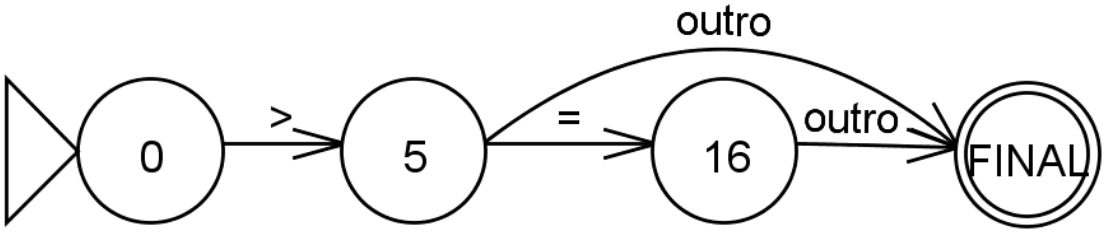
APÊNDICE C – AUTÔMATOS DESABILITADOS

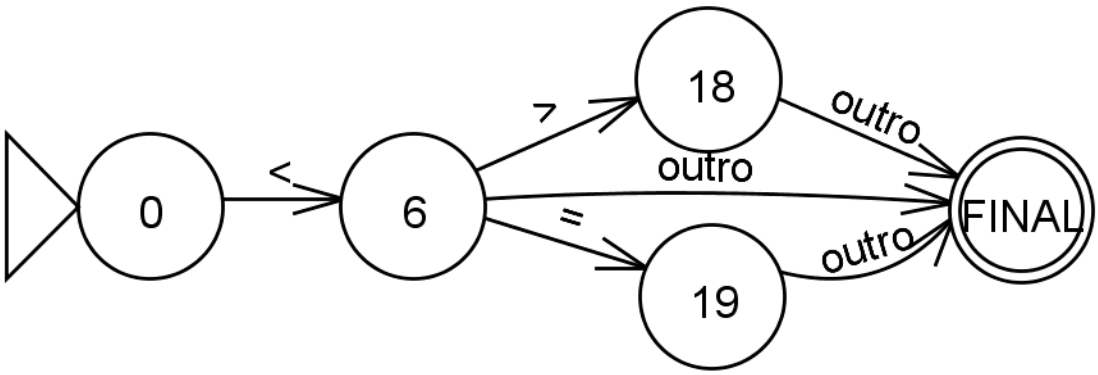


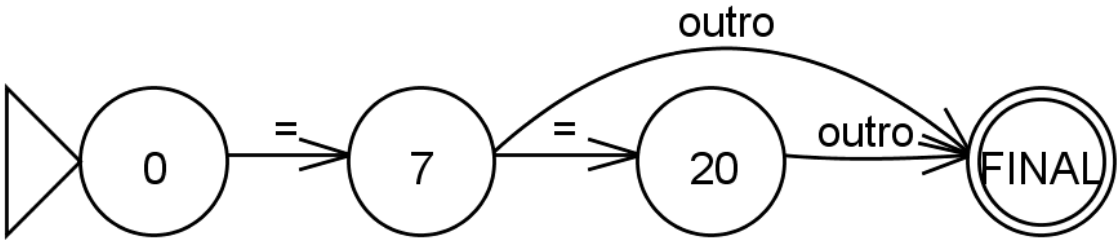


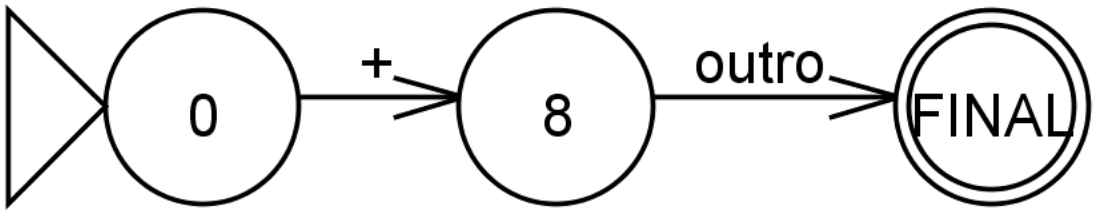


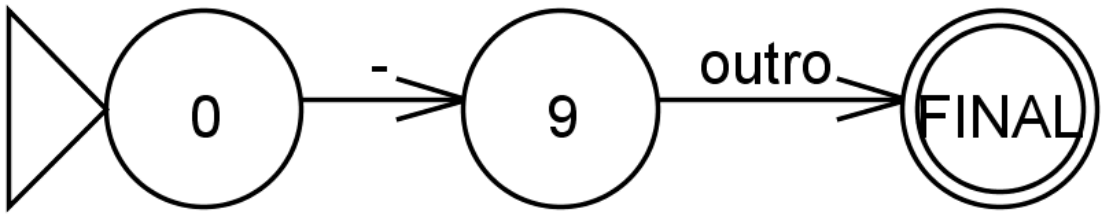


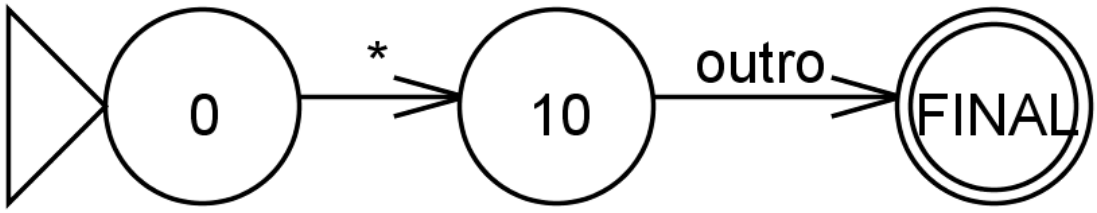

outro
0
>
5
=
16
outro
FINAL

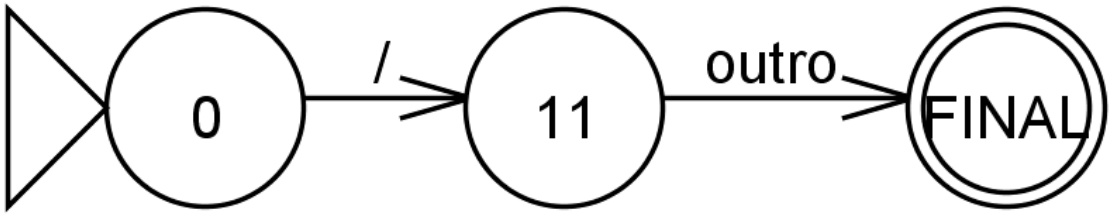

18
>
outro
0
<
6
outro
FINAL
=
outro
19

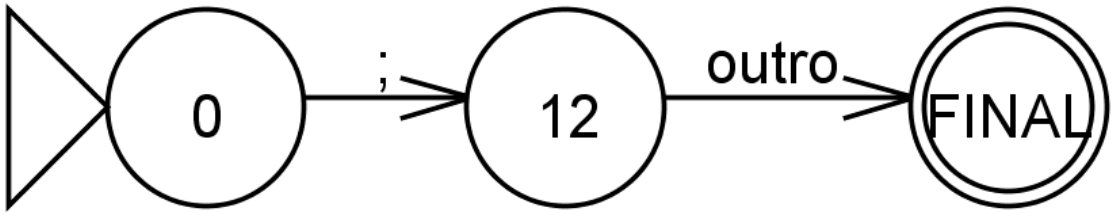

outro
0
=
7
=
20
outro
FINAL

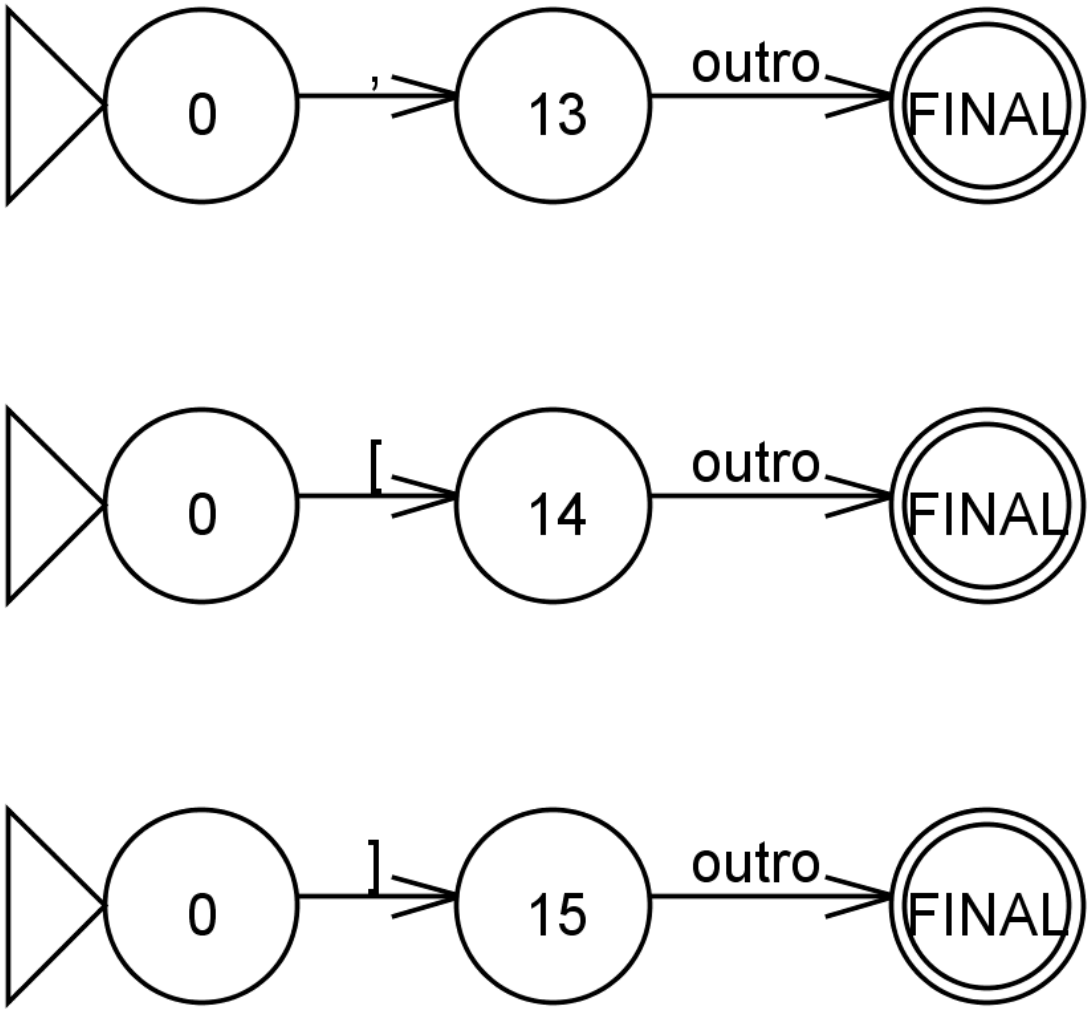

0
+
8
outro
FINAL


0
-
9
outro
FINAL


0
*
10
outro
FINAL


0
/
11
outro
FINAL


0
;
12
outro
FINAL


0
,
13
outro
FINAL
0
[
14
outro
FINAL
0
]
15
outro
FINAL