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).

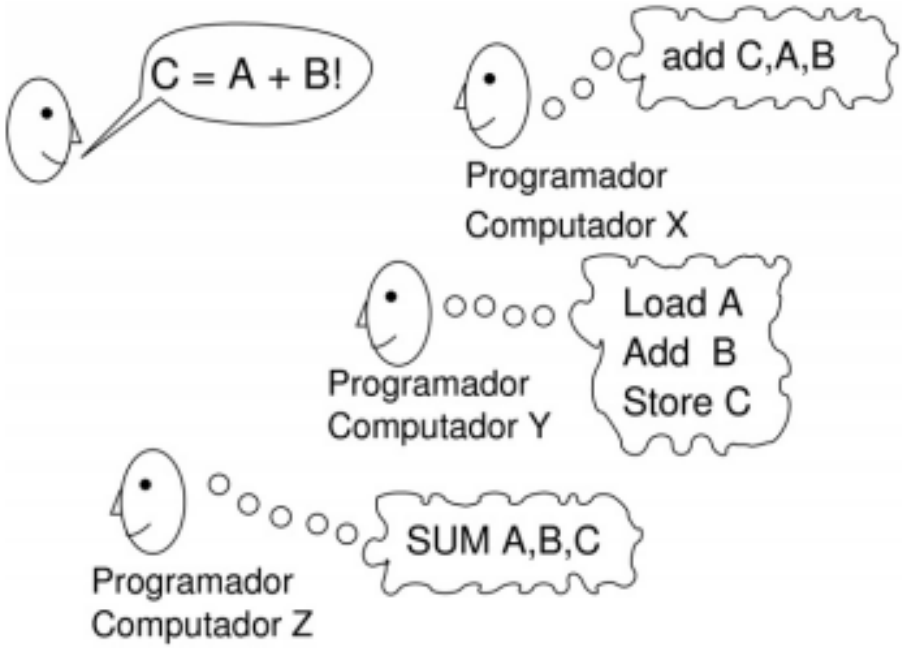
Para solucionar este problema, em busca de uma solução automática, procurando traduzir as especificações genéricas (RICARTE, 2008). Com este objetivo desenvolveu os compiladores.

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).

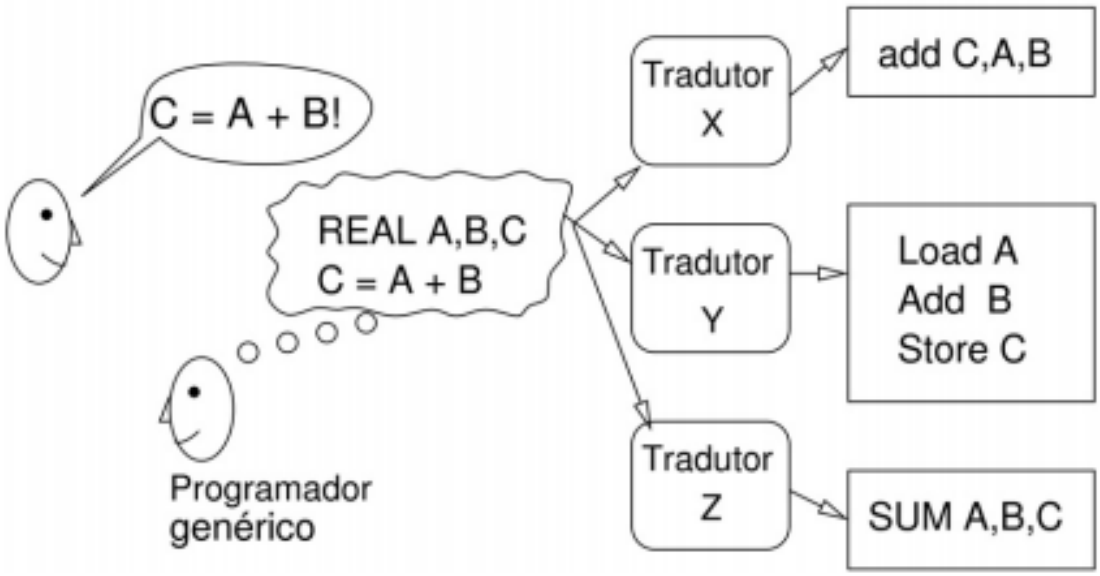
O compilador segue passos para a tradução correta do código fonte para o código de saída. Os passos de análises são: análise léxica (AL), análise sintática (AS) e análise semântica (ASE). Os passos de tradução, são a Geração de código intermediário, a otimização de código, a geração de código final e 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.

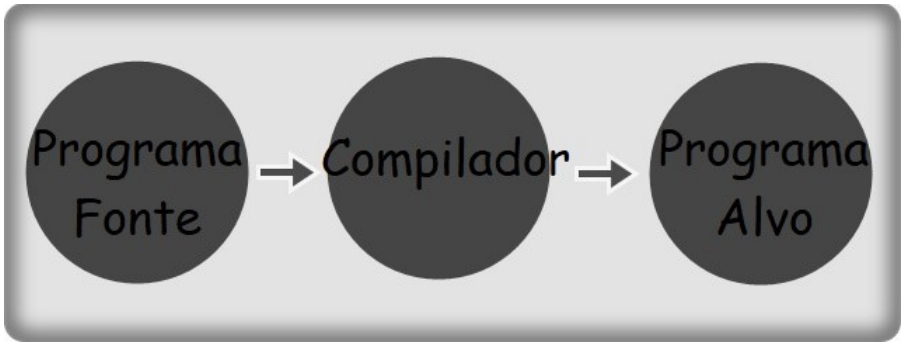
Na figura 4 é ilustrado um fluxo geral de como é convertido o código, comparando o compilador ao interpretador. O que difere entre ambos é em qual momento ocorre a inserção de dados, tempo de execução e a diferença na saída de arquivos após os processos, o compilador gera um arquivo com a linguagem alvo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

FIGURA 2 – A programação com compiladores.



FONTE: (RICARTE, 2008).

FIGURA 3 – Exemplo do Funcionamento de um Compilador.



FONTE: O próprio autor.

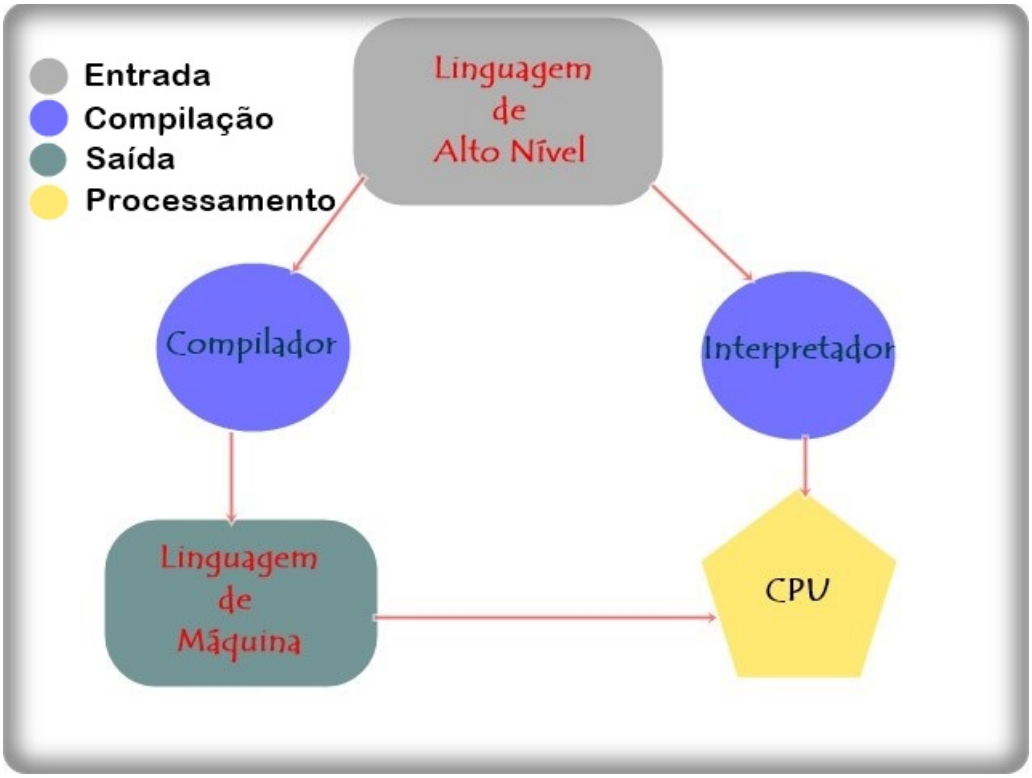
de máquina), e o interpretador executa diretamente a instrução, sem criar um arquivo fonte.

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

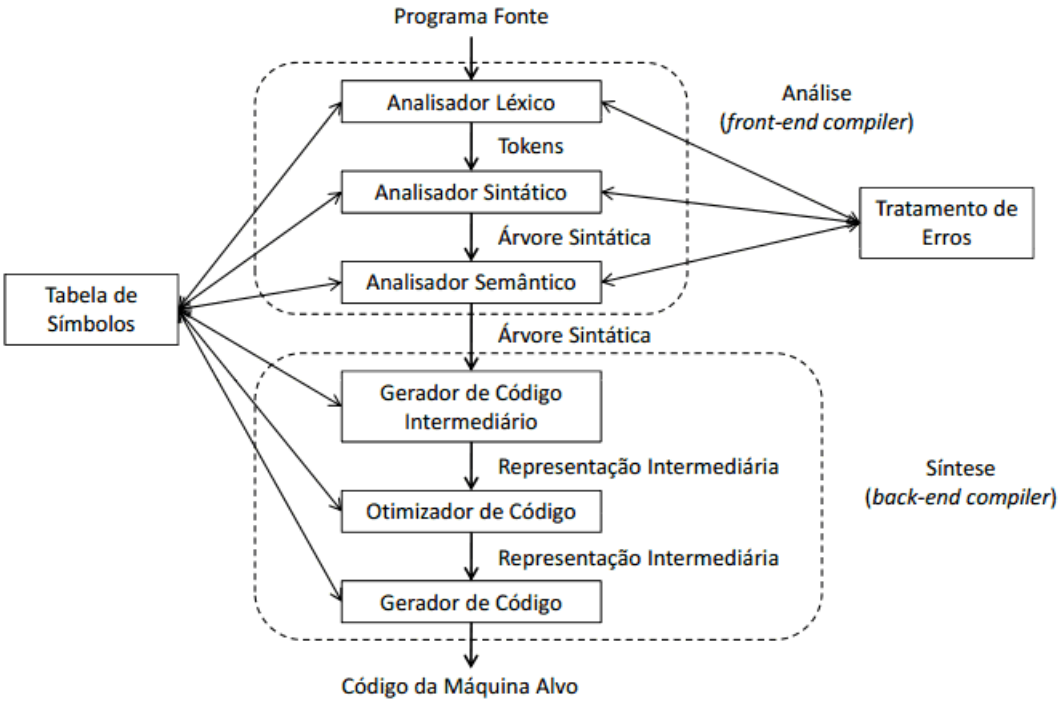
b) Atribuição =

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 (*/).

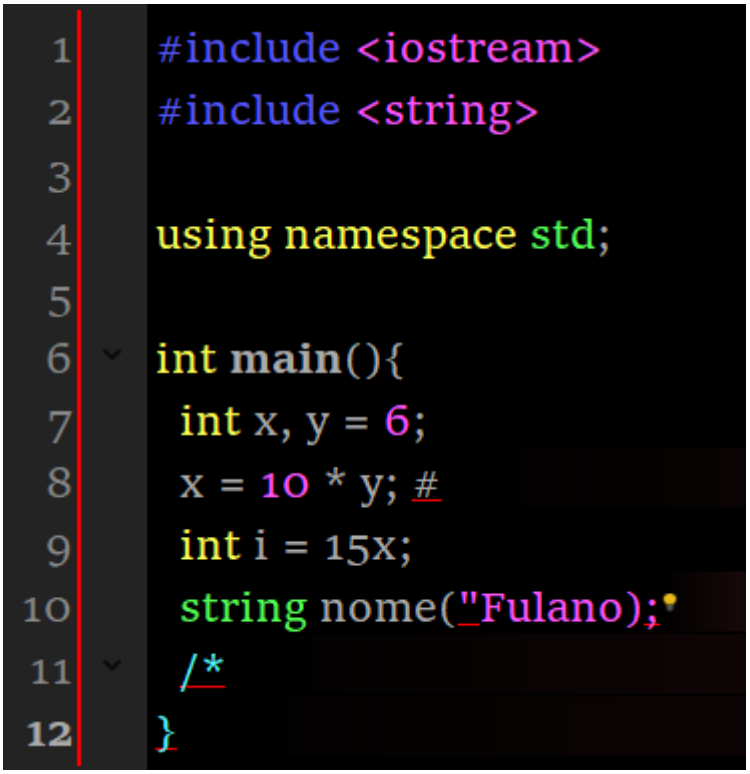
Na figura 5 ocorrem erros de análise sintática, que é o passo que abordaremos em seguida.

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.

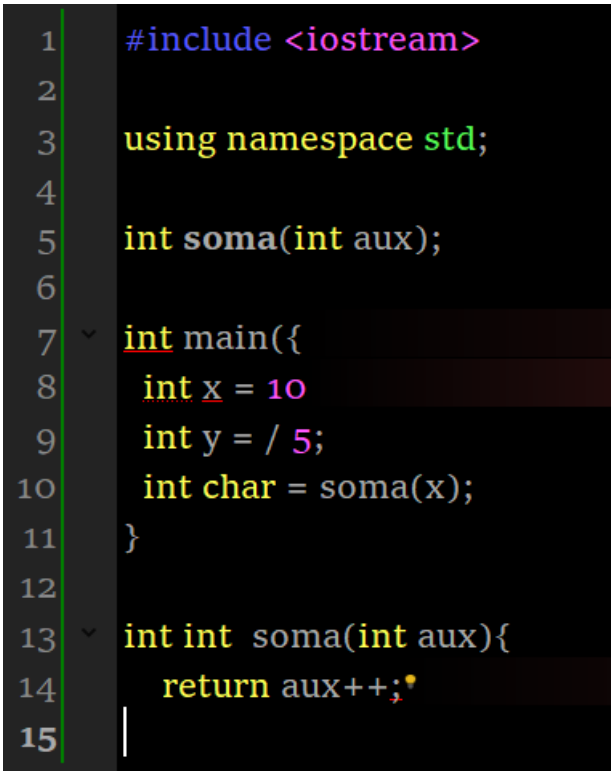
c) Na linha 9, o operador de divisão /, não consegue montar uma expressão de divisão por faltar o operador da esquerda.

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 };
```

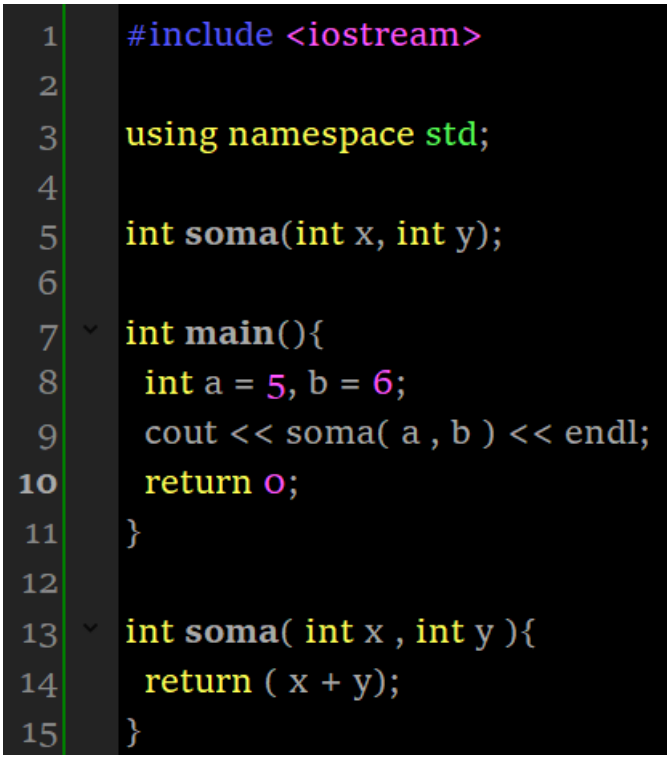
FONTE: O próprio autor.

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).

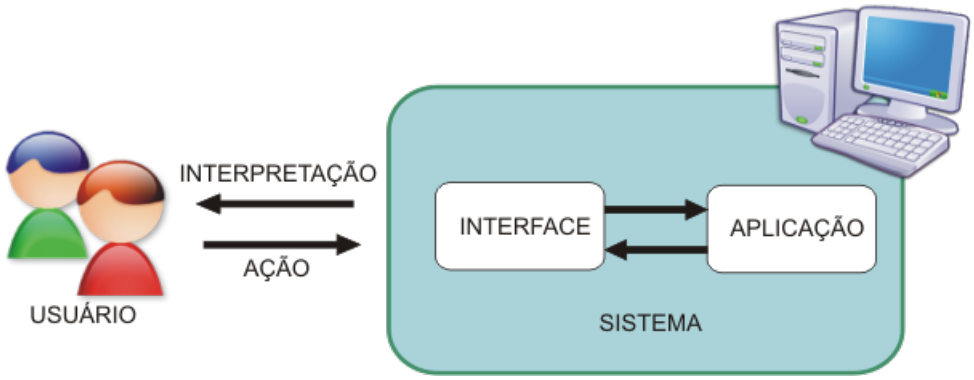
3 INTERAÇÃO HUMANO-COMPUTADOR(IHC)

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

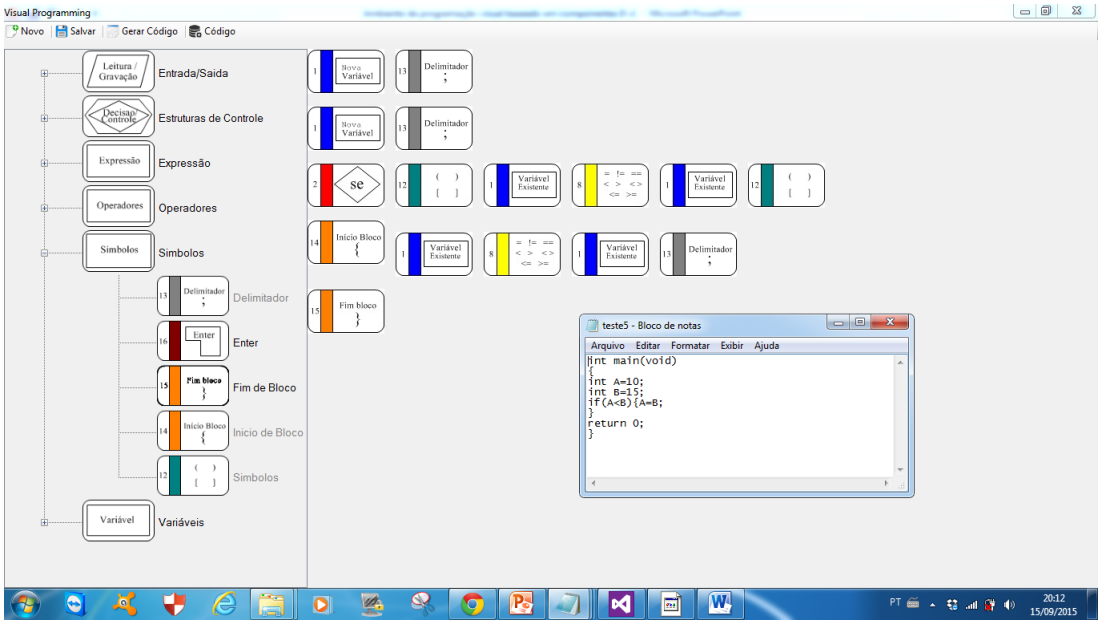
(BURIGO, 2015) apresenta um trabalho sobre a criação de um ambiente de desenvolvimento utilizando componentes vindos de fluxogramas, e transformando-os em código da linguagem C, para que auxilie pessoas que estão iniciando na área de programação, e também pessoas que já tem familiaridade na área, mas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

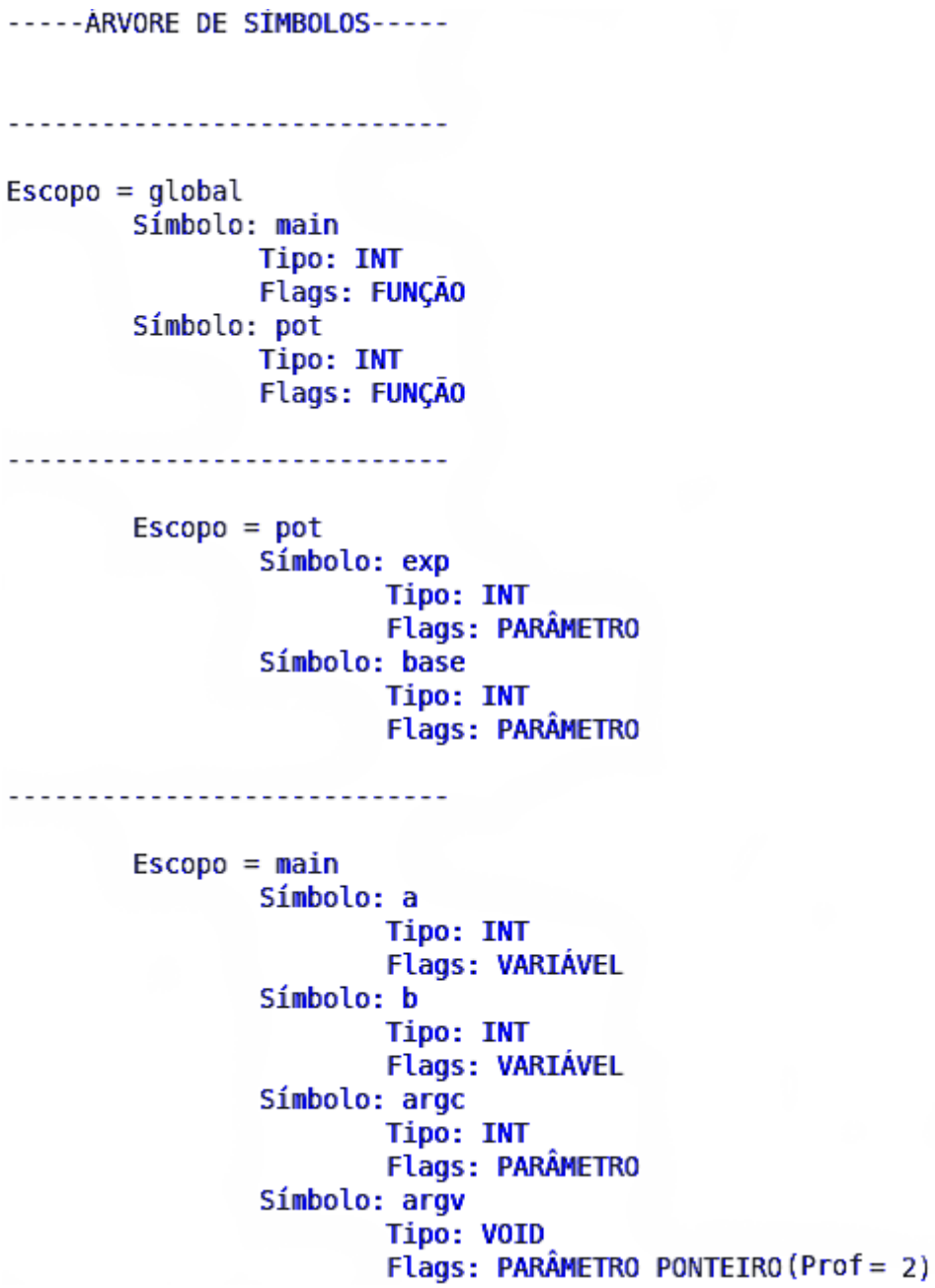
(JAIN NIDHI SEHRAWAT, 2014) apresenta um trabalho com sistema de compilação adaptativa, com o objetivo de fornecer uma documentação sobre o projeto e desenvolvimento do compilador, para auxiliar na compreensão do tema e criar técnicas eficazes para desenvolver.

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).

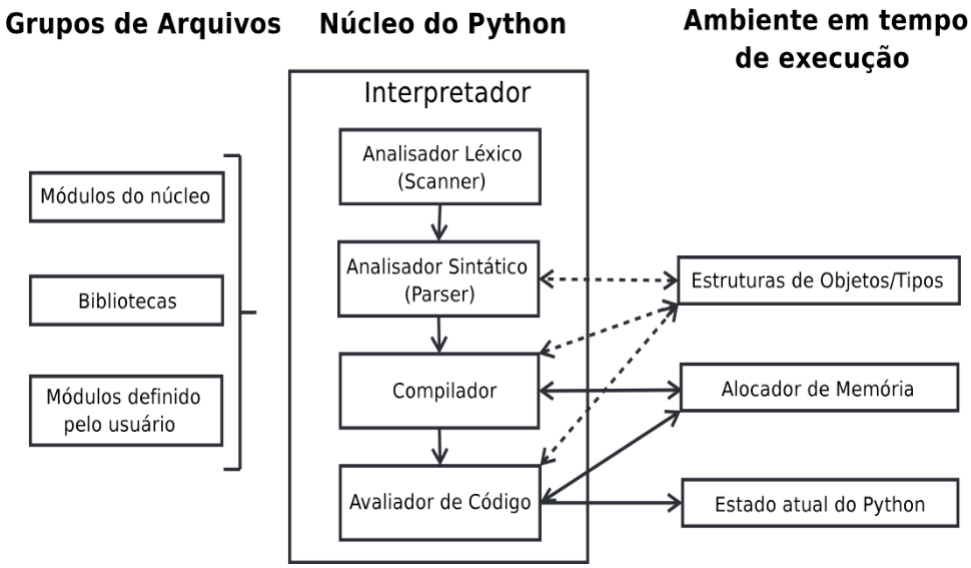
Este artigo realiza uma abordagem sobre a estrutura da arquitetura, do interpretador, a gramática da linguagem e as características.

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 capítulo iremos informar 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 individuos participantes.

As tecnologias utilizados para o desenvolvimento do interpretador, são : linugagem de programação C++, QT creator, Jflap e Análise Léxica.

5.1 LINGUAGEM C++

A linguagem utilizada para o desenvolvimento deste software, é C++, está linugagem foi escolhida por conta de ser a linguagem base no ensino da faculdade, e por ser a linguagem utilizada na ferramenta QT creator.

5.2 QT CREATOR

QT Creator um ambiente de desenvolvimento integrado (IDE), permitindo que desenvolva sistemas para plataformas multiplas, possibilitando tambem criar o design. Por ter essa versatilidade com o compilador integrado, dando a a opção de desenvolver a logica do interpretador e criar a interface, esta ferramenta foi utilizada no desenvolvimento.

5.3 JFLAP

JFLAP é um software gratuiuto que possibilidata a criação de automatos finitos não deterministicos, maquinas de turing e varios tipos de gramatica. Este software foi utilizado para criar todos os automatos da linugagem D+, utilizados para ilustrar na interface do interpretador por onde o código passou.

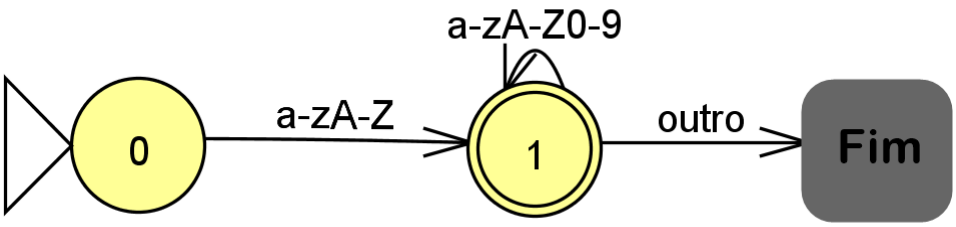
Na figura 16 é ilustrado um automato criado no JFLAP, ele é apresentado na interface desenvolvida quando aquele caminho de automato é acessado pela analise lexica.

Na figura 17 é ilustrado um automato que não foi acessado na análise léxica.

5.4 ESTRUTURAS ADOTADAS NO DESENVOLVIMENTO

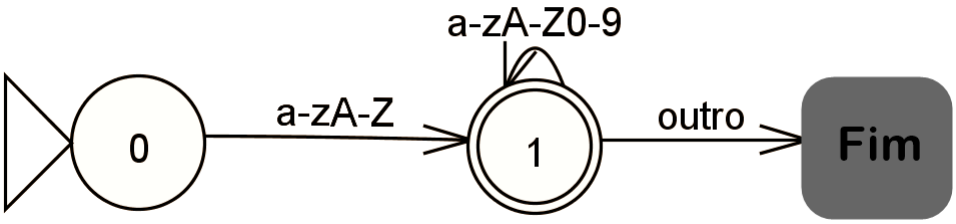
Nesta sesão, serão tratados as estruturas utilizadas para o desenvolvimento da logica da compilação, Analise léxica, Analise Sintatica, fila.

FIGURA 16 – Automato do estado 0 ao 1 ativado.



FONTE: O próprio autor.

FIGURA 17 – Automato do estao 0 ao 1 desativado.



FONTE: O próprio autor.

5.4.1 Analise lexica

A analise lexica foi completamente desenvolvida manualmente, utilizando os automatos criados no JFLAG, e as regras da linugagem D+. Os estados dos automatos foram representados por uma variavel declarada como state, e manipulada atravez de switch cases. Na figura 18 é ilustrado um trecho do codigo, usando esta variavel para comparar e atribuir um novo estado.

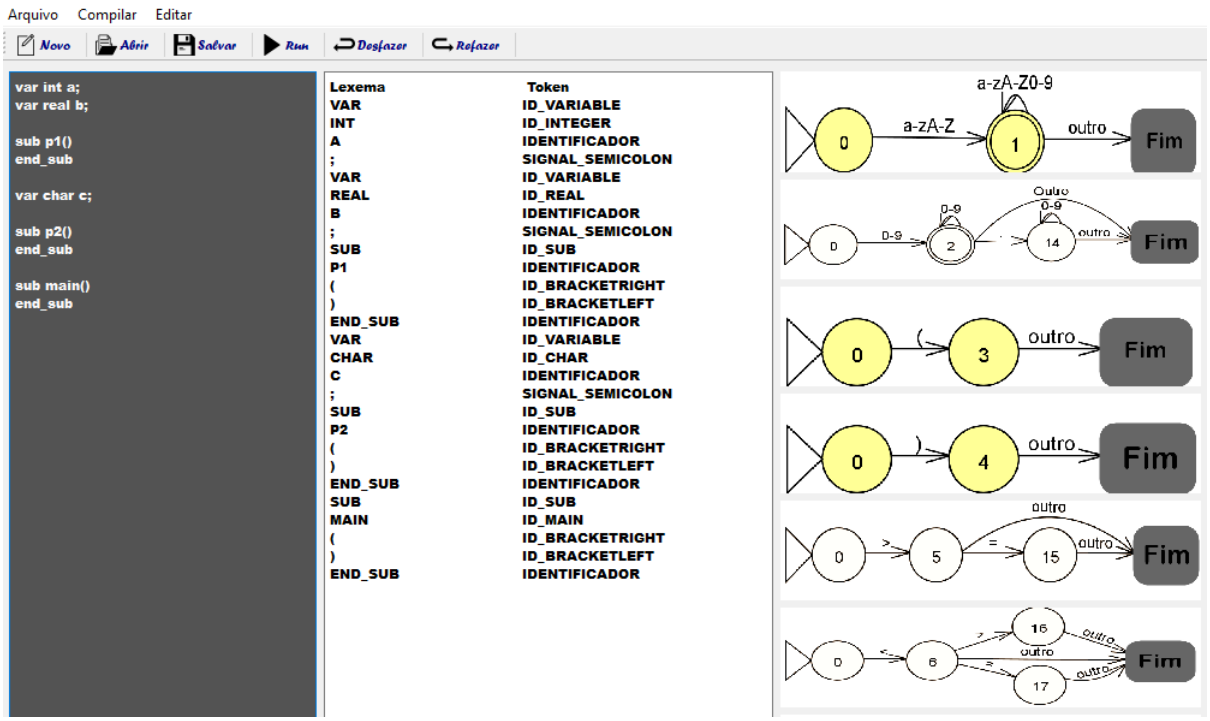
Na figura 19 é apresentada a interface operando em conjunto com a análise lexica, com o editor de texto a esquerda, a saída da análise léxica, com a geração de tokens e lexemas no centro e os automatos ativos por onde o processo passou, e desativados por onde não passou no lado direito.

FIGURA 18 – Código switch case dos estados.

```
265     switch(state) {
266       case 0:
267         stateoo();
268         break;
269       case 1:
270         stateo1();
271         break;
272       case 2:
273         stateo2();
274         break;
275       case 3:
276         stateo3();
277         break;
278       case 4:
279         stateo4();
280         break;
```

FONTE: O próprio autor.

FIGURA 19 – Interface integrada com a Análise Léxica.



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.

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.

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.