UNIVERSIDADE TUIUTI DO PARANÁ

JÔNATAS ALEXANDRE HILLE

COMPILADOR PARA A LINGUAGEM ORIENTADA A OBJETOS CHIMERA

UNIVERSIDADE TUIUTI DO PARANÁ

JÔNATAS ALEXANDRE HILLE

CURITIBA

2017

COMPILADOR PARA A LINGUAGEM ORIENTADA A OBJETOS CHIMERA

Trabalho de conclusão de curso apresentado ao Curso de Bacharelado em Ciências da Computação da Universidade Tuiuti do Paraná.

Professor Diógenes Cogo Furlan

CURITIBA

2017

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

Uma linguagem de programação é uma notação para escrever programas, que são especificações de um algoritmo para um computador. É comum uma linguagem de programação conter níveis de abstração para definir e manipular estruturas de dados ou para controlar um fluxo de execução. Essas linguagens são conhecidas como linguagens de alto nível.

Existem diversos paradigmas de linguagens de programação. Os dois mais utilizados são o paradigma imperativo (ou procedural) e o paradigma orientado a objetos, sendo que este último tem superado o paradigma imperativo em importância, por questões de maior adaptação em sistemas criados pelo mercado de software. Uma das limitações das linguagens imperativas é a não existência de uma forma simples para criar uma conexão forte entre dados e funcionalidades, o que é atendido por linguagens de programação orientadas a objetos. É importante destacar que as principais linguagens de programação orientada a objetos, como C#, Visual Basic, C++ e Java foram projetadas principalmente para apoiar o paradigma imperativo.

Só é possível executar um programa se a linguagem de programação abstrata, em forma de código fonte, for traduzida para uma linguagem de baixo nível. Esta tradução pode ser realizada por um compilador ou um interpretador.

As principais fases de um compilador são: (i) Leitura do código fonte, (ii) Análise Léxica, (iii) Análise Sintática, (iv) Análise Semântica, (v) Geração do código intermediário, (vi) Otimização de Código e (vii) Geração do Código Objeto.

A MEPA (Máquina de Execução Para Pascal) é uma máquina de pilha para desenvolvimento de código intermediário para Linguagens imperativas. Ela é uma máquina virtual que executa programas escritos em Pascal Simplificado, que tem como objetivo apresentar as instruções *Assembly* que causem o efeito esperado.

Neste trabalho, implanta-se a linguagem de programação Chimera por execução via compilador.

Seu objetivo principal é construir um compilador para uma linguagem de programação orientada a objetos a partir de uma linguagem estruturada, utilizando a MEPA como linguagem intermediária.

Para alcançar este objetivo foi necessário criar a gramática desta linguagem de programação e gerar o código intermediário, convertendo os conceitos de uma linguagem de programação orientada a objetos para a MEPA, que é uma linguagem estruturada.

Com o uso da MEPA queremos mostrar mais uma vez que os conceitos de programação orientada a objetos (POO) podem ser mapeados em conceitos de programação estruturada.

Devida a complexidade do assunto, o foco do trabalho é implementar os conceitos iniciais de POO, que são: (i) objetos, (ii) classes, (iii) métodos e atributos, (iv) encapsulamento, (v) herança e (vi) polimorfismo por inclusão.

Serão deixados para trabalhos futuros as implementações de herança múltipla, polimorfismo de sobrecarga, polimorfismo de coerção, polimorfismo paramétrico, interface e outros conceitos específicos ou avançados de POO.

O trabalho está dividido em seis capítulos, os quais são:

- Capítulo 2 – Trabalhos relacionados: Irá apresentar os trabalhos relacionados utilizados neste estudo.
- Capítulo 3 – Fundamentação Teórica: Neste capítulo é tratado os conceitos de um compilador, iniciando pela sua estrutura, depois explicando as análises léxica, sintática e semântica, encerrando com a geração de código intermediário, ambientes de execução e otimização de código.
- Capítulo 4 – Conceitos de Programação Orientada a Objetos: Capítulo dedicado a apresentar os principais conceitos de programação orientada a objetos, como classes, objetos, encapsulamento e herança.
- Capítulo 5 – Metodologia: Os detalhes das ferramentas e métodos utilizados no desenvolvimento deste trabalho são demonstrados neste capítulo.
- Capítulo 6 – Análise dos Resultados: Este capítulo detalha as técnicas e rotinas de testes aplicados, bem como os resultados obtidos e uma análise dos mesmos.
- Capítulo 7 – Conclusão: O último capítulo aborda as conclusões obtidas após a realização deste projeto e as sugestões para trabalhos futuros.

2 TRABALHOS RELACIONADOS

Este capítulo trata dos trabalhos relacionados ao projeto em construção neste trabalho. A grande maioria dos trabalhos científicos encontrados tem como objetivo melhorar a forma como é apresentada a disciplina de compiladores a novos estudantes, trazendo novas metodologias e técnicas para acelerar o aprendizado e melhorar a compreensão do assunto.

Na seção 2.1 apresenta-se um trabalho sobre as informações básicas sobre compiladores.

Na seção 2.2 é mostrado um trabalho sobre um projeto que ensina novos alunos a aprenderem orientação a objetos através da construção de um compilador.

Na seção 2.3 o trabalho apresentado demonstra a eficiência de se utilizar algumas ferramentas auxiliares para o ensino da matéria de compiladores.

A seção 2.4 apresenta uma monografia que tem como objetivo descrever uma ferramenta que possibilita aos alunos aprenderem lógica de programação, compiladores e também arquitetura de computadores.

E no último trabalho, na seção 2.5, os autores propõem uma nova abordagem para o analisador léxico, com o objetivo de reduzir o custo computacional desta primeira etapa de um compilador.

2.1 COMPILER BASIC DESIGN AND CONSTRUCTION (JAIN, 2014)

O objetivo deste artigo é providenciar informações básicas sobre compiladores e sua implementação. Os autores comentam que um programa escrito precisa ser interpretado e que se enquadre em uma sequência de instruções antes de ser processado por um computador. Sendo assim, o compilador é uma interface entre o programador e uma máquina.

O termo compilação simboliza a conversão de um algoritmo expressado em uma linguagem de origem *human-oriented* para um algoritmo expressado em uma linguagem alvo *hardware-oriented.*

O autor apresenta superficialmente a análise léxica e sintática, explicando o objetivo de cada uma e como elas podem trabalhar juntas para se alcançar o objetivo de se implementar um compilador.

2.2 COMPILER CONSTRUCTION AS AN EFFECTIVE APPLICATION TO TEACH OBJECT-ORIENTED PROGRAMMING (DEMAILLE, 2008)

Neste artigo foram reportados dez anos de experiência em como a construção de um compilador pode ser de grande aproveitamento para o ensino da programação orientada a objetos (POO) para os alunos de ciência da computação de uma universidade particular francesa, a EPITA (*Ecole Pour l'Informatique et les Techniques Avancées*).

O projeto da EPITA teve como objetivo principal ensinar a POO para os alunos de uma forma eficiente e também de uma forma que os alunos pudessem entender como uma linguagem de programação funciona.

Durante três (opcionalmente cinco) meses, enquanto C++ é ensinado, estudantes de graduação implementam os estágios de um compilador em C++ para uma linguagem orientada a objetos: Tiger. As aulas de compiladores são integradas com os projetos do curso. Em muitos casos, como os estudantes não irão desenvolver um compilador, construir um por completo não é o objetivo principal. O objetivo do ensino é no “como” e não no “o quê”.

O artigo apresenta quatro características dos compiladores que as tornam excelentes na demonstração da POO: (i) Dados e Algoritmos, (ii) Arquitetura e Padrão de Desenho, (iii) Linguagem de origem orientada a objetos e (iv) Linguagem de implementação orientada a objetos. Ele também explica como é a integração das aulas de compiladores com o projeto, bem como o *feedback* dos alunos que participaram do projeto.

2.3 COMPILER CONSTRUCTION (SINGH, 2013)

Este trabalho apresenta uma introdução detalhada aos compiladores, incluindo também um pouco da parte histórica. Porém, este não é o seu objetivo principal.

O foco do artigo é apresentar algumas ferramentas que ajudam no processo de aprendizado de estudantes de graduação que estejam cursando a matéria de compiladores. Essas ferramentas tornam o aprendizado mais objetivo para que a implementação de um compilador seja possível dentro do período de tempo estipulado para o curso.

O artigo detalha como utilizar estas ferramentas e conclui que elas aceleraram o processo de implementação de um compilador dentro do período de tempo destinado ao curso de compiladores na grande maioria das universidades.

2.4 UM COMPILADOR, UMA LINGUAGEM DE PROGRAMAÇÃO E UMA MÁQUINA VIRTUAL SIMPLES PARA O ENSINO DE UMA LÓGICA DE PROGRAMAÇÃO, COMPILADORES E ARQUITETURA DE COMPUTADORES (OLIVEIRA, 2014)

Esta monografia apresenta uma ferramenta que oferece um ambiente integrado de desenvolvimento que possibilita o aluno a desenvolver, depurar e executar programas na linguagem Simples.

O ambiente inclui um simulador para a Máquina Virtual Simples (MVS), que foi inspirada na MEPA (Máquina de Execução de Pascal). Assim como a MEPA, a MVS é uma máquina de pilha, característica muito apropriada para a tradução encontradas em linguagens de programação como o C ou Pascal.

Após a execução de cada uma das instruções do código-objeto carregado o simulador exibe, através de uma forma gráfica, o estado da memória interna e dos registradores da máquina virtual. Com a ferramenta também é possível realizar a execução passo a passo do código simples mostrando, a cada linha, qual é o código-objeto equivalente.

Hoje a ferramenta criada pelo autor do trabalho é utilizada para auxiliar na disciplina de compiladores na Universidade Federal de Alfenas, em Minas Gerais.

2.5 A NEW APPROACH OF COMPILER DESIGN IN CONTEXT OF LEXICAL ANALYZER AND PARSER GENERATION FOR NEXTGEN LANGUAGES (BHOWMIK, 2010)

Um dos primeiros passos do compilador é a análise léxica. Neste passo, os *tokens* são identificados e categorizados. Neste artigo, os autores propõem uma nova abordagem para a análise de complexidade do analisador léxico. O modelo apresenta diferentes etapas da geração dos *tokens* através de lexemas, além de uma melhor implementação do sistema de entrada.

O modelo também introduz a análise sintática e uma nova forma de implementar a tabela de símbolos utilizando uma pilha. Mas o objetivo principal é reduzir o custo computacional no momento da análise léxica.

Foi concluído que o novo modelo reduz o custo computacional e ainda apresenta uma nova maneira de implementar um analisador léxico.

2.6 COMPARAÇÃO

Na tabela 1 podemos ver a comparação entre os assuntos que os trabalhos relacionados abordam e os assuntos que este trabalho irá abordar.

TABELA 1 – TABELA COMPARATIVA

Trabalhos	**Análise Léxica**	**Análise Sintática**	**Análise Semântica**	**Geração de Código**	**Compilador**	**POO**
JAIN, 2014	X	X			X	
DEMAILLE, 2008					X	X
SINGH, 2013	X	X			X	
OLIVEIRA, 2014	X	X	X	X	X	
BHOWMIK, 2010	X	X			X	

FONTE: o próprio autor

3 COMPILADORES

Este capítulo tem como objetivo apresentar teoricamente a estrutura de um compilador. A primeira seção faz um breve resumo sobre essa estrutura, logo depois, são discutidas todas fases do compilador, iniciando pelas análises léxica, sintática e semântica. Após a fase de análises, são apresentadas três seções referentes a geração de código, começando pela geração de código intermediário, seguido pela geração do código-objeto e os ambientes de execução e, finalizando o capítulo, a otimização de código é abordada.

3.1 ESTRUTURA DE UM COMPILADOR

Compilador é um programa de computador que traduz um programa escrito em uma linguagem de alto nível para uma linguagem de máquina (baixo nível). Apesar da simples notação, um compilador é um sistema complexo, com diversos componentes internos, algoritmos e interações complexas.

Um compilador pode consistir em diversas fases, mas qualquer texto que se refira a construção de um compilador indica que existem pelo menos quatro fases em um processo de compilação. São elas a análise léxica, a análise sintática, a análise semântica e a geração de código, como apresentado na Figura 1.

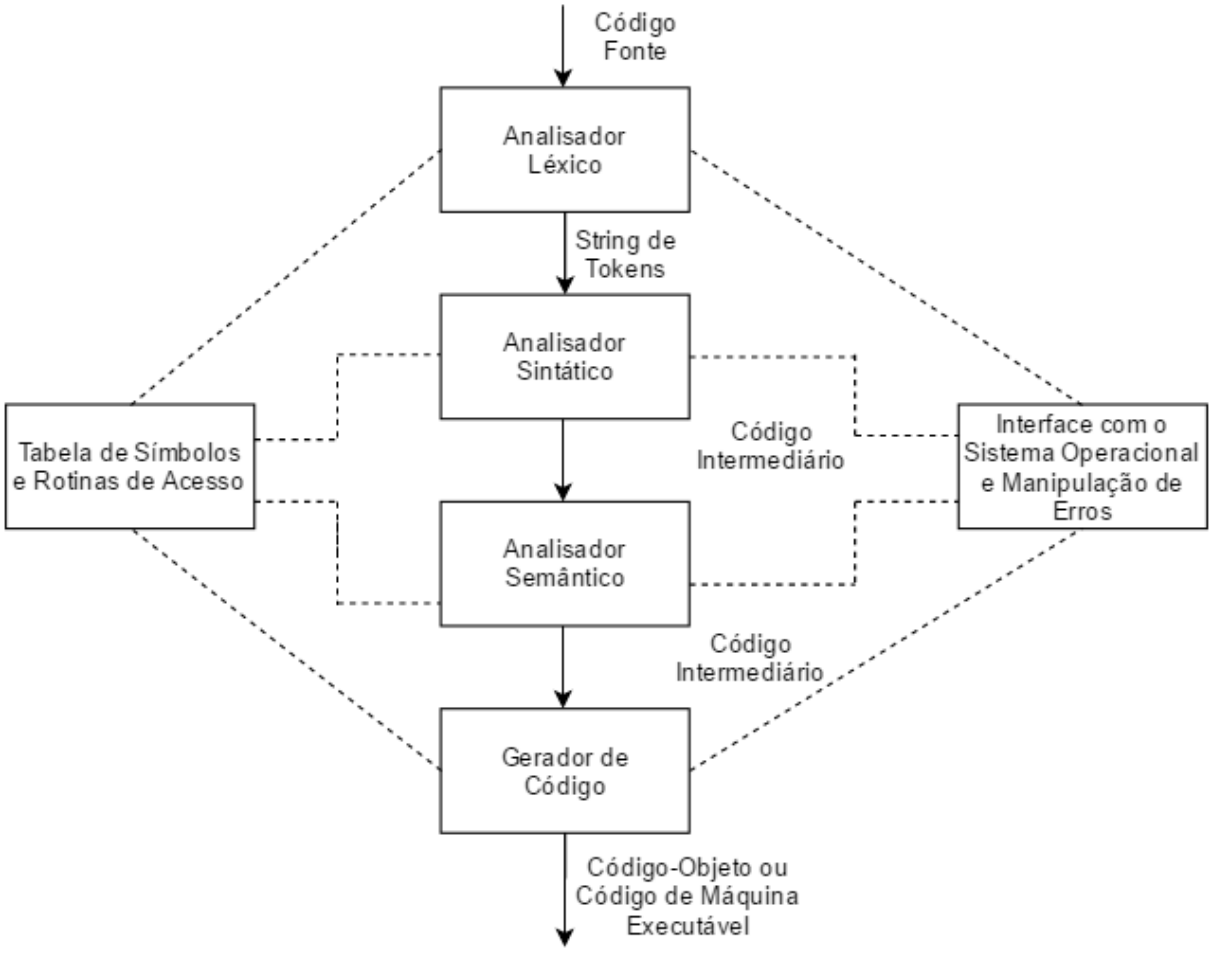
O analisador léxico analisa uma *string* de caracteres e a divide em *tokens* que sejam reconhecidos pelo vocabulário da linguagem e que podem gerar erros caso uma dessas *strings* não seja reconhecida pela linguagem. (MUCHNICK, 1997)

O analisador sintático processa a sequência de *tokens* gerados pela análise léxica e que produz uma representação intermediária, como árvores de análise (*parse trees*), e uma tabela de símbolo que registra os identificadores registrados no programa e alguns de seus atributos. Nesta fase também é possível gerar erros caso uma sequência de *tokens* tenha algum problema de sintaxe.

Na fase do analisador semântico, é verificado se o programa satisfaz as propriedades semânticas requeridas pela linguagem, como por exemplo verificar a compatibilidade dos tipos e o escopo dos identificadores. Caso esses requerimentos não sejam atendidos é possível alertar o usuário com mensagens de erro.

E por fim, a geração de código transforma o código intermediário equivalente a um código de máquina, seja em forma de código-objeto ou em código de máquina executável.

FIGURA 1 – ESTRUTURA DE UM COMPILADOR



FONTE: MUCHNICK, 1997, p.2.

Adicionalmente às quatro fases citadas acima, um compilador pode incluir outros processos como manipulação de erros e alertas, uma tabela de símbolos e suas rotinas de acesso e também uma interface com o sistema operacional e o usuário, na qual é possível ler e escrever arquivos, ler entradas do usuário, exibir mensagens, etc. Como ilustrado na Figura 1, todos estes processos podem fazer parte de todas as fases do compilador. (MUCHNICK, 1997)

Outro processo importante para um compilador que tenha como objetivo produzir um código-objeto com eficiência é a otimização de código. A otimização de código pode

ser realizada em dois momentos: ao final da geração de código intermediário e entre a geração de código final e o código-objeto.

3.2 ANÁLISE LÉXICA

A palavra léxica, no sentido tradicional do termo, significa “relacionado a palavras”. Em termos de linguagem de programação, palavras são objetos como nomes de variáveis, números, palavras-chave, símbolos especiais, etc. Essas entidades de tipos de palavras são tradicionalmente chamadas *tokens*. Um *token* representa um determinado padrão de caracteres reconhecido pela varredura da análise léxica a partir do início do código fonte e vão se formando à medida que esses caracteres são fornecidos. (MOGENSEN, 2011; LOUDEN,2004)

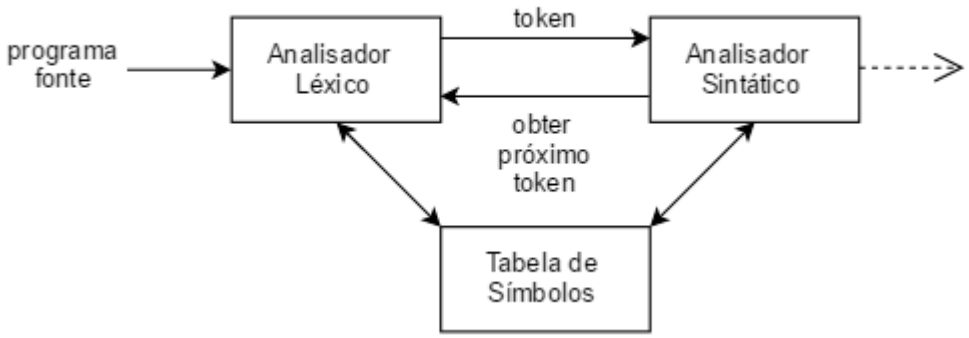
Um analisador léxico, também chamado de *scanner*, terá como entrada uma *string* de caracteres que será dividida em *tokens*. Além disso, o analisador léxico filtra tudo o que separa os *tokens*, como por exemplo, espaços em branco, quebras de linha e comentários. (MOGENSEN, 2011)

O objetivo principal de um analisador léxico é facilitar a análise sintática, que é a fase subsequente. Em teoria, o trabalho feito durante a análise léxica pode ser feito em conjunto com a análise sintática, porém, existem razões como eficiência, modularidade e tradição para se manter as duas fases separadas.

O analisador sintático inicia pedindo ao analisador léxico qual é o próximo *token*. O analisador léxico então faz a varredura no código até definir e gerar o próximo *token* e então o retorna para o analisador sintático. Esta interação ocorre até o fim da análise sintática, *token* a *token*, percorrendo todo o código-fonte.

Tanto as análises léxica e sintática podem fazer interações com a tabela de símbolos, armazenando informações para futuras etapas do compilador. Esse processo pode ser visto na Figura 2, no qual é apresentado o funcionamento do analisador léxico e sua interação com o analisador sintático e a tabela de símbolos.

FIGURA 2 – INTERAÇÃO ENTRE ANALISADOR LÉXICO E SINTÁTICO



FONTE: AHO, 1995

Na análise léxica, as especificações são geralmente escritas utilizando Expressões Regulares (ER). Os lexemas gerados são classificados através de um modelo matemático de um sistema com entradas e saídas discretas, chamado autômato finito (AF). (MOGENSEN, 2011)

3.2.1 Tokens, Padrões e Lexemas

Os termos "*token*", "padrão" e "lexema" tem seus significados específicos quando falamos de análise léxica. Um lexema é uma sequência de caracteres reconhecido pelo padrão de algum *token*. Um padrão são regras que associam os lexemas aos tokens, podendo existir um conjunto de lexemas diferentes representados pelo mesmo *token*. A Tabela 2 apresenta um exemplo dos *tokens* gerados para o seguinte enunciado em C:

```
int a = 32 + 415;
```

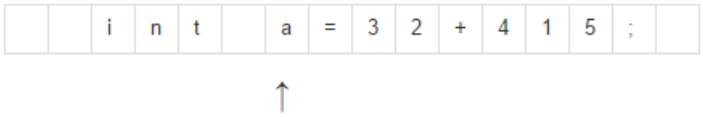
TABELA 2 – ANÁLISE LÉXICA

Lexema	Token	Padrão
int	Tipo	Int, float, string, etc.
a	Identificador	Letra seguida por letras e/ou números
=	Operador de atribuição	Operador =
32	Número	Qualquer representação numérica
+	Operador de adição	Operador +
415	Número	Qualquer representação numérica
;	Ponto e vírgula	Ponto e vírgula

FONTE: o próprio autor

Na Figura 3 é apresentado um exemplo de como a varredura em uma cadeia de caracteres é feita. Neste caso, a função de reconhecimento de *token* salta 2 espaços em branco, reconhece o identificador “int” como um *token*, salta um espaço em branco e passa a ler o próximo caractere da cadeia. A seta indica qual é o caractere que está sendo lido.

FIGURA 3 – VARREDURA EM UMA CADEIA DE CARACTERES



FONTE: o próprio autor

3.2.2 Expressões Regulares

Um conjunto de inteiros constantes ou o conjunto dos nomes de variáveis são conjuntos de palavras, na qual cada letra pertence a um alfabeto. Tais conjuntos, são chamados de linguagem. Para os inteiros, o alfabeto consiste nos dígitos entre 0-9 e para os nomes de variáveis o alfabeto contém ambos, letras e dígitos (e talvez um ou outro caractere, como o *underscore*). (MOGENSEN, 2011)

Dado um alfabeto, podemos definir um conjunto de palavras através das ER, que é uma notação compacta e fácil para os humanos usarem e entenderem. A ideia é que ERs que descrevam um conjunto de palavras simples possam ser combinadas com expressões regulares que descrevam um conjunto de palavras mais complexo.

Uma ER sobre um alfabeto Σ é definida na Tabela 3.

TABELA 3 – EXPRESSÃO REGULAR PARA UM ALFABETO Σ

L = Linguagem gerada	ER
L(ϕ)	{}
L(ε)	{ε} = ε
L(a)	{a} = a
L(a|b) = L(a) ∪ L(b) (união)	{a} ∪ {b} = {a,b} = a + b
L(ab) = L(a)L(b) (concatenação)	{a}{b} = {ab} = a.b ou ab
L(a*) = L(a)* (repetição ou fechamento)	{a}* = a*

FONTE: (LOUDEN, 2004)

Somente as operações de união, concatenação e repetição definem conjuntos regulares.

3.2.2.1 Exemplos

Para o alfabeto Σ = {a,b} são definidas as seguintes expressões regulares:

- a* , conjunto contendo nenhum, um ou múltiplos *a´*s. Ex.: aaaaa, aa, ε
- a , conjunto de apenas um caractere *a*. Ex.: a
- ab* , conjunto iniciado em *a* e contendo nenhum, um ou múltiplos *b´*s ao final. Ex.: abbb, a, ab
- (a+b)* , conjunto pode ser vazio ou conter qualquer combinação de *a´*s e *b´*s. Ex.: aaaab, ababab, aabbaabb, aaa, bbbba
- (ba)* , conjunto pode ser vazio ou conter repetições de *ba´*s. Ex.: ba, baba, ε
- b+a(ba)* , conjunto deve iniciar com *b* ou *a* e conter ou não repetições de *ba´*s. Ex.: bbababa, abababa, b

3.2.3 Autômatos Finitos

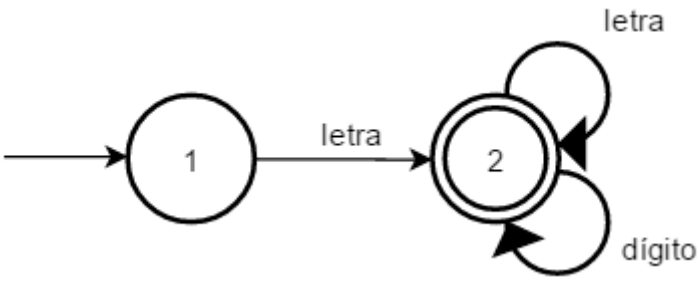
Um programa reconhecedor de uma linguagem recebe como entrada uma palavra e tem como saída uma indicação se esta palavra foi aceita ou não pela linguagem. Autômatos finitos (AF), ou máquinas de estados finitos, tem um poder para definir uma linguagem da mesma maneira que expressões regulares. As expressões regulares podem ser compiladas e convertidas em AFs e normalmente é muito mais simples trabalhar com eles do que com ERs. Os AFs podem ser utilizados para descrever o processo de reconhecimento de *tokens* em cadeias de caracteres de entrada, sendo possível a utilização deles para construir a análise léxica do compilador. (LOUDEN, 2004)

Seja o seguinte exemplo: o padrão para identificadores de variáveis definidos na maioria das linguagens de programação é dado pela seguinte definição regular:

```
identificador = letra(letra + dígito)*
```

Esta representação indica que a cadeia de caracteres deve iniciar com uma letra e continuar com qualquer sequência de letras e/ou dígitos. O processo de reconhecimento de uma cadeia como essa pode ser descrito pela Figura 4.

FIGURA 4 – UM AUTÔMATO FINITO PARA IDENTIFICADORES



FONTE: LOUDEN, 2004, pg. 47.

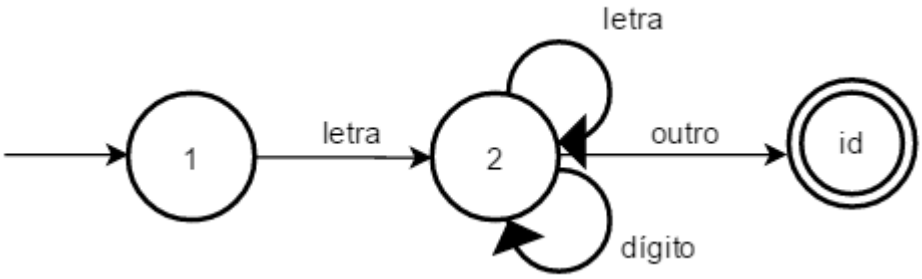
Autômatos finitos podem ser determinísticos (Deterministic Finite Automaton - DFA) ou não-determinístico (Nondeterministic Finite Automaton - NFA). A abordagem *determinística* segue uma regra única: para cada combinação de estado e símbolo de entrada não existe mais de uma possibilidade de escolha. Já abordagem *não-determinística* permite, para cada par, possuir mais de uma transição para fora de um estado para um mesmo símbolo de entrada. As duas variações possuem um número finito de estados, consequentemente, o nome “autômato finito”. (AHO et al., 1995)

Ambos os autômatos finitos são capazes de reconhecer os conjuntos regulares, ou seja, os dois tipos podem reconhecer precisamente o que as expressões regulares podem denotar. Enquanto os DFAs podem reconhecer se uma palavra pertence ou não a uma linguagem mais rapidamente que os NFAs, um DFA pode ser muito maior que um NFA equivalente. (AHO, 1995)

DFAs reconhecem as palavras de forma mais eficiente. Conforme citado acima, para o DFA existe apenas uma transição para cada estado e caractere. Porém, mesmo utilizando a definição matemática de DFAs, é preciso realizar alguns ajustes para que possamos programá-la ao código. Por exemplo, a definição matemática não nos diz o que fazer na presença de erros, também não nos define a ação relacionada a um estado de aceitação, ou um casamento de caracteres durante uma transação. (LOUDEN, 2004)

Na Figura 5 o autômato finito representado na Figura 4 é reescrito com uma nova transição “outro”, que indica que o caractere encontrado não pertence ao lexema que está sendo gerado. Quando essa transição é encontrada, o algoritmo devolve a identificação do *token* gerado.

FIGURA 5 – AUTÔMATO FINITO PARA UM IDENTIFICADOR COM DELIMITADOR E VALOR DE RETORNO



FONTE: LOUDEN, 2004, pg. 54.

3.3 ANÁLISE SINTÁTICA

A função principal da análise sintática, também conhecida como *parser*, é determinar se o texto de entrada é uma sentença sintaticamente válida para a linguagem de origem. Com base na gramática da linguagem de origem, a análise sintática analisa a sequência com que os *tokens* provenientes do código fonte se apresentam, efetuando a síntese da árvore sintática do mesmo. Podemos afirmar então, que antes de tudo, o analisador léxico pode ser visto como um validador, aceitando apenas cadeias de caracteres cujo conjunto forma a linguagem de origem a que se refere o analisador. (NETO, 1987)

Caso o analisador detecte uma sentença que não pertença à linguagem de origem, o analisador sintático deve indicar ao programador os erros de sintaxe encontrados, preferencialmente, indicando o tipo do erro, a sua causa e também a linha em que o erro foi encontrado.

Enquanto a análise léxica tem o papel de separar uma entrada de caracteres em *tokens*, o propósito da análise sintática é recombinar estes *tokens* em algo que reflita a estrutura de um texto. Normalmente, este “algo” é chamado de árvore de análise sintática do texto. Como é uma estrutura em árvore, as folhas são os *tokens* gerados pela análise léxica. Caso a leitura dessas folhas seja feita da esquerda para a direita, a sequência é a mesma que o texto de entrada. Consequentemente, o que é importante na

árvore sintática é como as suas folhas são combinadas para gerar a estrutura da árvore e como os nós são rotulados. (MOGENSEN, 2011)

Métodos mais avançados são requeridos nesta fase do que na fase léxica. Para se obter uma execução eficiente, antes de construir um analisador sintático, é preciso transformar uma notação compreensiva ao ser humano em uma notação de baixo nível que seja compreendida pelo computador. Esta notação compreendida pelo ser humano é chamada de gramática livre de contexto.

Uma gramática livre de contexto utiliza operações e convenções para nomes muito semelhantes às utilizadas por ERs. A principal diferença é que a estrutura de dados utilizada para representar uma estrutura sintática é recursiva, ao invés de linear, como no caso da análise léxica. Por exemplo, para varrer uma matriz bidimensional podemos aninhar declarações "*for"* e a estrutura sintática deve possibilitar este aninhamento, o que não é permitido em expressões regulares. Os algoritmos utilizados para representar essas estruturas também são bem diferentes, pois utilizam ativações recursivas ou uma pilha gerenciada explicitamente. (LOUDEN, 2004)

Diferentemente da análise léxica, que são representadas essencialmente por AFs, a análise sintática envolve uma escolha entre diversos métodos distintos, cada um com sua vantagem e desvantagem, apresentando propriedades e recursos próprios. Na verdade, existem duas categorias que englobam os algoritmos que constroem a análise sintática e a árvore sintática correspondente. São os métodos descendentes (*top-down*) e ascendentes (*bottom-up).*

Gramáticas são ferramentas essenciais para a especificação de uma linguagem e possuem aspectos importantes. Primeiramente, uma gramática serve para impor uma estrutura em uma sequência linear de *tokens*, que é o programa. Esta estrutura é importante porque a semântica do programa é especificada através dela.

Segundo, utilizando técnicas de linguagens formais, uma gramática pode ser utilizada para construir um analisador sintático para ela automaticamente. Este processo pode facilitar a construção de um compilador.

Terceiro, gramáticas ajudam programadores a escreverem programas sintaticamente corretos e apresentam respostas para perguntas mais detalhadas sobre a sintaxe. (GRUNE, 2001)

Nas seções a seguir, serão apresentadas uma descrição geral destes processos da análise sintática, iniciando com a gramática livre de contexto, derivações, árvores sintáticas e ambiguidade, e concluindo com um estudo sobre os métodos descendente e ascendente.

3.3.1 Gramática Livre de Contexto

Para descrever uma linguagem de programação, precisamos de uma notação mais poderosa do que expressões regulares. Como ERs, gramáticas livres de contexto descrevem conjuntos de palavras, ou seja, uma linguagem. Uma linguagem é definida sobre algum alfabeto, por exemplo, um conjunto de *tokens* produzidos por um analisador léxico ou um conjunto de caracteres alfanuméricos. Os símbolos em um alfabeto são chamados de símbolos terminais. (COOPER, 2012)

A gramática livre de contexto define recursivamente um conjunto de palavras. Cada conjunto é denotado por um nome (variável), que é chamado de símbolo não-terminal. Um conjunto de símbolos não-terminais é separado a partir do conjunto de símbolos terminais. Um dos símbolos não-terminais é escolhido para denotar a linguagem descrita por uma gramática. Isto é chamado de símbolo inicial da gramática. Os conjuntos são descritos através de um processo de produções. Cada produção descreve algumas das possíveis palavras que estão contidas no conjunto indicado por um símbolo não-terminal. Uma produção tem a seguinte forma:

$$N \rightarrow X_1...X_n$$

na qual N é o símbolo não-terminal e $X_1...X_n$ é vazio ou contém um ou mais símbolos, em que podem ser terminais ou não-terminais. O objetivo desta notação é dizer que um conjunto denotado por N contém palavras que são obtidas concatenando palavras de um conjunto denotado por $X_1...X_n$. Neste cenário, um símbolo terminal contém apenas um elemento, assim como caracteres de alfabetos em expressões regulares. (COOPER, 2012)

Alguns exemplos de gramáticas:

$$A \rightarrow a$$

na qual o conjunto denotado pelo símbolo não-terminal A contém a palavra de um caractere a.

$$A \rightarrow aA$$

na qual o conjunto denotado pelo símbolo não terminal A contém todas as palavras formadas com um a em frente a todas as palavras denotadas por A. Juntas, essas duas produções indicam que A contém todas as sequências não vazias e isto equivale a ER a+. (COOPER, 2012)

É possível definir uma gramática equivalente a expressão regular a* através de duas produções:

$$S \rightarrow \varepsilon$$

$$S \rightarrow aS$$

na qual a primeira produção indica que uma palavra vazia faz parte do conjunto de S e a segunda produção tem como símbolo inicial *a*, seguido pela variável S.

3.3.2 Derivações

Uma forma de descobrirmos se uma sentença pertence à linguagem definida pela gramática é através de derivações. Iniciando com um símbolo inicial, e então repetidamente substituir qualquer símbolo não-terminal por um de sua direita, conforme o exemplo de derivação logo abaixo.

Considere a gramática extremamente simplificada a seguir:

$$S \rightarrow id = (num + num * num) – num;$$

Para a sentença em C:

```
a = (3 + 5 * 4) - 2;
```

É possível derivar substituindo sempre a primeira da esquerda para a direita, da seguinte forma:

S → E

S → id = E;

S → id = (E) - E;

S → id = (E + E) - E;

S → id = (num + E) - E;

S → id = (num + E * E) - E;

S → id = (num + num * E) - E;

S → id = (num + num * num) - E;

S → id = (num + num * num) - num;

Existem diversas variações diferentes para a mesma sentença. A “derivação esquerda” sempre expande o símbolo não-terminal mais à esquerda e a “derivação direita” expande o símbolo não-terminal que está sempre mais à direita. (APPEL, 1998)

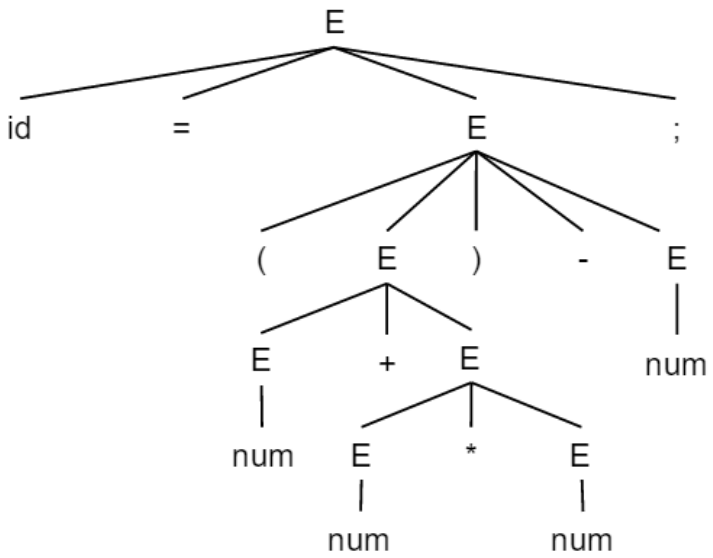
3.3.3 Árvores sintáticas

Uma árvore sintática é construída com a conexão de cada símbolo em derivação com um símbolo que foi derivado. Os nós interiores são rotulados como não-terminais, os nós-folha são rotulados por terminais e o filho de cada nó interno representa a substituição do não-terminal associado em um passo de derivação. (APPEL, 1998)

Por exemplo, a árvore sintática representada na Figura 6, corresponde a derivação exemplificada no tópico anterior pela sentença:

```
a = (3 + 5 * 4) - 2;
```

FIGURA 6 – ÁRVORE SINTÁTICA



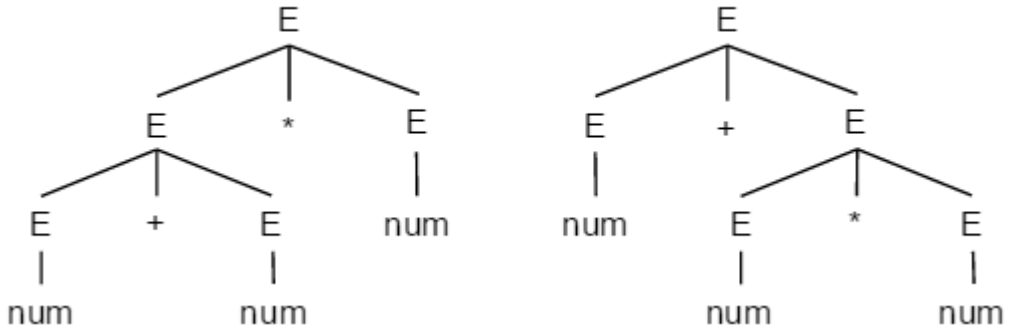
FONTE: o próprio autor

3.3.4 Ambiguidade

A ambiguidade, dentro do estudo dos compiladores, representa um problema semântico que consiste na dupla interpretação de uma sentença formal. No caso das

gramáticas, uma gramática é ambígua se ela pode derivar uma sentença em duas árvores sintáticas diferentes. A sentença `a = (3 + 5 * 4) - 2;` é ambígua e dá margem a dupla interpretação. A sentença que está entre os parênteses `3 + 5 * 4` pode derivar dois resultados diferentes: `(3 + 5) * 4 = 32` versus `3 + (5 * 4) = 23`. A Figura 7 demonstra a diferença na construção dessas duas árvores. (APPEL, 1998)

FIGURA 7 – ÁRVORES SINTÁTICAS DIFERENTES PARA A MESMA SENTENÇA



FONTE: o próprio autor

Portanto, gramáticas ambíguas podem ser um problema sério ao se construir um compilador, pois elas não especificam com precisão a estrutura sintática de um programa. Podemos comparar a gramática ambígua com um autômato não-determinístico, no qual dois caminhos separados podem aceitar a mesma cadeia de caracteres. No entanto, a ambiguidade em uma gramática não pode ser removida facilmente como é removido o não-determinismo em autômatos finitos, pois não conseguimos construir um algoritmo para isso. (LOUDEN, 2004)

Para eliminarmos a ambiguidade, é preciso transformar a gramática. Transformando a gramática, é possível forçar a construção de uma árvore de análise sintática correta, a qual é eliminada a ambiguidade. Evidentemente, é preciso definir em um caso ambíguo qual é a árvore correta e se ela reflete o significado para a tradução em código-objeto.

No caso citado acima, para a sentença `a = (3 + 5 * 4) - 2;`, devemos considerar qual é o cálculo correto. Se avaliarmos primeiro a adição `(3 + 5 = 8)` e depois a multiplicação `(8 * 4 = 32)`, o resultado será `(a = 32 - 2 = 30)`.

Porém, se avaliarmos primeiro a multiplicação `(5 * 4 = 20)` e depois a adição `(3 + 20 = 23)`, o resultado será `(a = 23 – 2 = 21)`. A convenção usual da matemática determina a segunda interpretação como a correta. Isso se deve a precedência da multiplicação sobre a adição.

A seguir serão apresentadas as diferenças entre dois tipos de análise sintática: a descendente e a ascendente.

3.3.5 Análise sintática descendente

O principal objetivo de um analisador sintático descendente é escolher a alternativa correta para não-terminais já descobertos. Se inicia construindo um nó do topo de uma árvore, chamado de símbolo inicial. Os nós do topo de uma sub árvore são construídos antes que qualquer nó abaixo deles. O nome “descendente”, ou em inglês *Top-Down*, vem da forma que a árvore é percorrida, em pré-ordem, da raiz para as folhas. (GRUNE, 2001; LOUDEN, 2004)

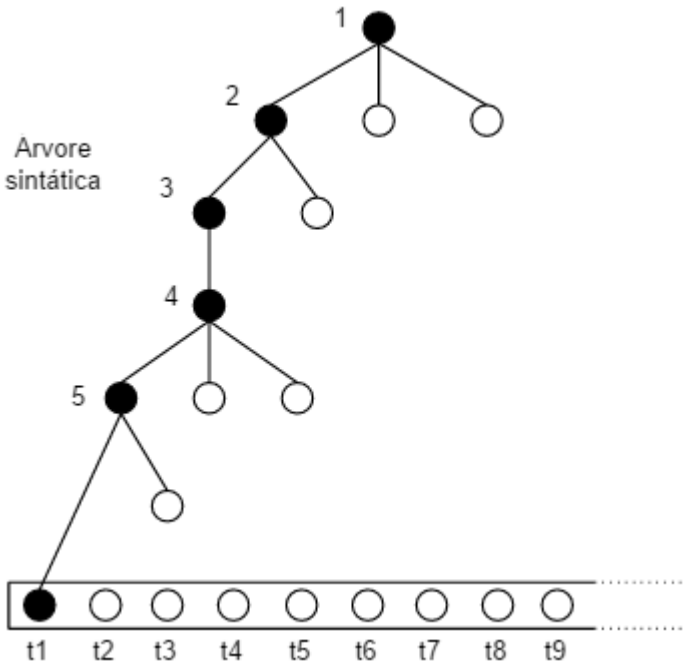
Utilizando as informações fornecidas pelo código fonte, o analisador descendente determina qual a alternativa correta para o primeiro nó. O processo de determinar a alternativa correta para o nó filho mais à esquerda é repetido em cada nível, até encontrar o último filho à esquerda, que é um símbolo terminal. O símbolo terminal então corresponde ao primeiro *token* do programa. Isso ocorre porque o analisador sintático descendente escolhe as alternativas dos nós superiores precisamente. (GRUNE, 2001)

O analisador prossegue construindo o próximo nó em pré-ordem. Vejamos a Figura 8, na qual os pontos preenchidos representam os nós que já foram construídos e os pontos sem preenchimento indicam os nós que os rótulos já foram descobertos, mas ainda não foram construídos. Podemos notar que ainda não sabemos nada sobre o restante da árvore sintática, apenas descobrimos o *token* t1. (GRUNE, 2001)

Existem duas formas de analisadores sintáticos descendentes: analisadores com retrocesso e analisadores preditivos. O analisador com retrocesso testa diferentes possibilidades de análise sintática de entrada, realizando o retrocesso se alguma delas falhar, verificando todas as possibilidades de derivação. Já o analisador preditivo tenta prever a construção seguinte na cadeia de entrada com base em um ou mais rótulos de verificação à frente. Analisadores com retrocesso são mais lentos e muitas vezes

necessitam tempo exponencial para execução. Portanto, eles não são indicados para compiladores e por isso não serão abordados neste trabalho. (LOUDEN, 2004)

FIGURA 8 – UM ANALISADOR SINTÁTICOS DESCENDENTE RECONHECENDO O PRIMEIRO TOKEN DE UM CÓDIGO FONTE



FONTE: GRUNE, 2001, pg. 114.

3.3.5.1 Analisador Sintático LL(1)

O analisador sintático preditivo que mais se demonstra adequado para este trabalho é o LL(1). O primeiro "L" do nome LL(1) quer dizer *left* e vem do fato de que a análise sintática é feita da esquerda para a direita, sempre trabalhando com a variável mais à esquerda. O segundo "L" se refere que o analisador acompanha uma derivação sempre à esquerda para cada cadeia de entrada. O número 1 entre parênteses significa que é utilizado apenas um símbolo de verificação à frente.

O ideal é que todas as produções estejam na Forma Normal de Greibach (FNG), na qual todas as produções são do tipo "A → a α", como na gramática G1, no qual iremos verificar se a sentença "+a*ba" ϵ G1.

G1: E → a | b | +EE | *EE

Utilizando a técnica de verificação da esquerda para a direita, trabalhando sempre com a variável mais à esquerda, é possível notar a facilidade da análise. Essa é a razão dessa classe de analisadores sintáticos serem chamados de LL(*left to right parsing, producing leftmost derivation* – Analisador da esquerda para à direita, produzindo derivações mais à esquerda). (KOWALTOWSKI, 1983)

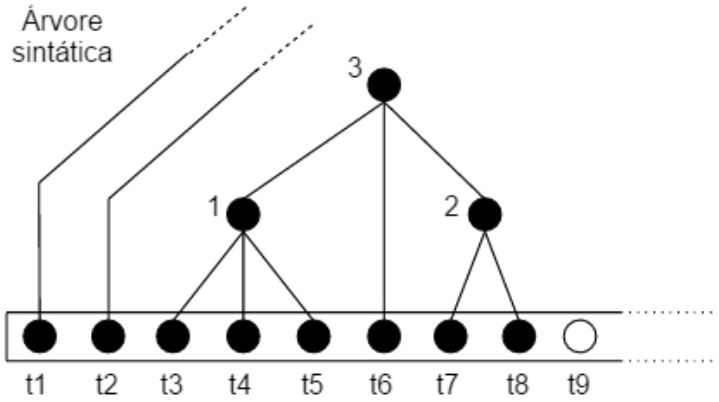
3.3.6 Análise Sintática Ascendente

O analisador sintático ascendente, como o próprio nome já diz, é construído de forma ascendente, ou em inglês *Bottom-Up*. O analisador é construído em pós-ordem, em que o topo das sub árvores são construídos após os nós inferiores serem construídos. Quando o analisado ascendente constrói um nó, todos nós filhos já foram construídos, e estão presentes na árvore e seus rótulos já são conhecidos. (GRUNE, 2001)

O analisador ascendente sempre irá construir o nó que está no topo da primeira sub árvore completa. Uma sub árvore completa é aquela que todos os nós filhos já foram construídos. Os *tokens* são considerados sub árvores de nível 1 e são construídos assim que são conhecidos. (GRUNE, 2001)

A Figura 9 demonstra o analisador ascendente após ter sido construído o primeiro, segundo e terceiro nó. Os pontos preenchidos representam os nós que já foram construídos. O primeiro nó expandiu os *tokens t3*, *t4* e *t5*; o segundo expandiu os *tokens t7* e *t8*; e o terceiro expandiu o primeiro nó, o *token t6* e o segundo nó. Nada se sabe ainda sobre a existência de outros nós, mas ramificações foram construídas para cima dos *tokens t1* e *t2* que ainda não fazem parte de uma sub árvore como os *tokens t3* a *t8*, caso contrário, essa sub árvore seria a primeira a ser construída. Em suma, o principal objetivo do analisador sintático ascendente é repetidamente encontrar o primeiro nó cujo todos os filhos já foram construídos. (GRUNE, 2001)

FIGURA 9 – UM ANALISADOR ASCENDENTE CONSTRUINDO O PRIMEIRO, SEGUNDO E TERCEIRO NÓS



FONTE: GRUNE, 2001, pg. 115.

Diferente da análise descendente, na qual apenas o algoritmo LL(1) é praticável e eficiente, existem muitos algoritmos que podem ser implementados para um analisador ascendente. Todos esses algoritmos diferem apenas na quantidade de símbolos de verificação à frente que serão utilizados. A última parte do algoritmo, onde é feita a redução destes símbolos a não-terminais, é a mesma técnica para todos eles.

Alguns desses algoritmos são (AHO, 1995; GRUNE, 2001):

- Análise de precedência: utilizado em analisadores mais simples para expressões aritméticas;
- BC(*k,m*): *bounded-context* (contexto limitado) com *k tokens* no contexto da esquerda e *m tokens* no contexto da direita; relativamente forte, muito popular na década de 70, especialmente o BC(2,1), mas hoje já é considerado ultrapassado;
- LR(0): *left to right parsing, producing reversed rightmost derivation*. Muito utilizado para aprender a teoria dos algoritmos LRs, mas não recomendado para ser utilizado de forma mais prática;
- SLR(1): *Simple* LR é uma versão melhorada do LR(0), mas ainda possui algumas limitações;
- LR(1): como o LR(0), mas verificando um símbolo à frente. Porém, tem um custo muito alto;

- LALR(1): *Lookahead* LR é uma versão menos custosa do que a LR(1), com um custo intermediário, podendo até, com algum esforço, ser implementado de forma mais eficiente. Este método irá funcionar na maioria das gramáticas das linguagens de programação e é o mais utilizado hoje em dia.

Analisadores LR são complexos e trabalhosos para serem construídos manualmente para uma gramática típica de uma linguagem de programação. Normalmente existe a necessidade de utilizar uma ferramenta especializada que irá gerar um analisador LR automaticamente. Caso a gramática contenha ambiguidade ou outras construções que sejam difíceis de decompor, a ferramenta pode localizá-las e informar ao projetista do compilador as ocorrências encontradas. (AHO, 1995)

3.4 ANÁLISE SEMÂNTICA

Um compilador precisa ir além de reconhecer se uma sentença pertence ou não a uma gramática. O programa pode estar gramaticalmente correto e ainda conter erros sérios que podem comprometer a sua execução. Para verificar estes erros, o compilador executa um nível adicional de verificação que considera cada sentença em seu contexto atual. Por exemplo, os analisadores léxicos e sintáticos não conseguem nos informar se o número de parâmetros informado na chamada de uma função é o mesmo que em sua declaração, esta análise requer uma verificação de longo alcance, onde é validada a relação entre a chamada da função com a sua declaração. Esta é a principal tarefa de um analisador semântico. (APPEL, 1998; COOPER, 2012; GRUNE 2001)

Para assegurar que certos tipos de erros sejam detectados e reportados é preciso que o compilador verifique se o programa segue as convenções sintáticas e semânticas da linguagem fonte. Essa checagem é chamada de verificação semântica estática. Abaixo seguem alguns exemplos de verificações semânticas estáticas. (AHO, 1995)

- *Identificador não declarado.* O compilador deve relatar um erro caso um identificador seja utilizado pelo programa sem ter sido declarado.

- *Identificador redeclarado.* No caso de um mesmo identificador ser declarado mais de uma única vez no mesmo escopo o compilador apresenta o erro de duplicidade.
- *Verificação de tipos.* Verifica se um operador foi aplicado a um operando de tipo incompatível, por exemplo, atribuição de uma *string* a um inteiro.
- *Atribuição a constantes.* Constantes não podem ser alteradas, caso uma constante seja alterada no programa, o compilador mostra o erro que uma constante não pode ser atribuída.
- *Chamada de funções com número de parâmetros inconsistentes.* Uma função, quando chamada, deve possuir o mesmo número de parâmetros que a sua declaração.

A análise semântica requer a construção de uma tabela de símbolos para acompanhar e verificar o significado das declarações e efetuar a verificação de tipos e inferência em expressões e declarações, de forma a determinar sua correção pelas regras da linguagem. (LOUDEN, 2004)

As tarefas da análise semântica são chamadas de ações semânticas. As ações semânticas realizam todo tipo de operações, necessárias à compilação, e que não estejam compreendidas nas funções das análises léxica e sintática. Nos compiladores que são dirigidos por sintaxe, determinando que todo conteúdo semântico de um programa é fortemente relacionado a sua sintaxe, o analisador sintático aciona a execução destas ações no momento em que ocorre um reconhecimento ou quando determinadas transições ocorrem durante a análise do código fonte. (NETO, 1987)

Ao contrário da análise sintática, que podemos formalizar facilmente utilizando um método padrão como o de Backus-Naur (BNF), a análise semântica não é tão clara ao ponto de ser expressa formalmente através de uma BNF. Linguagens de programação possuem características que são difíceis e algumas até impossíveis de serem descritas por tal método. Por exemplo, a regra segundo a qual todas as variáveis devem ser declaradas antes de serem referenciadas. (LOUDEN, 2004; NETO, 1987; KOWALTOWSKI, 1983)

A forma mais usual de descrever mais detalhadamente a estrutura de uma linguagem de programação é a gramática de atributos. Identificando atributos, ou propriedades, é possível escrever regras semânticas, que expressam como a computação desses atributos se relacionam com as regras gramaticais da linguagem. A gramática de atributo é convenientemente utilizada para especificar processamentos sensíveis ao contexto. Elas também são mais úteis para linguagens que obedeçam ao princípio da semântica dirigida pela sintaxe. (LOUDEN, 2004)

3.4.1 Gramática de atributos

Uma gramática de atributos consiste em uma gramática livre de contexto argumentada por um conjunto de regras que especificam a computação. Cada regra define uma propriedade, ou atributo, em termos de valores de outros atributos. As regras associam o atributo com um símbolo específico da gramática. Cada símbolo da gramática que ocorre em uma árvore sintática corresponde a um atributo. As regras são funcionais, elas não implicam em nenhuma validação de ordenação e definem cada valor do atributo de forma exclusiva. (COOPER, 2012)

Essas regras são chamadas de regras semânticas. Um conjunto de regras semânticas é uma Definição Dirigida pela Sintaxe (DDS). (KOWALTOWSKI, 1983)

Normalmente, as gramáticas de atributos são descritas em forma de tabelas, com cada regra gramatical listada junto com o conjunto de regras semânticas, conforme a Tabela 4.

TABELA 4 – GRAMÁTICA DE ATRIBUTOS

Regra gramatical	Regras semânticas
Regra 1	Equações de atributos associadas
.	.
.	.
.	.
Regra n	Equações de atributos associadas

FONTE: LOUDEN, 2004; pg.263

Atributos são valores anexados a um ou mais nós em uma árvore sintática. Podemos distinguir os atributos baseado na direção do fluxo do seu valor:

- *Atributos sintetizados:* passam informações semânticas acima em uma árvore sintática, de suas folhas para a raiz (|↑|). Uma regra que defina um atributo para o lado esquerdo da produção, cria um atributo sintetizado. Um atributo sintetizado pode gerar valores do próprio nó, de seus descendentes e de constantes; (COOPER, 2012; MOGENSEN, 2011)
- *Atributos herdados:* passam informações semânticas abaixo ou lateralmente em uma árvore sintática (|↓| ou |+|). Uma regra que defina um atributo para o lado direito da produção, cria um atributo herdado. Uma vez que a regra de atribuição pode nomear qualquer símbolo utilizado na produção correspondente, um atributo herdado pode gerar valores do próprio nó, de seu nó "pai", de algum nó "irmão" e de constantes. (COOPER, 2012; MOGENSEN, 2011)

Como exemplo, considere a gramática que gera números positivos, dada a seguir:

número → *número dígito* | *dígito*

dígito → 0|1|2|3|4|5|6|7|8|9

O atributo mais significativo de um número é o seu valor, que daremos o nome de *val*. Um dígito, como base, tem o valor do dígito que ele representa. Por exemplo, a regra gramatical *dígito*→5 indica que o dígito tem o valor 5. Podemos expressar esta regra pela regra semântica *dígito.val*=5, e associamos esta regra a regra gramatical *dígito*→5.

Se o número for baseado na regra *número*→*dígito*, o número conterá apenas um dígito e a regra semântica que expressa esse fato é

número.val→*dígito.val*

Se o número contiver mais de um dígito, precisamos derivar. Ao fazer a derivação, devemos expressar a relação entre o valor do símbolo à esquerda da regra gramatical e os valores dos símbolos à direita. Como teremos duas ocorrências de *número* na regra gramatical, devemos fazer a diferenciação entre elas, pois o *número* à esquerda terá um valor diferente do *número* à direita. Essa diferenciação é feita através de índices. Realizando a derivação, a regra fica assim:

$número_1.val \rightarrow número_2.val\ dígito.val$

Vai-se utilizar como exemplo o número 54. Este número derivado à esquerda, é expresso da seguinte forma:

número => *número dígito* => *dígito dígito* => `5` *dígito* => `5 4`.

Considerando a regra gramatical *número₁.val*→*número₂.val dígito.val* como primeiro passo de derivação, temos a variável *número₂* que corresponde ao dígito `5`, e *dígito* que corresponde ao dígito `4`. Respectivamente, os valores de cada um são `5` e `4`. Para que *número₁* obtenha o valor `54`, devemos carregar o `5` uma casa decimal para a esquerda e somarmos ao `4`. Podemos criar essa equação multiplicando *número₂* por `10` e somando o resultado ao valor de *dígito*: `54 = 5 * 10 + 4`. Essa equação matemática corresponde a seguinte equação de atributo

número₁.val→*número₂.val* * 10 + *dígito.val*

A gramática de atributos completa para este exemplo é dada na tabela 5.

TABELA 5 – GRAMÁTICA DE ATRIBUTOS COMPLETA

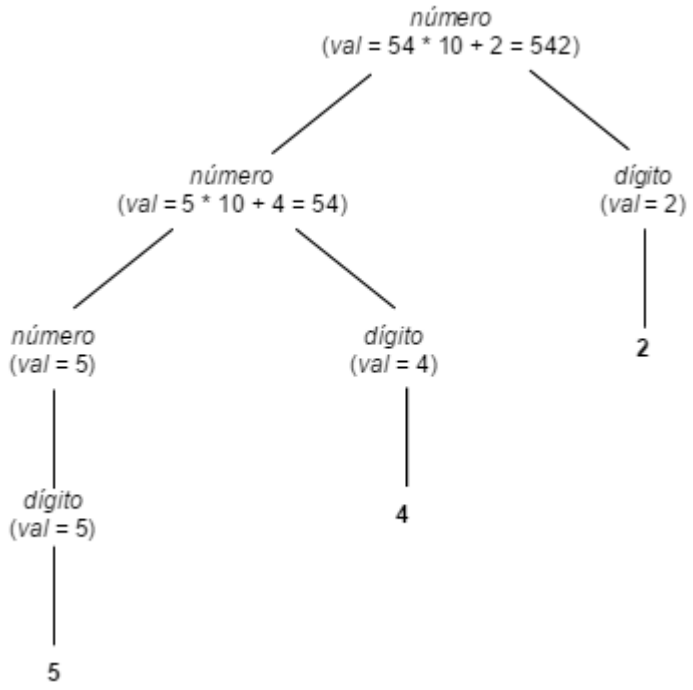
Regra gramatical	Regras semânticas
número₁→*número₂ dígito*	*número₁.val*→*número₂.val* * 10 + *dígito.val*
número→*dígito*	*número.val*→*dígito.val*
dígito→0	*dígito.val* = 0
dígito→1	*dígito.val* = 1
dígito→2	*dígito.val* = 2
dígito→3	*dígito.val* = 3
dígito→4	*dígito.val* = 4
dígito→5	*dígito.val* = 5
dígito→6	*dígito.val* = 6
dígito→7	*dígito.val* = 7
dígito→8	*dígito.val* = 8
dígito→9	*dígito.val* = 9

FONTE: LOUDEN, 2004; pg.265

Ainda utilizando a gramática acima, podemos expressar o significado da equação de atributos para qualquer cadeia em particular utilizando a árvore sintática para aquela cadeia. Na Figura 10 pode-se visualizar a árvore sintática para o número `542`. O cálculo correspondente à equação de atributos apropriada é dado abaixo de cada nó interior. Para um compilador computar e utilizar os atributos definidos pelas equações da gramática de atributos, é importante enxergar as equações de atributos como computações na

árvore de análise sintática, pois elas definem as restrições de ordem na computação dos atributos. (LOUDEN, 2004)

FIGURA 10 – ÁRVORE DE ANÁLISE SINTÁTICA DEMONSTRANDO AS COMPUTAÇÕES DE ATRIBUTOS



FONTE: LOUDEN, 2004, pg.265

3.4.2 Tabela de símbolos

Uma função importante de um compilador é gravar os nomes de variáveis utilizadas no código fonte e coletar informações sobre vários atributos de cada nome.

Conceitualmente, a tabela de símbolos se ajusta melhor na fase de análise semântica, mas apesar de não ter sido comentada, com poucas exceções, em seções anteriores deste trabalho, a tabela de símbolos é utilizada por praticamente todas as fases do compilador. Ela é utilizada para controlar as informações de escopo e dos atributos a respeito dos identificadores. A tabela de símbolos é pesquisada a cada vez que um identificador é encontrado no código fonte. Caso seja uma declaração, o identificador é inserido na tabela de símbolos e seus atributos podem ser adicionados à medida que as informações se tornarem disponíveis. (AHO, 1995)

Os atributos de um identificador podem prover informações sobre a alocação para um nome, sobre um tipo de dado e também qual o escopo deste identificador. No caso de procedimentos ou funções, podem ser armazenados a quantidade e o tipo dos parâmetros, o método para passar esses parâmetros (por valor ou por referência), e o tipo de seu retorno. (GRUNE, 2001)

Apesar da análise sintática conseguir nos informar se a sintaxe de uma declaração de tipos está correta, ela não consegue fornecer a informação se uma variável foi ou não declarada antes de ser utilizada. A gramática livre de contexto não tem como combinar uma instância de um nome de variável com outra, isso requer uma análise mais profunda. A tabela de símbolos é criada para resolver este tipo de problema, o compilador insere o nome da variável declarada e toda vez que uma variável é referenciada no código, uma pesquisa é feita na tabela de símbolos, verificando se a referência existe na tabela. Caso não exista a referência, o compilador retorna o erro de variável não declarada.

A estrutura de dados utilizada para a tabela de símbolos deve ser de tal forma que o compilador encontre os registros de maneira rápida e que grave ou obtenha as informações de um registro rapidamente. Uma estrutura de dados padrão para isso é a tabela *hashing*, que associa chaves de pesquisa a valores. Outras estruturas de árvores também podem ser utilizadas. (LOUDEN 2004)

Na Tabela 6 é dado um exemplo de uma tabela de símbolos simples com suas propriedades.

TABELA 6 – EXEMPLO DE UMA TABELA DE SÍMBOLOS

#	Id	Categoria	Tipo	Escopo	Num. Params	Pass by	Valor
1	Ca	CONSTANTE	INT	GLOBAL	0		3
2	Cs	CONSTANTE	STRING	GLOBAL	0		“Olá”
3	Px	VAR	FLOAT	LOCAL	0	REF	
4	Py	VAR	INT	LOCAL	0	VALUE	
5	Fat	FUNCTION	INT	GLOBAL	2		
6	Main	SUB		GLOBAL	1		

FONTE: o próprio autor

Nas colunas são representados alguns atributos dos identificadores, como:

- ID, que é o nome que a variável recebeu no programa;
- Categoria, que representa a categoria do identificador, que pode ser uma constante, uma variável, uma função e outras declarações que possam existir na linguagem definida;
- O Tipo do identificador também é armazenado, utilizado principalmente para a verificação de tipos;
- O Escopo que o identificador pertence, que pode ser global ou local;
- O Num. Params, que é a quantidade de parâmetros que a função ou procedimento aceitam;
- Pass by, que identifica se esses parâmetros foram passados por valor ou referência;
- Valor, que é o valor armazenado por uma constante.

Cada linha é a inclusão de um novo identificador no programa.

3.4.3 Tipos de dados e verificação de tipos

Antes do compilador gerar um código executável para a máquina alvo, ele precisa garantir que cada parte deste programa faça sentido para as regras de tipos da linguagem. Para que isso seja possível, o compilador necessita manter as informações sobre os tipos de dados. As informações sobre os tipos de dados podem ser estáticas, dinâmicas ou uma mistura das duas. Serão utilizados apenas os tipos de dados estáticos neste trabalho, que são utilizados por linguagens mais tradicionais, como o C. Essas informações também são utilizadas para determinar a quantidade de memória requerida para alocação de cada variável e a forma de acesso à memória. Essas informações são armazenadas na tabela de símbolos. (LOUDEN, 2004; COOPER, 2012)

Linguagens de programação modernas armazenam diversos tipos de dados, incluindo números, caracteres, booleanos, ponteiros para objetos e outros. O compilador deve manipular esses tipos de dados. Para que isso ocorra de uma forma segura, o compilador precisa saber qual o tamanho da representação deste dado na máquina alvo. Caso seja um número, o tamanho dele é de 16, 32 ou 64 bits? Caso seja um conjunto de caracteres, o tamanho será fixo no momento da compilação ou será determinado em

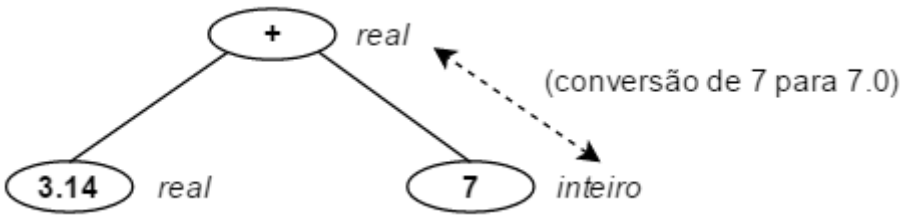
tempo de execução? Estaticamente, essas questões podem ser respondidas examinando as declarações das variáveis. (COOPER, 2012)

A verificação de tipos realiza uma análise mais profunda. Com as informações já inseridas na tabela de símbolos, é possível verificar se o valor atribuído a uma variável equivale a estrutura a qual ela foi definida em sua declaração. No caso de acontecer uma atribuição incompatível, é retornado um erro de incompatibilidade de tipos.

Essas validações se estendem a outras estruturas existentes nas linguagens de programação. Como por exemplo, o retorno de uma função, onde o tipo retornado da função deve ser compatível com a variável ou expressão que o retorno está sendo atribuído.

Nem sempre uma diferença entre o tipo esperado e o tipo atribuído correspondem a uma violação que resulte em erro. Um exemplo comum, é a mistura de tipos numéricos em expressões aritméticas, como `3.14 + 7`, no qual um número real e um inteiro são somados. Nesses casos, muitos compiladores definem uma hierarquia entre os tipos, de modo que seja realizada uma conversão de tipo no objeto de menor hierarquia, de modo que a operação possa ser realizada no tipo de maior hierarquia. No caso de `3.14 + 7`, o tipo real é de maior hierarquia que o tipo inteiro, ou seja, o número `7` deve ser convertido para o tipo real antes de efetuar a soma com o valor `3.14`. Essa conversão automática é denominada coação, a qual pode ser expressa implicitamente pelo verificador de tipos. A Figura 11 demostra a alteração do tipo de uma expressão para o tipo de uma sub expressão, na qual a diferença de tipos implica na conversão de `7` para `7.0`. (NETO, 1987; LOUDEN, 2004)

FIGURA 11 – CONVERSÃO DE TIPOS



FONTE: LOUDEN, 2004; pg. 335

3.5 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Uma tarefa fundamental do compilador é a tradução propriamente dita do programa-fonte para a forma de código intermediário. Normalmente, a geração de código intermediário vem acompanhada das ações da análise semântica. (NETO, 1987)

Nesta etapa, ocorre a conversão de instruções específicas da linguagem fonte para um conjunto de conceitos mais gerais para que possam ser expressados em código de máquina. Isto não quer dizer que é preciso converter diretamente um código com abstração sintática de alto nível para código de máquina. Muitos compiladores utilizam uma linguagem intermediária como um passo entre a linguagem de alto nível e o código de máquina, que é de baixo nível. Em geral, o código intermediário consiste praticamente em expressões e instruções de fluxo de controle. (MOGENSEN, 2011; GRUNE, 2001)

A grande diferença entre o código intermediário e o código objeto final é que o código intermediário não especifica detalhes da máquina alvo, como por exemplo, quais registradores serão utilizados e quais endereços de memória serão referenciados. (PRICE, 2001)

Além de estruturar o compilador em tarefas menores e ter a possibilidade de otimizar o código intermediário, de modo a obter um código objeto final mais eficiente, usar uma linguagem intermediária tem outras vantagens. Caso seja necessário gerar código objeto para diferentes arquiteturas de computadores, apenas uma conversão para código intermediário é necessário. Somente a conversão de código intermediário para o código de máquina é que precisará ser reescrito para cada versão de arquitetura.

Outra vantagem é que é possível utilizar o mesmo código intermediário para diversas linguagens de alto nível que precisam ser compiladas. Todas elas poderão compartilhar a tradução do código intermediário para código de máquina.

Com essas vantagens, é possível construir um programa que possa ser distribuído para todas as arquiteturas, desde que a máquina esteja equipada com um interpretador para qual o programa foi construído, sem a necessidade de precisar criar binários para cada uma dessas arquiteturas.

Uma das desvantagens de utilizar um interpretador intermediário é que na maioria dos casos será mais lento do que executar um código que foi traduzido diretamente para

o código de máquina. No entanto, esta abordagem teve vários sucessos, como por exemplo, o Java. (MOGENSEN, 2011)

3.5.1 Linguagens intermediárias

Existem diversos tipos de códigos intermediários e todos eles pertencem a alguma categoria de representação. As categorias de representação podem ser de 3 tipos:

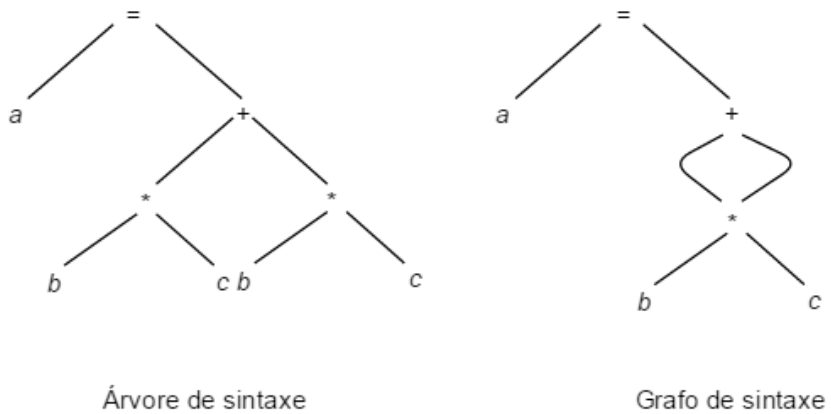
- representações gráficas: árvore e grafo de sintaxe;
- notação pré-fixada e pós-fixada;
- código de três endereços: quádruplas e triplas

3.5.1.1 Representações gráficas

A representação gráfica de uma árvore de sintaxe é uma forma condensada de árvore de derivação na qual os operandos da linguagem constituem os nós interiores da árvore. Outra simplificação da árvore de sintaxe é que cadeias de produções simples, por exemplo, A→B, B→C, são eliminadas. (PRICE, 2001)

A representação por grafo de sintaxe, além de possuir as mesmas informações da árvore de sintaxe, faz a fatoração das sub expressões idênticas, eliminando-as. As duas representações gráficas aparecem na Figura 12, para o seguinte enunciado de atribuição em C: `a = b * c + b * c`

FIGURA 12 – ÁRVORE E GRAFO DE SINTAXE



FONTE: AHO, 1995; pg. 201

3.5.1.2 Notações pré-fixadas e pós-fixadas

As notações pré-fixada e pós-fixadas são representações lineares de uma árvore sintática. A linguagem *Assembly* é um exemplo de código linear. A notação consiste em uma sequência de instruções que são executadas conforme os nós aparecem e do número de operandos que o operador espera receber.

A lógica por trás de uma forma linear é simples. O código fonte que serve de entrada para o compilador é linear, como também é o código de máquina. Muitos dos primeiros compiladores utilizaram a representação linear pois era uma notação natural para os autores, uma vez que já tinham prática de programar em *Assembly*. (COOPER, 2012)

Para avaliação de expressões pré-fixadas, pode-se utilizar uma pilha e executar um processo que leia a expressão da direita para a esquerda, empilhando cada operando até encontrar um operador. O operador então é aplicado a todos os operando que estão no topo da pilha. Um processo semelhante pode ser executado para a avaliação de expressões pós-fixadas; neste caso, a expressão é lida da esquerda para à direita. (PRICE, 2001)

Alguns exemplos das representações lineares de notações pré-fixadas e pós-fixadas são mostrados na Tabela 7.

TABELA 7 – NOTAÇÕES PRÉ E PÓS FIXADAS

Notação		
Infixada	Pré-fixada	Pós-fixada
(b * c) + (b * c)	+*bc*bc	bc*bc*+
a * (b + c)	*a+bc	abc+*
a + b * c	+a*bc	abc*+

FONTE: o próprio autor

3.5.1.3 Código de Três Endereços

Na representação de código de três endereços, cada instrução faz referência a no máximo três endereços de memória. Conforme o enunciado da forma geral:

$$x = y \; op \; z$$

no qual x, y e z são variáveis, constantes ou objetos de dados temporários criados pelo compilador; *op* está no lugar de qualquer operador. É possível notar que nesta representação não são permitidas expressões aritméticas construídas, na medida que só há um operador no lado direito de um enunciado. Desta forma, para obter-se a representação do enunciado `a = x + y * z` para código de três endereços, é possível realizar a tradução da seguinte forma:

$$t_1 = y * z$$

$$t_2 = x + t_1$$

$$a = t_2$$

onde t_1 e t_2 são variáveis temporárias. (AHO, 1995)

Um código de três endereços pode ser representado através de quádruplas ou triplas. As quadruplas são constituídas de quatro campos: um operador, dois operando e o resultado. As triplas são formadas por: um operador e dois operando, essa representação utiliza ponteiros para a própria estrutura, evitando a nomeação de variáveis temporárias. (PRICE, 2001)

3.6 AMBIENTES DE EXECUÇÃO E GERAÇÃO DE CÓDIGO

Até aqui foram discutidas as fases de um compilador que dependem apenas das propriedades da linguagem-fonte, que são independentes da linguagem-alvo (de máquina) e das propriedades da máquina-alvo e de seu sistema operacional.

No presente capítulo, será considerada a tarefa final de um compilador, que é gerar o código executável para uma máquina-alvo, o qual deve ser uma representação fiel da semântica do código fonte. A geração de código executável pode requerer uma análise adicional, como efetuar uma otimização de código, que será o assunto da próxima seção. (LOUDEN, 2004)

As características gerais para a geração de código são as mesmas para uma ampla variedade de arquiteturas. O ambiente de execução é a estrutura de registros e de memória do computador-alvo. Isto se refere ao gerenciamento de memória, no qual o ambiente de execução deve ser o responsável pelo gerenciamento da área disponível de memória para o armazenamento dos dados, a comunicação com o sistema operacional e

com outros programas, e a execução de funções especiais, representadas por pacotes de rotinas, da biblioteca do sistema ou de aplicação. (NETO, 1987)

Existem três tipos de ambientes de execução, cuja estrutura básica não depende da arquitetura da máquina-alvo. Esses três tipos são: ambiente totalmente estático, ambiente baseado em pilhas e ambiente totalmente dinâmico. Praticamente todas as linguagens de programação utilizam um dentre esses três tipos de ambientes de execução.

Nesta seção, será apresentada a organização da memória durante a execução de programas, a estrutura geral de cada um dos três tipos de ambientes e ao fim do capítulo, é apresentada a MEPA (Máquina de Execução para Pascal) (KOWALTOWSKI, 1983), que é um ambiente baseado em pilhas.

3.6.1 Organização de memória durante a execução de programas

A memória, durante a execução de um programa, é dividida em duas áreas. Uma área para armazenar o código do programa, chamada de área de código e outra área para armazenar os dados do usuário, chamada de área de dados. Normalmente, a área de código não muda durante a execução do programa. A área de dados, armazena as constantes globais e estáticas e também os literais na área global/estática.

Além da área de dados global, existem as variáveis locais, as quais podem ser alteradas durante a execução do programa. Existem diversas formas de se organizar a memória para alocação de dados dinâmicos. Uma organização típica é a divisão da memória em uma área de pilha e uma área de *heap*. A memória para um ambiente de execução pode ser dividida da seguinte forma:

- Área de código, para armazenar o código de destino
- Área global/estática, para armazenar variáveis globais ou literais
- Pilha, usada para dados cuja alocação ocorre de forma LIFO, do inglês *last in, first out*, que quer dizer, último a entrar, primeiro a sair.
- *Heap*, que não tem relação com a estrutura de dados *heap* usada em algoritmos, utilizada para alocação dinâmica que não obedece ao protocolo LIFO, por exemplo, alocação de ponteiros em C.

Uma organização geral do armazenamento em um ambiente de execução, que tem todas as categorias de memórias descritas acima, é apresentada na Figura 13.

FIGURA 13 – ARMAZENAMENTO EM UM AMBIENTE DE EXECUÇÃO

FONTE: LOUDEN, 2004; pg. 351

3.6.1.1 Registro de ativação de procedimentos

O registo de ativação de procedimentos é uma unidade de alocação de memória que contém memória alocada para os dados locais de um procedimento ou função na medida que é ativado. Quando a função é chamada, o registro de ativação para essa função é criado e armazenado em uma pilha. Os componentes para o registro de ativação são:

- Valor de retorno – Valor do retorno é utilizado para armazenar o valor que a função retorna após a sua execução;
- Ponteiro de argumentos – que aponta para área do registro de ativação reservada para argumentos (valores de parâmetros), que são entradas para uma função, enviados no momento que a função é chamada;
- Ponteiro de quadros – aponta para o registro de ativação corrente, muito útil em recursividade;

- Estado de máquina – consiste em valores do contador do programa, registros da máquina, etc;
- Dados locais – armazena os dados locais da função chamada;
- Temporários - utilizados para armazenar os resultados intermediários durante a execução de expressões grandes.

O ambiente de tempo de execução determina a sequência de operações que devem ser executadas quando uma função é chamada, essas operações são identificadas como sequência de ativação. As operações adicionais que ocorrem durante o retorno de um procedimento ou função são consideradas parte da sequência de ativação, mas podem ser identificadas como sequência de retorno. O ativador é responsável por computar os argumentos e colocá-los no registro de ativação durante a sequência de chamada. O ativado tem que cuidar do controle e acessar ponteiros, juntamente com os temporários e dados locais. Qualquer informação de registro adicional deve também ser ajustada e pode ser feita em cooperação entre ativador e ativado. (LOUDEN, 2004)

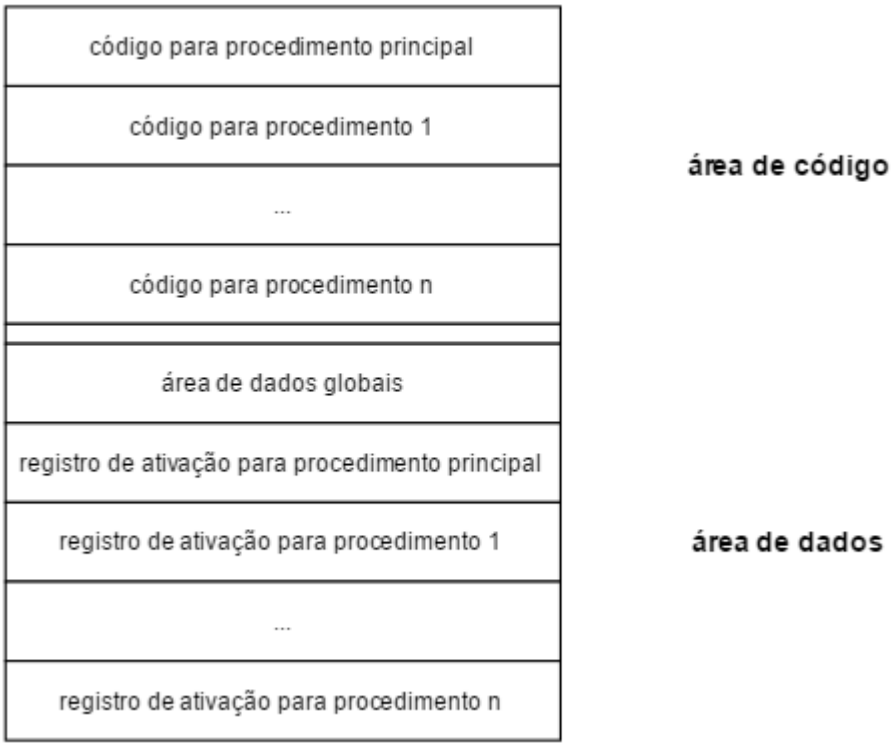
3.6.2 Ambiente Totalmente Estático

Em um ambiente totalmente estático, os dados são armazenados em locais fixos da memória durante a execução do programa. A linguagem que utiliza um ambiente estático não terá o conceito de ponteiros, de recursividade e o de alocação dinâmica de memória. Todas as variáveis e o local dos registros de ativação são alocadas estaticamente antes da execução. Este ambiente não requer nenhum suporte em tempo de execução pois os nomes estão vinculados ao armazenamento em tempo de compilação. O exemplo de linguagem com essas características é o FORTRAN77.

A organização de memória para um ambiente estático é mostrada na Figura 14. A área de código consiste no código para o procedimento principal, seguido pelo código do procedimento 1 até o procedimento n. Após a área de código, a área de dados contém a área de dados globais, seguido pelo registro de ativação para o procedimento principal e os procedimentos de 1 a n. Quando um procedimento é ativado, cada argumento é armazenado no registro de ativação e o valor de retorno é gravado, e ocorre um salto

para o começo do código do procedimento ativado. No retorno, um salto simples é efetuado para o endereço de retorno. (LOUDEN, 2004)

FIGURA 14 – ORGANIZAÇÃO DE MEMÓRIA EM UM AMBIENTE ESTÁTICO



FONTE: LOUNDE, 2004; pg. 354

3.6.3 Ambiente Baseado em Pilhas

Uma das grandes limitações do ambiente totalmente estático é a impossibilidade de se ter procedimentos recursivos. Procedimentos recursivos necessitam de mais de um registro de ativação criado no momento de sua chamada, sendo que em dado momento, podem existir na memória vários registros de ativação de um mesmo procedimento. A cada ativação, variáveis locais desses procedimentos recebem novas posições de memória. A solução é um ambiente baseado em pilha.

A pilha de execução, também chamada de pilha de ativação ou pilha de registros de ativação, aumenta ou diminui conforme as ativações ocorridas no programa em execução. Cada novo registro de ativação é colocado no topo da pilha quando uma ativação de procedimento ocorrer, quando esse procedimento termina, os dados correspondentes são desempilhados. Esse ambiente requer uma estratégia de controle e

acesso a variáveis mais complexa que um ambiente totalmente estático. (LOUDEN, 2004)

A quantidade de informação requerida por um ambiente baseado em pilha depende fortemente das propriedades da linguagem compilada. A organização dos ambientes baseados em pilha podem ser três: ambiente baseado em pilha sem procedimento local; ambiente baseado em pilha com procedimentos locais e ambiente baseado em pilha com procedimentos como parâmetros. C e Pascal são exemplos de linguagens que utilizam um ambiente baseado em pilha.

Ambientes baseados em pilha requerem três ponteiros:

- Ponteiro de quadro (fp) – Aponta para o registro de ativação corrente para acessar as variáveis locais;
- Ponteiro de quadro velho (fp velho) – Representa o valor anterior do fp;
- Ponteiro de pilha (sp) – Sempre aponta para o topo da pilha.

Como exemplo, será considerada a implementação recursiva simples do algoritmo de Euclides para computar o máximo divisor comum (MDC) de dois inteiros positivos (15 e 10), cujo código em C é dado na Figura 15.

FIGURA 15 – ALGORITMO DE EUCLIDES

```
1  #include <stdio.h>
2
3  int mdc(int a, int b){
4      if (b == 0) return a;
5      else return mdc(b, a%b) ;
6  }
7
8  void main( ){
9      int x, y;
10     scanf (“%d%d”, & x, & y);
11     printf (“MDC de %d e %d = %d”, x, y, mdc(x,y));
12 }
```

FONTE: LOUNDE, 2004; pg. 357

Supondo-se que o usuário queira saber o MDC de 15 e 10 e insira estes valores ao programa, a função `main()` então faz a ativação de `mdc(15,10)`. Como `b < > 0`, uma ativação recursiva é feita, `mdc(10,5)`, pois `15 % 10 = 5`, e isso resulta em

uma terceira ativação `mdc(5,0)`, pois `10 % 5 = 0`, que então retorna o valor `5`. No momento da terceira ativação o conteúdo da pilha pode ser visualizado na Figura 16.

FIGURA 16 – AMBIENTE BASEADO EM PILHA PARA O ALGORITMO DE EUCLIDES

FONTE: o próprio autor

3.6.4 Ambiente Totalmente Dinâmico

Ambientes totalmente dinâmicos devem permitir manter na memória registros de ativação de procedimentos que já foram terminados enquanto for necessário. Em um ambiente baseado em pilha, na qual uma referência para uma variável local é retornada, ocorre uma referência inválida quando a função termina, pois, o registro de ativação desta função foi removida da pilha. Um exemplo simples para esse caso é o retorno do endereço de uma variável local, como mostrado no código em C na Figura 17.

FIGURA 17 – CÓDIGO QUE RETORNA UMA REFERÊNCIA QUE SERÁ INVÁLIDA

```
1  int* ref_invalida(){
2          int x;
3          return &x;
4  }
```

FONTE: LOUNDE, 2004. Pg. 378

Um ambiente totalmente dinâmico é significativamente mais complexo que um ambiente baseado em pilha, pois é preciso rastrear as referências durante a execução, e a habilidade de liberar áreas inacessíveis da memória em tempos arbitrários durante a execução do programa, este processo é conhecido como *garbage collection*.

A estrutura básica do registro de ativação continua a mesma, isto é, alocando espaço para parâmetros e variáveis locais, e ainda tem a necessidade de existir a vinculação de controle e de acesso. A única diferença é a liberação de espaço que ocorre em um tempo arbitrário após o retorno do procedimento.

3.6.5 Máquina de Execução Para Pascal (MEPA)

Conforme analisado nas seções anteriores, traduzir um programa fonte para uma linguagem de máquina é uma tarefa árdua. Cada arquitetura possui suas próprias características e suas linguagens de máquina precisam se preocupar com cada uma delas, o que para nós, seria uma perda de foco se entrarmos em cada um desses detalhes. Então, ao invés de se traduzir os programas em Chimera para linguagem de máquina diretamente, será utilizada uma máquina virtual. A máquina virtual é um simulador, que se comporta da mesma forma que uma máquina real se comportaria.

A MEPA é uma máquina de pilha, característica muito apropriada para tradução de estruturas encontradas em linguagens de programação *Pascal-Like* (Derivadas da linguagem Pascal) e *C-Like* (derivadas da linguagem C), como estruturas aninhadas, recursividade, etc. (OLIVEIRA, 2014)

A MEPA possui uma memória composta em três regiões (KOWALTOWSKI, 1983):

- A região de programa (P), que contém as instruções da MEPA a serem executadas;
- A região da pilha de dados (M), que contém os valores manipulados pelas instruções da MEPA;
- O vetor de registradores de base (D), que contém os ponteiros (endereços) para a região M, ou então valores indefinidos. Sendo útil para os casos de mudança de escopo de um programa.

Além dessas três regiões de memória, a MEPA possui dois registradores especiais que são utilizados para descrever o efeito das instruções:

- O registrador de programa *i*, é um contador do programa, que contém o endereço da próxima instrução a ser executada, denotada P[*i*];
- O registrador *s*, que indica o elemento do topo da pilha, sendo o valor do elemento descrito como M[*s*].

Após o programa MEPA ser carregado na região P e os registradores terem sido inicializados, o funcionamento da MEPA é bem simples. As instruções apontadas pelo registrador *i* são executadas até que seja encontrada uma instrução de parada, ou um erro ocorra. Então, com exceção das instruções que envolvem desvios, o contador *i* é incrementado e passa a apontar para a próxima instrução do vetor P. O algoritmo de funcionamento da MEPA pode ser visto na Figura 18.

FIGURA 18 – ALGORITMO DE FUNCIONAMENTO DA MEPA

```
Enquanto (i válido and (not FIM) ) do
   Executa P[i]
   Se P[i] indica desvio
      então i := endereço do desvio
      senão i := i + 1
   FimSe
FimEnquanto
```

FONTE: KOWALTOWSKI, 1983

3.6.5.1 A linguagem da MEPA

Como a MEPA tem a função de gerenciar a memória e os registradores da máquina, é preciso se preocupar apenas com a linguagem intermediária criada para a MEPA e ela irá gerar o código objeto se preocupando com os detalhes da máquina de destino.

A linguagem da MEPA foi definida com instruções de palavras de quatro letras e a sua versão básica é apresentada na Tabela 8. As notações utilizadas para os parâmetros foram:

- c valor armazenado na pilha

- k nível léxico. Endereço relativo a memória
- n endereço de memória
- m valor incrementado ou decrementado em s
- p endereço de linha de código

TABELA 8 – VERSÃO BÁSICA DA LINGUAGEM MEPA

Instrução	Execução	Significado
Armazenamentos e E/S		
CRCT c	s:=s+1; M[s]:=c;	Carrega Constante
CRVL k, n	s:=s+1; M[s]:=M[D[k]+n];	Carrega Valor
CREN	s:=s+1; M[s] := D[m]+n;	Carrega Endereço. Usado para parâmetros passados por referência
CRVI	s:=s+1; M[s] := M[M[D[m]+n]];	Carrega Valor Indireto
ARMZ k, n	M[D[k]+n]:=M[s]; s:=s-1;	Armazena Valor em Variável Local
ARMI k, n	M[M[D[m]+n]] := M[s] ; s := s+1;	Armazena Indireto
LEIT	s:=s+1; M[s]:= stdin;	Leitura
IMPR	stdout:=M[s]; s:=s-1;	Imprime
Expressões		
SOMA	M[s-1]:=M[s-1]+M[s]; s:=s-1;	Soma
SUBT	M[s-1]:=M[s-1]-M[s]; s:=s-1;	Subtração
MULT	M[s-1]:=M[s-1]*M[s]; s:=s-1;	Multiplicação
DIVI	M[s-1]:=M[s-1]/M[s]; s:=s-1;	Divisão
MODI	M[s-1]:=M[s-1]%M[s]; s:=s-1;	Resto da divisão
CONJ	M[s-1]:=M[s-1] and M[s]; s:=s-1;	Conjunção (and)
DISJ	M[s-1]:=M[s-1] or M[s]; s:=s-1;	Disjunção (or)
NEGA	M[s]:= 1 - M[s];	Negação (not)
CMME	Se M[s-1] < M[s] **então** M[s-1]:=1 **senão** M[s-1]:=0; s:=s-1;	Compara se Menor
CMMA	Similar CMME	Compara se Maior
CMIG	Similar CMME	Compara se Igual
CMDG	Similar CMME	Compara se Desigual
CMAG	Similar CMME	Compara se Maior ou Igual
CMEG	Similar CMME	Compara se Menor ou Igual
INRV	M[s]:=-M[s];	Inverte Sinal
Desvios		
DSVF p	Se M[s]=0 **então** i:=p **senão** i:=i+1; s:=s-1;	Desvia se Falso
DSVS p	i:=p;	Desvia sempre

NADA		Nada
Procedimentos		
CHPR p, k	M[s+1]:=i+1; M[s+2]:=k; s:=s+2; i:=p;	Chama Procedimento
ENPR k	s:=s+1; M[s]:=k; D[k]:=s+1;	Entra no Procedimento
RTPR k, n	D[k]:=M[s]; i:=M[s-2]; s:=s-(n+3);	Retorna do Procedimento
Programa e Alocação de Memória		
INPP	s:=-1; D[0]:=0;	Inicia Programa Principal
PARA		Volta ao SO
AMEM m	s:=s+m;	Aloca Memória
DMEM m	s:=s-m;	Desaloca Memória

FONTE: KOWALTOWSKI, 1983

3.7 OTIMIZAÇÃO DE CÓDIGO

Otimização de código é uma técnica utilizada em compiladores que tem como objetivo melhorar a performance do código-objeto produzido pelo compilador, consumindo menos recursos computacionais. Esta melhoria pode ser em velocidade de processamento do programa e/ou no tamanho total do código objeto gerado pelo compilador, ocupando menos espaço em disco.

Primeiro, é preciso deixar claro que “otimização” pode ser um termo equivocado. Raramente a aplicação de otimização em um código resulte em um desempenho ótimo.

Um processo de otimização de código deve seguir algumas regras para que seja eficiente, as principais delas são (NETO, 1987):

- O código-objeto final, de forma alguma, pode modificar o significado do programa;
- A otimização deve aumentar a velocidade do programa e, se possível, deve utilizar menos recursos computacionais;
- A otimização deve ser rápida, sem aumentar significativamente o tempo do processo de compilação geral.

A otimização pode ser realizada em vários níveis dentro de um processo de compilação e podem ser caracterizadas como independentes de máquina, sem se importar com a arquitetura da máquina de destino do código-objeto, ou dependentes de

máquina, as quais devem se preocupar com as características da linguagem de máquina alvo. Os níveis que podem ocorrer a otimização podem ser:

- Na construção do compilador, em que o projetista utiliza técnicas eficientes de programação e melhores algoritmos;
- Após a geração do código intermediário, no qual é possível remover regiões inacessíveis pelo código, eliminar sub expressões equivalentes e realizar outras formas de otimização no código intermediário;
- Após a geração do código-objeto, em que o compilador pode otimizar a performance ao utilizar hierarquia de memória e registradores da CPU.

4 CONCEITOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

A programação orientada a objetos (POO), é uma abordagem para o desenvolvimento de software em que a estrutura do software é baseada em objetos que interagem uns com os outros para realizar uma determinada tarefa. Essa interação ocorre em forma de mensagens que são passadas entre os objetos. Em resposta a uma mensagem, um objeto pode executar uma ação.

Se for analisada a forma como executamos nossas tarefas diárias no mundo real, é possível identificar que interagimos em um mundo orientado a objetos. Em uma entrevista concedida à revista *Rolling Stone*, Steve Jobs foi questionado pelo entrevistador Jeff Goodell a dar uma explicação simples sobre POO, a resposta de Steve foi clara e de fácil entendimento para qualquer leigo compreender os conceitos de POO. (JOBS, 2011)

> ***Jeff Goodell:*** *Você poderia explicar, simplificadamente, exatamente o que é um software orientado a objetos?*
>
> ***Steve Jobs:*** *Objetos são como pessoas. Eles vivem, respiram coisas que têm conhecimento, sabem como fazer essas coisas e tem memória dentro deles para que possam lembrar dessas coisas. E, ao invés de interagir com eles em um nível muito baixo, você interage com eles em um nível muito alto de abstração, como estamos fazendo aqui.*
> *Vejamos um exemplo: Se eu sou seu objeto de lavanderia, você pode me dar suas roupas sujas e me enviar uma mensagem que diz: "Você pode pegar minhas roupas lavadas, por favor? " Acontece que eu sei aonde fica a melhor lavanderia de São Francisco. E eu falo inglês, e tenho dólares em meu bolso. Então, eu saio e pego um táxi e digo ao motorista para me levar a este lugar em São Francisco. Vou pegar suas roupas lavadas, salto de volta para o taxi e volto aqui. Eu lhe dou suas roupas limpas e digo: "Aqui estão suas roupas limpas. "*

> *Você não tem ideia de como eu fiz isso. Você não sabe onde a lavanderia fica. Talvez você fale francês e não consiga chamar um táxi. Você não pode nem pagar por um, pois você não tem dólares em seu bolso. No entanto, eu sabia como fazer tudo isso. E você, não precisava saber nada disso. Toda essa complexidade estava escondida dentro de mim, e fomos capazes de interagir em um nível muito alto de abstração. Isso é o que os objetos são. Eles encapsulam a complexidade e as interfaces para essa complexidade são de alto nível.*

Programação orientada a objetos gira em torno de alguns conceitos chaves: classes, objetos, propriedades, métodos, herança e encapsulamento. Todos estes conceitos são importantes para POO e o objetivo deste capítulo é tratar com mais detalhes cada um deles.

4.1 CLASSES E OBJETOS

Uma classe é um modelo do qual pode-se criar objetos. A definição de uma classe inclui as especificações formais para a classe e quaisquer dados ou métodos contidos nela. Ou seja, a classe é o tipo do objeto.

Um objeto é a instância de uma classe, da mesma maneira que uma variável é a instância de um tipo de dado. Uma classe é um tipo do objeto. É possível criar diferentes objetos usando uma mesma classe, porque uma classe é apenas um modelo, enquanto objetos são instâncias concretas, baseados em um modelo. Objetos encapsulam métodos e instanciam variáveis.

Um exemplo prático de objeto é um gato. Podemos dizer que o gato possui certas características como cor, nome e peso. Ele também pode desenvolver atividades como miar, dormir, esconder e escapar. As características do objeto em POO são chamados de propriedades ou atributos e as ações são chamados de métodos.

Existe uma analogia com nossa linguagem falada:

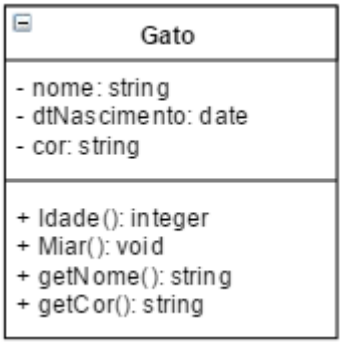
- Objetos são nomeados normalmente com substantivos (Livro, Animal)

- Métodos são verbos (Ler, Correr, Dormir)
- Valores das propriedades são adjetivos

Por exemplo, na sentença “O gato dourado está dormindo”. “O gato” (substantivo) é o objeto, “dourado” (adjetivo) é o valor da propriedade *cor*, e “dormindo” (verbo) é uma ação, ou um método em POO. (STEFANOV, 2008)

A Figura 19 mostra o diagrama UML de uma classe, com atributos e métodos.

FIGURA 19 – CLASSE COM ATRIBUTOS E MÉTODOS



FONTE: (DENNIS et al., 2009)

4.2 ENCAPSULAMENTO

Encapsulamento é ocultar e proteger suas informações dentro de objetos. De fato, objetos e classes, são geralmente definidos em torno de dados.

É possível controlar o acesso aos dados ou aos métodos de uma classe tornando os seus membros:

- *privados*, visíveis apenas para código dentro da mesma classe.
- *protegidos*, visíveis para a mesma classe e as classes que são derivadas desta classe.
- *públicos*, visíveis para código dentro e fora da classe.

O encapsulamento é uma parte crucial da POO, quando ocultamos dados dentro de um objeto, podemos prover métodos de acesso que permitam códigos fora deste objeto acessar estes dados. Desta forma, é possível controlar o acesso a esses dados,

como por exemplo, não permitir dados que o objeto considere ilegal ser armazenado. Normalmente estes métodos de acesso a dados são implementados em pares, como métodos "*get*" e "*set*", no qual o método *get* obtém o valor do dado, e o método *set* atribui um valor para este mesmo dado. (HOLZNER, 2001)

4.3 HERANÇA

Herança é o processo de derivação de uma classe, chamada classe derivada, de uma outra classe, a classe base, sendo capaz de fazer o uso dos atributos e métodos da classe base na classe derivada.

Herança de classe permite o reuso de uma classe derivando uma nova classe a partir de outra classe existente e sendo possível customizar esta nova classe da maneira que lhe convém. (HOLZNER, 2001)

Por exemplo, ao criar-se uma classe chamada `Animal`. Esta classe suporta apenas três métodos, `Comer`, `Dormir` e `Respirar`. Os métodos desta classe são comuns para todos os animais, por exemplo, é possível criar uma instância de um objeto desta classe chamada `Gato`, outra `Passaro` e outra `Elefante`, todos esses animais podem comer, dormir e respirar.

Utilizando apenas a classe `Animal`, não é possível definir a diferença entre um gato e um canário, por exemplo, e com o tempo, utilizar apenas esta classe não fará sentido. Se quisermos continuar a utilizar a classe `Animal`, provavelmente será preciso customizar esta classe para obter comportamentos mais específicos para cada animal.

Uma maneira de customizar esta classe para diversos animais é copiá-la e criar dúzias delas, uma específica para cada animal. Porém, se o código dentro da classe `Animal` é longo, ou se ela contém dados internos importantes, talvez não seja uma boa ideia criar diversas classes separadas para cada tipo de animal. Caso queira-se alterar o código de um método, será preciso alterá-lo em todas as classes separadamente.

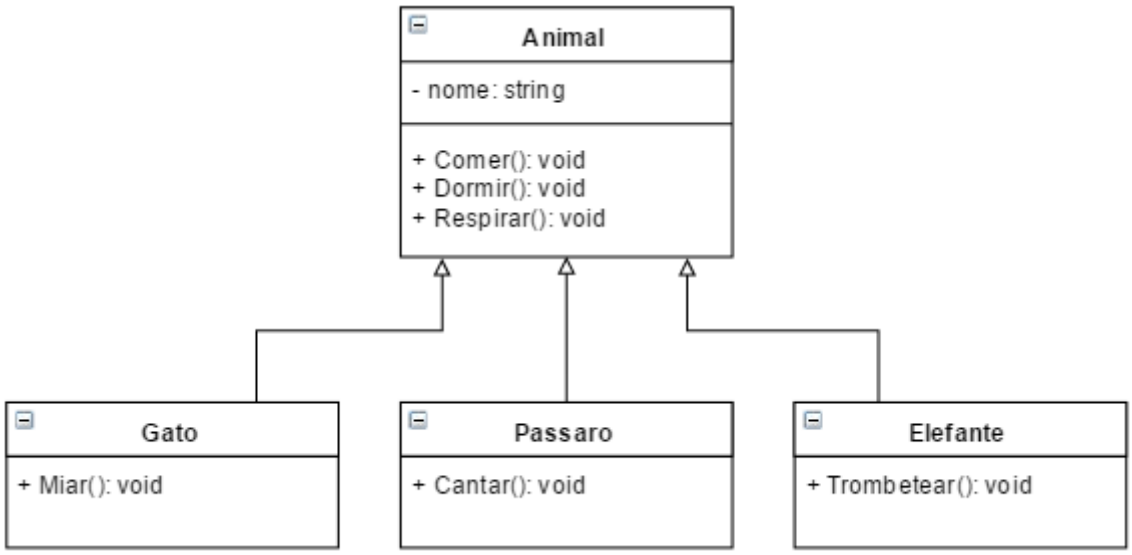
Ao invés de se optar pela escolha acima, é possível definir o que os animais têm em comum e colocar esses métodos e atributos em uma classe base. Neste caso, a classe `Animal`, que contém os métodos `Comer`, `Dormir` e `Respirar`, que são comuns para todos os animais, será a classe base. Utilizando herança, podemos derivar novas classes

a partir desta classe base, e estas classes derivadas podem ser atribuídas para cada animal específico, contendo suas próprias características, enquanto continuam com as funcionalidades de comer, dormir e respirar herdadas da classe base.

Na Figura 20, temos um exemplo de herança. Todas as classes derivadas da classe base `Animal` tem métodos próprios e também compartilham os métodos em comum da classe base.

Em UML, para herança, utilizamos a seta partindo da classe derivada apontando para a classe base com o triângulo sem preenchimento, conforme o exemplo da Figura 20. (DENNIS, 2009)

FIGURA 20 – EXEMPLO DE HERANÇA



FONTE: o próprio autor

Seguindo com o exemplo acima, é possível derivar uma classe `Cachorro` da classe `Animal`, e também adicionar o método `Latir` e `Abanar_rabo`. Sendo assim, a classe `Cachorro` terá os métodos `Comer`, `Dormir`, `Respirar`, `Latir` e `Abanar_rabo`, mas tudo o que foi feito foi adicionar o método `Latir` e `Abanar_rabo`, os outros métodos já estarão lá para serem usados pois derivam da classe base `Animal`. A Figura 21 demonstra a implementação deste exemplo em C#.

FIGURA 21 – EXEMPLO DE HERANÇA EM C#

```
using System;
namespace Exemplo
{
    class Animal
    {
        public void Comer(){ Console.WriteLine("Comendo...");}
        public void Dormir(){ Console.WriteLine("Dormindo...");}
        public void Respirar(){ Console.WriteLine("Respirando...");}
    }//Fim da classe Animal

    class Cachorro : Animal
    {
        public void Latir(){ Console.WriteLine("Latindo...");}
        public void Abanar_rabo(){ Console.WriteLine("Abanando o rabo...");}
    }//Fim da classe Cachorro
}
```

FONTE: o próprio autor

4.4 POLIMORFISMO

Polimorfismo é a diversidade de características e comportamentos para objetos semelhantes. Por exemplo, dois irmãos gêmeos idênticos são bem semelhantes, porém, eles possuem características e comportamentos diferentes.

Na programação orientada a objetos classes mais abstratas podem representar o comportamento de classes concretas que a referenciam. Desta forma, é possível tratar vários tipos de objetos de uma forma mais homogênea, através do tipo mais abstrato.

É possível ter quatro tipos de polimorfismo em uma linguagem de programação orientada a objetos, isto não quer dizer que todas as linguagens implementem os quatro tipos. Esta seção é dedicada para tratar destes tipos de polimorfismo. (HOLZNER, 2001)

4.4.1 Sobrecarga de métodos

Um método sobrecarregado pode lidar com diferentes números de argumentos e tipos de argumentos; por exemplo, é possível sobrecarregar o método `Imprimir`, que imprime na tela um texto que foi passado como parâmetro, que pode ser uma cadeia de caracteres, um único caractere, ou até mesmo um objeto que contenha diversas cadeias de caracteres que sejam concatenadas neste método `Imprimir`. Quando é realizada a sobrecarga do método `Imprimir` corretamente, o método pode lidar com todos esses

tipos de dados, e isto faz do método `Imprimir` muito útil e com um propósito mais abrangente. (HOLZNER, 2001)

A Figura 22 mostra um exemplo de método sobrecarregado em C++, de um programa em que é possível imprimir quatro tipos de dados diferentes.

FIGURA 22 – EXEMPLO DE MÉTODO SOBRECARREGADO EM C++

```
1  void Imprimir(char meu_char){cout << meu_char;}
2  void Imprimir(char* minha_string_em_C){cout << minha_string_em_C;}
3  void Imprimir(int meu_codigo_ASC){cout << static_cast<char> (meu_codigo_ASC);}
4  void Imprimir(string minha_string){cout << minha_string;}
```

FONTE: HOLZNER, 2001

Para sobrecarregar um método com diferentes listas de argumentos, é preciso apenas definir o método várias vezes, cada uma com uma lista de argumentos diferentes. Essas listas de argumentos são chamadas de assinaturas. Quando um método é chamado, o compilador verifica os argumentos que foram passados e determina se eles combinam com qualquer assinatura de método que foi definida. Em caso de não encontrar a combinação ou se foram definidas duas assinaturas iguais para o mesmo método, o compilador retornará uma mensagem de erro. (HOLZNER, 2001)

4.4.2 Inclusão

Quando é dito que com polimorfismo por inclusão é possível criar objetos semelhantes que se comportem de maneiras diferente, é dito que cada objeto poderá implementar métodos com assinaturas idênticas, mas que se comportem de maneira diferente ao serem ativados. Por exemplo, é possível dizer que cada animal respira de uma maneira diferente. Então, podemos criar o método virtual `Respirar` na classe base, que se comportará de uma maneira genérica, e criar o mesmo método nas classes derivadas com um comportamento diferente para cada tipo de animal. Caso uma classe derivada de `Animal` não implemente o método `Respirar`, ela irá ter o comportamento genérico descrito na classe base `Animal`. Também é possível criar o método abstrato `Comunicar`, no qual será obrigatório que cada classe derivada implemente este método. Desta forma, cria-se uma coleção de objetos `Animal` e

realiza-se uma iteração dentro desta coleção chamando o método `Comunicar` e o `Respirar`. Para que cada animal obrigatoriamente se comunique de sua própria maneira, é preciso utilizar os conceitos de métodos abstratos.

4.4.2.1 Métodos virtuais e métodos abstratos

Quando uma classe derivada herda um método da classe base ela herda também a sua implementação. Talvez o projetista da classe base queira deixar a classe derivada implementar a sua própria versão deste método, sobrescrevendo o método implementado na classe base. Métodos que podem ser sobrescritos são chamados de métodos virtuais.

Por padrão, na maioria das linguagens de programação orientada à objetos, não podemos simplesmente sobrescrever a implementação de um método da classe base. Para que a sobrescrita do método seja possível, devemos incluir a palavra-chave *virtual* na definição do método na classe base.[1] Este é o motivo de serem chamados métodos virtuais. (CLARK, 2013)

Ainda utilizando o nosso exemplo da classe `Animal` da seção anterior, podemos criar uma nova classe `Peixe` que não implemente o método `Comunicar` e deixar ele ser virtualmente implementado na classe Animal. Desta forma, toda vez que o objeto instanciado da classe `Peixe` chamar o método `Comunicar` o programa irá executar o código implementado na classe `Animal`. Caso um dia este método seja implementado na classe `Peixe`, o que será visto na seção 4.5, o programa irá passar a chamar este método da classe derivada ao invés do da classe base. O método `Comunicar` foi implementado nas outras classes derivadas e toda vez que um desses objetos chamam o método `Comunicar,` é executado o código que foi implementado em suas próprias classes e não o método da classe base. A Figura 23 mostra a implementação dos métodos virtuais `Comunicar` e `Respirar` na classe `Animal` e o comportamento diferenciado em cada uma das classes derivadas, quando implementado.

[1] Em JAVA, por exemplo, todos os métodos que não sejam estáticos são por padrão “métodos virtuais”. Somente os métodos marcados com a palavra-chave **final** e os métodos privados, não são virtuais, portanto não podem ser sobrescritos.

FIGURA 23 – EXEMPLO DE METODOS VIRTUAIS EM C++

```cpp
// classe base
class Animal{
public:
    // métodos virtuais genéricos (caso a classe derivada não implemente)
    virtual void Comunicar() { cout << "Nao sei me comunicar" << endl; }
    virtual void Respirar() { cout << "Respirando..." << endl; }
};
// classe derivada
class Cachorro : public Animal{
public:
    // Sobrescrita para Comunicar
    virtual void Comunicar() { cout << "Latindo...!" << endl; }
};
// classe derivada
class Gato : public Animal{
public:
    // Sobrescrita para Comunicar
    virtual void Comunicar() { cout << "Miando...!" << endl; }
};
// classe derivada
class Peixe : public Animal{
public:
    // Sobrescrita para Respirar
    virtual void Respirar() { cout << "Borbulhando...!" << endl; }
};
```

FONTE: o próprio autor

Ao contrário de métodos virtuais, que tem uma versão implementada na classe base e que podem ou não serem sobrescritos pelas classes derivadas, métodos abstratos não podem sem implementados na classe base e são obrigatoriamente implementados nas classes derivadas. Métodos abstratos são apenas assinaturas de métodos, definindo apenas o tipo do seu retorno e a quantidade de parâmetros que devem obter. Outra diferença é que métodos abstratos devem ser apenas declarados em classes abstratas, ou seja, que não podem ser instanciadas.

4.4.2.2 Classe abstrata

Uma classe abstrata é um tipo de classe especial que não pode ser instanciada por terem métodos abstratos, ela pode ser utilizada apenas como uma classe base, ou seja, como uma referência. No entanto, uma classe abstrata pode ser herdada por suas subclasses. (KOFFMAN, 2008)

Classes abstratas podem ser implementadas parcialmente ou não serem implementadas de nenhuma forma. Assim, quando parcialmente implementadas, além de possuírem métodos abstratos, que devem obrigatoriamente ser implementados por suas classes herdadas e não pela classe abstrata, também possuem métodos que são implementados por ela mesma e que serão utilizados por objetos instanciados de suas classes herdadas. Na Figura 24 temos um exemplo de implementação de uma classe abstrata em Java.

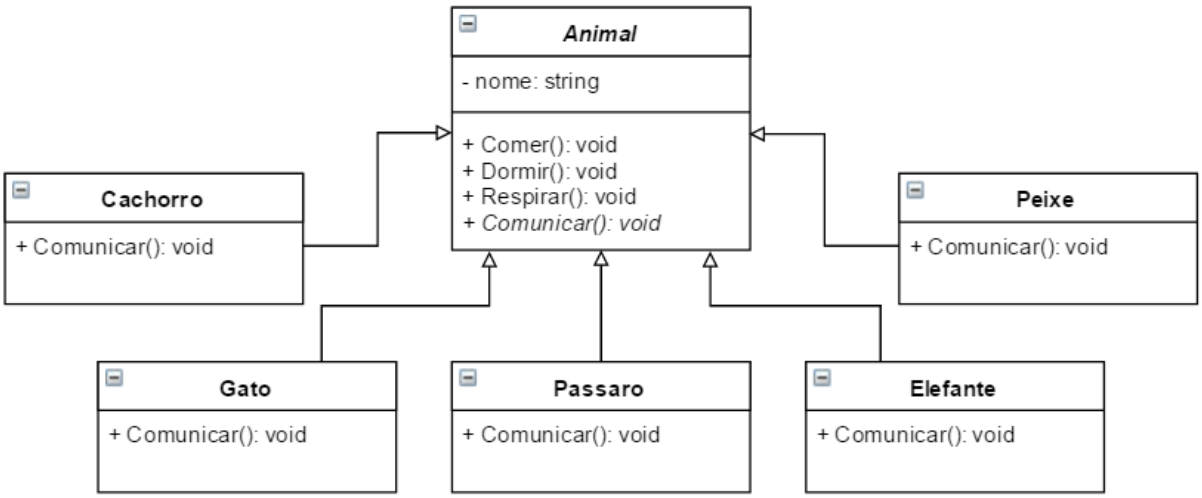
FIGURA 24 – EXEMPLO DE CLASSE ABSTRATA EM JAVA

```
1  public abstract class Animal {
2      //Método abstrato - Classes derivadas obrigatoriamente devem implementá-lo
3      public abstract void Comunicar();
4  }
```

FONTE: o próprio autor

Na Figura 25, é utilizada a classe abstrata `Animal` da Figura 24 a qual foi adicionado os métodos `Comer`, `Dormir` e `Respirar`, que poderão ser utilizados por todas as classes derivadas. As classes derivadas, obrigatoriamente, devem implementar o método `Comunicar`. Em UML, no caso de classe abstrata, o nome da classe deve estar em itálico. (DENNIS, 2009)

FIGURA 25 – CLASSE ABSTRATA



FONTE: o próprio autor

Na Figura 26 temos um programa escrito em C++, que possui o conceito de polimorfismo para o exemplo acima, além disso, o exemplo também contém herança, métodos virtuais, método abstrato, classe abstrata e também o conceito de encapsulamento, com um atributo privado e métodos públicos. Este exemplo conceitua todo nosso aprendizado de POO até aqui. Também se tem o resultado após executarmos o programa proposto.

FIGURA 26 – EXEMPLO DE POLIMORFISMO EM C++

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// classe base abstrata
class Animal{
private:
    string nome;
public:
    Animal(string _nome) { nome = _nome; } //Construtor
    // método abstrato, todas classes derivadas devem implementar
    virtual void Comunicar() = 0; //Este método torna a classe Animal abstrata
    // método virtual, genérico, caso não seja implementado na classe derivada
    virtual void Respirar() { cout << "Respirando..." << endl; }
    // get nome
    string getNome() { return nome; }
    // destrutor virtual
    virtual ~Animal() {}
};
// classe derivada
class Cachorro : public Animal{
public:
    Cachorro(string _nome) : Animal(_nome) {}//Construtor
    // Implementação do método abstrato para Comunicar
    virtual void Comunicar() { cout << "Latindo...!" << endl; }
};
// classe derivada
class Gato : public Animal{
public:
    Gato(string _nome) : Animal(_nome) {}//Construtor
    // Implementação do método abstrato para Comunicar
    virtual void Comunicar() { cout << "Miando...!" << endl; }
};
// classe derivada
class Peixe : public Animal{
public:
    Peixe(string _nome) : Animal(_nome) {}//Construtor
    // Implementação do método abstrato para Comunicar
    virtual void Comunicar() { cout << "Glubglub...!" << endl; }
    // Implementação do método virtual para Respirar
    // sobrescrevendo o método Respirar da classe base
    virtual void Respirar() { cout << "Borbulhando...!" << endl; }
};
```

```
45  //Inicio do Programa
46  int main(int argc, char* args[]){
47      // Coleção de ponteiros para a classe base
48      vector<Animal*> animais;
49      // Alocação dinâmica para as instâncias de classes derivadas
50      // da classe Animal e adiciona à coleção
51      animais.push_back(new Cachorro("Cachorro Rex"));
52      animais.push_back(new Gato("Gato Tobias"));
53      animais.push_back(new Peixe("Peixe Romeo"));
54      //Iteração entre os objetos, cada um interagindo conforme sua cacacterística
55      for (vector<Animal*>::iterator i = animais.begin(); i != animais.end(); ++i){
56          //Imprimo o nome do animal
57          cout << "Nome: " << (*i)->getNome() << endl;
58          //Imprimo como ele está se comunicando
59          cout << "Como se comunica: "; (*i)->Comunicar();
60          //Imprimo como ele Respira
61          cout << "Respira: "; (*i)->Respirar();
62          cout << endl;
63      }
64      // Libero a memória
65      for (vector<Animal*>::iterator i = animais.begin(); i != animais.end(); ++i){
66          delete *i;
67      }
68      animais.clear(); // esvazio a coleção
69      return 0;
70  }
```

```
Resultado:

Nome: Cachorro Rex
Como se comunica: Latindo...!
Respira: Respirando...

Nome: Gato Tobias
Como se comunica: Miando...!
Respira: Respirando...

Nome: Peixe Romeo
Como se comunica: Glubglub...!
Respira: Borbulhando...!
```

FONTE: o próprio autor

4.4.3 Coerção

Polimorfismo por coerção é suportado através da sobrecarga de operadores, ou seja, ocorre quando se converte um elemento de um tipo, no tipo apropriado para o método. Permite que um argumento seja convertido para o tipo esperado por uma função, evitando assim o erro de tipo. É uma operação semântica que reduz o tamanho do código (um compilador determina as conversões de tipo necessárias e as insere automaticamente). Em C++ é suportado através da sobrecarga de operadores, conforme exemplo da Figura 27, no qual é feita a sobrecarga do operador < para ordenação.

FIGURA 27 – EXEMPLO DE COERÇÃO EM C++

```cpp
#include <iostream>
#include<vector>
#include <algorithm>
#include<string>
using namespace std;

class Animal
{
public:
    int chave;
    string nome;

    bool operator < (const Animal& _outro)
    {
        return (this->chave < _outro.chave);
    }
};

void main()
{
    vector<Animal> v_animais;
    Animal dog;
    dog.chave = 3;
    dog.nome = "Bella";
    v_animais.push_back(dog);

    Animal cat;
    cat.chave = 1;
    cat.nome = "Murano";
    v_animais.push_back(cat);

    cout << "Antes da ordenacao:" << endl;
    for (auto it = v_animais.begin(); it != v_animais.end(); it++)
        cout << it->chave << " - " << it->nome << endl;

    //Ordeno o vetor pela função sort,
    //que utilizará a sobrecarga do operador < da classe Animal
    sort(v_animais.begin(), v_animais.end());

    cout << "Apos a ordenacao:" << endl;
    for (auto it = v_animais.begin(); it != v_animais.end(); it++)
        cout << it->chave << " - " << it->nome << endl;
}
```

```
Resultado:
Antes da ordenacao:
3 - Bella
1 - Murano
Apos a ordenacao:
1 - Murano
3 - Bella
```

FONTE: o próprio autor

4.4.4 Paramétrico (Tipos genéricos)

O polimorfismo paramétrico é o fundamento da programação genérica, que envolve escrever código de uma forma que é independente de qualquer tipo de dados em particular. Classes ou funções genéricas podem trabalhar com tipos de dados que serão especificados somente no momento em que a classe ou a função seja instanciada.

Como primeiro exemplo, na Figura 28 é ilustrado um programa que faz a soma de dois números. Ao invés de criar-se uma função para cada tipo de dados, é possível criar um Template, que é o modelo para criar uma classe ou função genérica em C++, que irá realizar a soma para todos os tipos que suportem o operador "+", inclusive *strings*.

FIGURA 28 – EXEMPLO DE UMA FUNÇÃO GENÉRICA EM C++

```
#include <iostream>
#include <string>

using namespace std;
//Função genérica para soma de 2 produtos
template <typename T>
T Soma (T a, T b)  {
   return a + b;
}

int main () {
   int i1 = 100;
   int i2 = 20;
   cout << "Soma dos inteiros: " << Soma(i1, i2) << endl;

   double d1 = 3.14;
   double d2 = 33.7;
   cout << "Soma dos doubles: " << Soma(d1, d2) << endl;

   string s1 = "Oi ";
   string s2 = "Mundo";
   cout << "Soma das string: " << Soma(s1, s2) << endl;

   return 0;
}
```

```
Resultados:

Soma dos inteiros: 120
Soma dos doubles: 36.84
Soma das string: Oi Mundo
```

FONTE: o próprio autor

Na Figura 29, como nosso próximo exemplo, temos uma classe genérica que faz o uso de templates. A classe tem como objetivo implementar uma lista encadeada simples, podendo ser utilizada por vários tipos de dados. Abaixo do protótipo da classe, temos a função `main` implementando dois tipos de listas diferentes, uma para inteiros e outra para *strings*, e logo após, temos o resultado do programa.

FIGURA 29 – EXEMPLO DE UMA CLASSE GENÉRICA EM C++

```
template <class Tipo>
class ListaSimples
{
private:
    struct No
    {
        Tipo valor; //armazena a informação do nó
        No* proximo; //ponteiro para o próximo nó
    };
    No* lista; //apontará para o primeiro nó da lista, ou nulo caso não existam nós
public:

    void Iniciar(); // Iniciar a lista
    void Finalizar(); // Finaliza a lista, liberando a memória
    bool Underflow() const; // Verifica se a lista está vazia
    void Inserir(Tipo); // Insere um item na lista
    bool Remover(Tipo&); // Remove um item da lista
    No* BuscaNo(Tipo); // Busca um nó na lista
};
//Programa
void main(){
    ListaSimples<int> listaInt; //lista de inteiros
    ListaSimples<string> listaString; //lista de strings

    listaInt.Iniciar(); //Iniciar lista de inteiros
    listaString.Iniciar(); //Iniciar lista de strings
    //Lista de inteiros
    listaInt.Inserir(1); //Insererir int 1
    listaInt.Inserir(2); //Insererir int 2
    listaInt.Inserir(3); //Insererir int 3
    //Lista de string
    listaString.Inserir("Ola"); //Insererir string "Ola"
    listaString.Inserir("mundo"); //Insererir string "Mundo"
    //Impressão dos dados da lista
    cout << listaInt.ImprimeListaSimples() << endl; //Imprimir lista de inteiros
    cout << listaString.ImprimeListaSimples() << endl; //Imprimir lista de strings
    //Finalizar a lista
    listaInt.Finalizar();
    listaString.Finalizar();
}
```

```
1 2 3
Ola mundo
```

FONTE: o próprio autor

4.5 INTERFACE

Muitas vezes o conceito de classe abstrata é confundido com interface. Porém, classe abstrata e interface possuem algumas diferenças e devem ser usadas em contextos diferentes.

Uma interface é como um contrato, não possui nenhuma implementação. Ela possui apenas declaração de métodos que não possuem suas definições. Diferentemente de uma classe abstrata, uma interface não pode conter variáveis e nem construtores. Os métodos declarados em uma interface devem ser implementados pela classe que utiliza a interface. Em outras palavras, uma interface é puramente uma definição.

Interfaces são úteis quando é preciso definir as funcionalidades que as classes herdadas devem implementar, mas deixar todos os detalhes de como essa implementação deve ser feita inteiramente para a classe derivada. Uma classe que implemente uma interface deve implementar todos os métodos da interface.

Uma das grandes vantagens da interface é que na grande maioria das linguagens de programação uma classe pode implementar múltiplas interfaces, diferentemente de uma classe abstrata. (CLARK, 2013)

A Figura 30 mostra a implementação de interface em C#, todos os métodos da interface `Animal` devem ser implementados pelas classes que a utilizam, neste caso, a classe `Cachorro`.

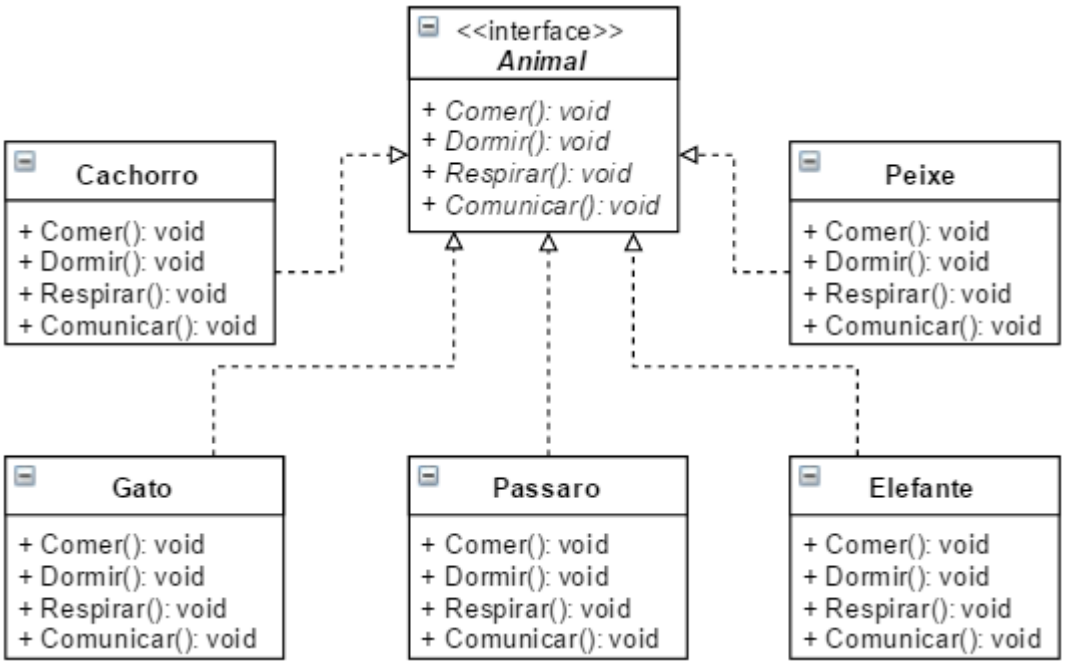
FIGURA 30 – EXEMPLO DE INTERFACE EM C#

```
1  using System;
2  namespace Exemplo_Interface
3  {
4      interface Animal
5      {
6          void Comer();
7          void Dormir();
8          void Respirar();
9      }//Fim da interface Animal
10     class Cachorro : Animal
11     {
12         public void Comer() { Console.WriteLine("Comendo..."); }
13         public void Dormir() { Console.WriteLine("Dormindo..."); }
14         public void Respirar() { Console.WriteLine("Respirando..."); }
15         public void Latir() { Console.WriteLine("Latindo..."); }
16         public void Abanar_rabo() { Console.WriteLine("Abanando o rabo..."); }
17     }//Fim da classe cachorro
18     class Program
19     {
20         static void Main(string[] args)
21         {
22             Cachorro rex = new Cachorro(); //Criar um cachorro
23             rex.Comer(); //Comendo...
24             rex.Abanar_rabo(); //Abanando o rabo...
25             Console.ReadLine();
26         }
27     }
28 }
```

FONTE: o próprio autor

Como classes e interfaces são diferentes, em UML, uma interface deve possuir o estereótipo “<<interface>>” ao topo do primeiro retângulo. Além disso, todas as classes que implementam uma interface devem ter um relacionamento com a interface através de uma seta tracejada, conforme é mostrado na Figura 31. (DENNIS, 2009)

FIGURA 31 – EXEMPLO DE INTERFACE



FONTE: o próprio autor

5 METODOLOGIA

Este trabalho tem como ponto de partida o compilador D+ (FURLAN, 2016), construído durante a disciplina de compiladores. Muitos dos conceitos de compiladores aprendidos durante a disciplina foram utilizados e reforçados neste trabalho. No entanto, a linguagem D+ foi escrita para uma programação estruturada e não comporta, atualmente, a escrita de um programa orientado a objetos, que é o objetivo deste trabalho.

Com relação ao desenvolvimento do compilador, ele será escrito na linguagem C++, utilizando os conceitos de orientação a objetos e as estruturas de dados fornecidas pelo C++, versão 14.

5.1 FERRAMENTAS

A principal ferramenta utilizada neste trabalho é o Microsoft Visual Studio Community 2017, que foi utilizado para a programação do compilador na linguagem C++. O Visual Studio é um ambiente de desenvolvimento integrado (IDE) gratuito, fornecido pela Microsoft para criação de aplicativos Windows. Além da familiaridade com a ferramenta, por utilizá-la diariamente como profissional da área de desenvolvimento de software, o maior ganho dessa ferramenta é o excelente ambiente de depuração, no qual a inspeção dos elementos e de seus valores é flexível, além de ser possível ter um alto controle sobre a aplicação que está sendo desenvolvida.[1]

Para transformar os arquivos MEPA em executáveis, é preciso rodar uma máquina virtual de 16 bits do DOS. A máquina virtual DOSBox foi utilizada para gerar os arquivos executáveis.[2]

A ferramenta Confluence, que é uma wiki não gratuita, foi utilizada para armazenar informações, imagens, códigos fonte e diagramas de UML.[3]

Junto ao Confluence, são integradas diversas macros e uma muito utilizada neste trabalho é o draw.io. Draw.io é uma aplicação de diagramação web construída em cima

[1] https://www.visualstudio.com/pt-br/vs/community
[2] http://www.dosbox.com/download.php?main=1
[3] https://www.atlassian.com/software/confluence

do mxGraph (um componente de diagramação feito em JavaScript e de código aberto). Todo o conteúdo de UML será feito por essa ferramenta, que caso não seja utilizada a partir do Confluence, pode ser utilizada gratuitamente e integrada ao Google Drive, OneDrive e Dropbox, mantendo os diagramas salvos na nuvem.[1]

E por fim, outra ferramenta utilizada é o Microsoft Excel. O Excel é um editor de planilhas que está incluso no pacote de aplicações do Microsoft Office. Hoje é, com grande vantagem, o aplicativo de planilhas eletrônicas dominante no mercado, disponível para plataformas Windows e Macintosh, além de dispositivos móveis como Android, iOS e Windows Phone. Neste trabalho, o Excel será utilizado para criar a tabela de análise do primeiro símbolo e a tabela de derivação da gramática para a linguagem Chimera.[2]

5.2 GRAMÁTICA DA LINGUAGEM CHIMERA

A gramática utilizada como base para Chimera é a do compilador D+. Esta gramática pode ser vista na Tabela 9. Entretanto, a linguagem para o D+ é estruturada e não permite uma programação orientada a objetos.

TABELA 9 – GRAMÁTICA D+

Declarações	
1.	programa → lista-decl
2.	lista-decl → decl lista-decl | decl
3.	decl → decl-const | decl-var | decl-proc | decl-func
4.	decl-const → CONST ID = literal ;
5.	decl-var → VAR espec-tipo var ;
6.	espec-tipo → INT | REAL | CHAR | BOOL | STRING
7.	decl-proc → SUB espec-tipo ID (params) bloco END-SUB
8.	decl-func → FUNCTION espec-tipo ID (params) bloco END-FUNCTION
9.	params → lista-param | Ɛ
10.	lista-param → lista-param , param | param
11.	param → VAR espec-tipo lista-var BY mode
12.	mode → VALUE | REF
Comandos	
13.	bloco → lista-com
14.	lista-com → comando lista-com | Ɛ
15.	comando → cham-proc | com-atrib | com-selecao | com-repeticao | com-desvio | com-leitura | com-escrita | decl-var | decl-const
16.	com-atrib → var = exp ;

[1] https://www.draw.io
[2] http://office.microsoft.com/excel

17.	com-selecao → IF exp THEN bloco END-IF | IF exp THEN bloco ELSE bloco END-IF
18.	com-repeticao → WHILE exp DO bloco LOOP | DO bloco WHILE exp ; | REPEAT bloco UNTIL exp ; | FOR ID = exp-soma TO exp-soma DO bloco NEXT
19.	com-desvio → RETURN exp ; | BREAK ; | CONTINUE ;
20.	com-leitura → SCAN (lista-var) ; | SCANLN (lista-var) ;
21.	com-escrita → PRINT (lista-exp) ; | PRINTLN (lista-exp) ;
22.	cham-proc → ID (args) ;
Expressões	
23.	lista-exp → exp , lista-exp | exp
24.	exp → exp-soma op-relac exp-soma | exp-soma
25.	op-relac → <= | < | > | >= | == | <>
26.	exp-soma → exp-mult op-soma exp-soma | exp-mult
27.	op-soma → + | - | OR
28.	exp-mult → exp-mult op-mult exp-simples | exp-simples
29.	op-mult → * | / | DIV | MOD | AND
30.	exp-simples → (exp) | var | cham-func | literal | op-unario exp
31.	literal → NUMINT | NUMREAL | CARACTERE | STRING | valor-verdade
32.	valor-verdade → TRUE | FALSE
33.	cham-func → ID (args)
34.	args → lista-exp | Ɛ
35.	var → ID | ID [exp-soma]
36.	lista-var - var , lista-var | var
37.	op-unario → + | - | NOT

FONTE: FURLAN, 2016

Para que a seja possível escrever um programa orientado à objetos, o primeiro passo deste trabalho foi reformular a gramática D+, adicionando-se novas palavras reservadas, novas regras de sintaxe e de semântica, conforme pode ser visto na Tabela 10.

TABELA 10 – GRAMÁTICA CHIMERA

Declarações	
1.	programa → lista-decl
2.	lista-decl → decl lista-decl | decl
3.	decl → decl-const | decl-var | decl-proc | decl-func | decl-struct | decl-class
4.	decl-const → CONST ID = literal ;
5.	decl-var → VAR espec-tipo lista-decl-var; | VAR POINTER espec-tipo lista-decl-var;
6.	espec-tipo → INT | FLOAT | CHAR | BOOL | STRING
7.	decl-proc → SUB espec-tipo ID (params) bloco END-SUB
8.	decl-func → FUNCTION espec-tipo ID (params) bloco END-FUNCTION
9.	params → lista-param | ε
10.	lista-param → lista-param , param | param
11.	param → VAR espec-tipo lista-decl-var BY mode
12.	mode → VALUE | REF
13.	decl-struct → STRUCT ID lista-decl-struct END_STRUCT
14.	lista-decl-struct → lista-decl-struct decl-var | decl-var
15.	decl-class → CLASS ID bloco-class END_CLASS | CLASS ID : ID bloco-class END_CLASS

16.	bloco-class →bloco-class espec-acesso : lista-membro-class | espec-acesso : lista-membro-class
17.	espec-acesso → PRIVATE | PROTECTED | PUBLIC
18.	lista-membro-class → lista-membro-class membro-class | membro-class
19.	membro-class → decl-var | decl-proc | decl-func
Comandos	
20.	bloco → lista-com
21.	lista-com → comando lista-com | Ɛ
22.	comando → cham-proc | com-atrib | com-selecao | com-repeticao | com-desvio | com-leitura | com-escrita | decl-var | decl-const
23.	com-atrib → var = exp ;
24.	com-selecao → IF exp THEN bloco END-IF | IF exp THEN bloco ELSE bloco END-IF
25.	com-repeticao → WHILE exp DO bloco LOOP | DO bloco WHILE exp ; | REPEAT bloco UNTIL exp ; | FOR ID = exp-soma TO exp-soma DO bloco NEXT
26.	com-desvio → RETURN exp ; | BREAK ; | CONTINUE ;
27.	com-leitura → SCAN (lista-var) ; | SCANLN (lista-var) ;
28.	com-escrita → PRINT (lista-exp) ; | PRINTLN (lista-exp) ;
29.	cham-proc → id-composto (args) ;
30.	id-composto → ID | ID_SEL_ELEMENTO id-composto | ID_SEL_PONTEIRO id-composto
Expressões	
31.	lista-exp → exp , lista-exp | exp
32.	exp → exp-soma op-relac exp-soma | exp-soma
33.	op-relac → <= | < | > | >= | == | <>
34.	exp-soma → exp-mult op-soma exp-soma | exp-mult
35.	op-soma → + | - | OR
36.	exp-mult → exp-mult op-mult exp-simples | exp-simples
37.	op-mult → * | / | DIV | MOD | AND
38.	exp-simples → (exp) | var | cham-func | literal | op-unario exp
39.	literal → NUMINT | NUMREAL | CARACTERE | STRING | valor-verdade
40.	valor-verdade → TRUE | FALSE
41.	cham-func → id-composto (args)
42.	args → lista-exp | Ɛ
43.	var → id-composto | id-composto [exp-soma]
44.	lista-var → var , lista-var | var
45.	lista-decl-var → var-decl , lista-decl-var | var-decl
46.	var-decl → ID | ID [exp-soma]
47.	op-unario → + | - | NOT

FONTE: o próprio autor

As principais inclusões na tabela acima foram as regras de declaração de estrutura e classe, acesso a membros da classe e identificador composto. Essas e outras alterações podem ser vistas nas seguintes regras:

- Regra 3: Adicionado declaração de estrutura e classe na lista de declarações;
- Regra 5: Adicionado declaração de ponteiro;
- Regra 13: Declaração de estrutura;
- Regra 14: Lista de declarações para uma estrutura;
- Regra 15: Declaração de classe;

- Regra 16: Bloco de uma classe;
- Regra 17: Especificações de acesso para os membros de uma classe;
- Regra 18: Lista de membros de uma classe;
- Regra 19: Membros de uma classe;
- Regra 29: Ajustado para aceitar identificador composto na chamada de procedimentos;
- Regra 30: Identificador composto;
- Regra 41: Ajustado para aceitar identificador composto na chamada de funções;
- Regra 43: Ajustado para aceitar variáveis com identificador composto;
- Regra 45: Lista de declaração de variável;
- Regra 46: Declaração de variável;

Após a criação da gramática, foram aplicadas algumas técnicas para garantir que a gramática esteja na Forma Normal de Greibach (FNG). Uma gramática livre de contexto está na FNG se todas as regras de produção à direita iniciam com um símbolo terminal, opcionalmente seguidas de variáveis. Uma exceção para esta regra pode ser definida permitindo-se a utilização de produções vazias (épsilon, ε) por ser um membro da linguagem definida. As técnicas utilizadas foram:

- Análise do Primeiro Símbolo (APS) – No qual é construída uma tabela a partir da análise do primeiro símbolo de cada produção, descobrindo para qual produção uma variável deve derivar em cada caso diferente de símbolo de entrada.
- Produções ε - Originalmente a análise do primeiro símbolo não permite produções vazias, essa técnica permite adaptarmos a APS, analisando os símbolos seguintes às variáveis que podem derivar em ε.
- Fatoração à Esquerda (FE) – Para a gramática LL(1), devemos adiar o impasse entre duas regras de produção que possam ser aplicadas. Para corrigirmos isso, utilizamos a técnica de FE.

- Eliminação de Recursão à Esquerda (ERE) - Uma gramática é dita recursiva à esquerda se ela tiver um não-terminal A tal que A =>Aα para alguma cadeia *w*. Para eliminarmos a recursividade à esquerda, reescrevemos a produção com recursividade para a direita.

Na Tabela 11, é possível ver a gramática após a aplicação das técnicas acima mencionadas. Esta é a versão da gramática que pode ser programada para a construção do compilador Chimera.

TABELA 11 – GRAMÁTICA CHIMERA NA FNG

Declarações	
1.	programa → lista-decl
2.	lista-decl → decl lista-decl | ε
3.	decl → decl-const | decl-var | decl-proc | decl-func | decl-struct | decl-class
4.	decl-const → CONST ID = literal ;
5.	decl-var → VAR espec-tipo lista-decl-var; | VAR POINTER espec-tipo lista-decl-var;
6.	espec-tipo → INT | FLOAT | CHAR | BOOL | STRING
7.	decl-proc → SUB espec-tipo ID (params) bloco END-SUB
8.	decl-func → FUNCTION espec-tipo ID (params) bloco END-FUNCTION
9.	params → lista-param | ε
10.	lista-param → param lista-param'
11.	lista-param' → , param lista-param'
12.	param → VAR espec-tipo lista-decl-var BY mode
13.	mode → VALUE | REF
14.	decl-struct → STRUCT ID lista-decl-struct END_STRUCT
15.	lista-decl-struct → decl-var lista-decl-struct | ε
16.	decl-class → CLASS ID bloco-class END_CLASS | CLASS ID : ID bloco-class END_CLASS
17.	bloco-class → espec-acesso : lista-membro-class bloco-class | ε
18.	espec-acesso → PRIVATE | PUBLIC
19.	lista-membro-class → membro-class lista-membro-class | ε
20.	membro-class → decl-var | decl-proc | decl-func
Comandos	
21.	bloco → lista-com
22.	lista-com → comando lista-com | Ɛ
23.	comando → id-composto comando' | com-selecao | com-repeticao | com-desvio | com-leitura | com-escrita | decl-var | decl-const
24.	comando' → (args) ; | = exp ; | [exp-soma] = exp ; | ε
25.	com-selecao → IF exp THEN bloco com-selecao'
26.	com-selecao' → END-IF | ELSE bloco END-IF
27.	com-repeticao → WHILE exp DO bloco LOOP | DO bloco WHILE exp ; | REPEAT bloco UNTIL exp ; | FOR ID = exp-soma TO exp-soma DO bloco NEXT
28.	com-desvio → RETURN exp ; | BREAK ; | CONTINUE ;
29.	com-leitura → SCAN (lista-var) ; | SCANLN (lista-var) ;
30.	com-escrita → PRINT (lista-exp) ; | PRINTLN (lista-exp) ;
31.	id-composto → ID | ID_SEL_ELEMENTO id-composto | ID_SEL_PONTEIRO id-composto
Expressões	

32.	lista-exp → exp lista-exp'
33.	lista-exp' → , lista-exp | ε
34.	exp → exp-soma exp'
35.	exp' → op-relac exp-soma exp' | ε
36.	op-relac → <= | < | > | >= | == | <>
37.	exp-soma → exp-mult exp-soma'
38.	exp-soma' → op-soma exp-soma' | ε
39.	op-soma → + | - | OR
40.	exp-mult → exp-simples exp-mult'
41.	exp-mult' → op-mult exp-simples exp-mult' | ε
42.	op-mult → * | / | DIV | MOD | AND
43.	xp-simples → (exp) | id-composto exp-simples' | literal | op-unario exp
44.	exp-simples' → [exp-soma] | (args) | ε
45.	literal → NUMINT | NUMREAL | CARACTERE | STRING | valor-verdade
46.	valor-verdade → TRUE | FALSE
47.	args → lista-exp | ε
48.	lista-var → id-composto lista-var'
49.	lista-var' → , lista-var | [exp-soma] lista-var" | ε
50.	lista-var" → , lista-var | ε
51.	lista-decl-var → ID lista-decl-var'
52.	lista-decl-var' → , lista-decl-var | [exp-soma] lista-decl-var" | ε
53.	lista-decl-var" → , lista-decl-var | ε
54.	op-unario → + | - | NOT

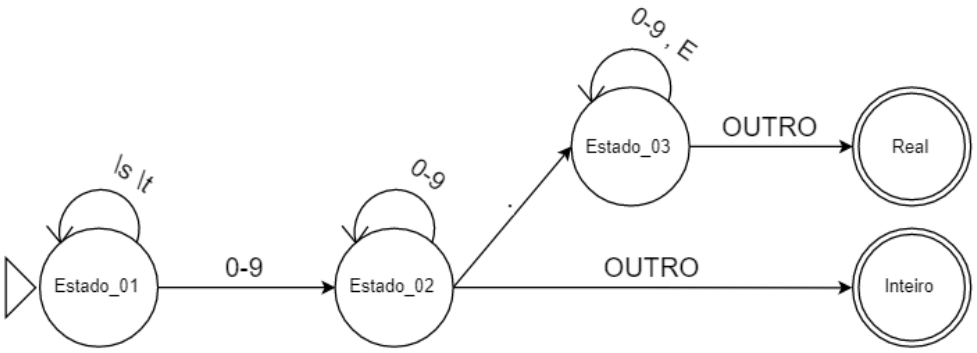
FONTE: o próprio autor

5.3 ANÁLISE LÉXICA

A análise léxica foi implementada manualmente, sem o auxílio de uma ferramenta que a construa automaticamente, como Lex[1]. Por este motivo, antes de iniciarmos a análise léxica, é preciso montar os estados possíveis para a gramática manualmente, através do diagrama de transição. Um exemplo deste diagrama é dado pela Figura 32, na qual são definidos os *tokens* para os números reais e inteiros. Esta representação indica que a cadeia de caracteres deve iniciar no Estado_01 com um número de 0 a 9, repeti-los ou não no Estado_02 até encontrar um ponto para passar ao Estado_03 ou então se encerrar com outro caractere definindo um número inteiro. Caso tenha entrado no Estado_03, pode-se repetir ou não os números de 0 a 9 até encontrar qualquer caractere diferente definindo um número real. Todos os diagramas de transição utilizados neste trabalho podem ser vistos no APÊNDICE A.

[1] http://dinosaur.compilertools.net/lex/

FIGURA 32 –DIAGRAMA DE TRANSIÇÃO PARA OS NÚMEROS INTEIROS E REAIS



FONTE: o próprio autor

O funcionamento do analisador léxico é composto pelos seguintes passos:

- Início da leitura do arquivo que contém o código fonte;
- Leitura da linha e armazenamento em um *buffer;*
- Análise dos caracteres contidos na linha em *buffer*, separando e identificando os *tokens* e lexemas;
- Armazenamento do *token*, lexema e linha em uma estrutura de dados;
- O processo de leitura da linha, análise dos caracteres e armazenamento em uma lista é repetido até o analisador léxico encontrar o *token* que indica o fim do programa.

A programação foi realizada por comandos de seleção encadeados. Começando pelo estado inicial do diagrama de transição, já definido anteriormente, o analisador trabalha através do comando de seleção "*case*", que conforme o caractere lido, irá decidir qual é o próximo estado a ser verificado.

Toda vez que o analisador léxico chega a um estado final, um *token* é reconhecido. O *token* e seu lexema são então adicionados a uma lista que irá armazenar esses valores para serem utilizados na análise sintática. O analisador então retorna ao estado inicial, para continuar a busca por um novo *token*, enquanto não encontrar o *token* que indica o fim do programa.

Durante este processo, podem surgir caracteres inválidos. Quando caracteres inválidos são encontrados, o processo de compilação é interrompido e uma mensagem é apresentada com a linha e o motivo da causa do erro.

Também foi construída uma estrutura de dados que contém todas as palavras reservadas, que são palavras específicas para o compilador reconhecer os seus comandos. Os *tokens* gerados por essas palavras reservadas não podem estar fora da sintaxe permitida para cada uma delas, esta verificação é feita pela análise sintática.

Ao fim da compilação do programa, é gerado um arquivo *.csv* com todos os *tokens* e lexemas gerados, além da linha onde estão inseridos no programa.

Um exemplo do arquivo *.csv Token*/Lexema gerado a partir de um trecho de código da Figura 33 é mostrada na Tabela 12.

FIGURA 33 – PROGRAMA 5.3.1

```
1  class filmes_class
2      public:
3          var int id;
4          var string nome;
5          var int ano;
6  end_class
```

FONTE: o próprio autor

TABELA 12 – EXEMPLO ARQUIVO TOKEN/LEXEMA CHIMERA PARA O PROGRAMA 1

LEXEMA	TOKEN	LINHA
class	CLASSE	1
filmes_class	IDENTIFICADOR	1
var	VAR	2
int	TIPO_INT	2
id	IDENTIFICADOR	2
;	PONTO_VIRGULA	2
var	VAR	3
string	TIPO_STRING	3
nome	IDENTIFICADOR	3
;	PONTO_VIRGULA	3
var	VAR	4
int	TIPO_INT	4
ano	IDENTIFICADOR	4
;	PONTO_VIRGULA	4
end_class	END_CLASS	5

FONTE: o próprio autor

5.4 ANÁLISE SINTÁTICA

Esta fase também será executada pelo compilador, sem a utilização de ferramentas que geram a análise sintática automaticamente, como o YACC[1]. Por este motivo, a metodologia utilizada para a análise sintática deste trabalho é a LL(1), pois ela possui uma abrangência aceitável em relação a quantidade de linguagens que é capaz de reconhecer e é de fácil implementação manual.

Com a gramática já reescrita para a FNG e com a tabela de derivação em mãos, foi realizada a implementação recursiva LL(1) que irá fazer a análise sintática do compilador. Cada produção resultou em uma função, que pode ou não ser recursiva.

Como a análise léxica já foi concluída antes deste passo, já possuímos os pares de *tokens* e lexemas em uma lista, a análise sintática inicia o seu trabalho varrendo essa lista a partir do primeiro par. Ao ler o primeiro *token*, a produção inicial da gramática é chamada e irá ativar as produções seguintes conforme a sequência dos *tokens* lidos da lista. Caso a análise sintática consiga chegar ao *token* que representa o fim do programa, significa que a compilação foi bem-sucedida. Se em algum momento um *token* não for aceito pela sintaxe, o processo de compilação é interrompido e é retornado um erro com os detalhes do problema para o usuário.

O código fonte da Figura 34 é da gramática D1, extremamente simplificada, que é de um trecho de um programa que faz a declaração de constantes.

D1: programa -> lista_decl
lista_decl -> decl lista_decl_1 | decl
decl -> decl_const
decl_const -> CONST ID = literal ;

[1] http://dinosaur.compilertools.net/

FIGURA 34 – EXEMPLO DE ANALISADOR LL(1)

```
1  void Sintatica::Programa(){
2      Lista_decl();
3      if (token == FIM_PROGRAMA)
4          cout << "Compilacao - OK" << endl;
5      else
6          cout << "Erro de compilacao" << endl;
7  }
8  void Sintatica::Lista_decl(){
9      if (token == CONSTANTE) {
10         Decl();
11         Lista_decl_1();
12     }
13     else
14         Erro(ERR_LIST_DECL);
15 }
16 void Sintatica::Lista_decl_1(){
17     if (token == CONSTANTE) {
18         Lista_decl();
19     }
20     else if (token != FIM_PROGRAMA)
21         Erro(ERR_LIST_DECL);
22 }
23 void Sintatica::Decl(){
24     if (token == CONSTANTE) {
25         Decl_const();
26     }
27     else
28         Erro(ERR_DECL);
29 }
30 void Sintatica::Decl_const(){
31     Aceitar(CONSTANTE, ERR_CONTANTE);
32     Aceitar(OP_ATRIBUICAO, ERR_OP_ATRIBUICAO);
33     Literal();
34     Aceitar(PONTO_VIRGULA, ERR_PONTO_VIRGULA);
35 }
```

FONTE: o próprio autor

5.5 TABELA DE SÍMBOLOS

Para armazenar as informações dos identificadores declarados é utilizada uma tabela de símbolos. Essas informações devem ser armazenadas enquanto for processado o trecho do programa que utiliza esta declaração.

Como foi escolhido o analisador descendente recursivo, boa parte do compilador será construído complementando as rotinas já criadas durante a análise sintática. A manipulação da tabela de símbolos, as validações semânticas e também a geração de código intermediário fazem parte dessa complementação.

Foi criada uma estrutura chamada S_Simbolos que armazena diversas propriedades da tabela de símbolos, algumas delas são: chave única, identificador, tipo, se é um ponteiro, se é um *array*, valor, quantidade de parâmetros, a forma de passagem do parâmetro, o escopo pai deste símbolo, qual classe este símbolo pertence, o acesso (público ou privado), a linha do código que está o símbolo e outras informações pertinentes para verificação dos símbolos. O código desta estrutura pode ser visto na Figura 35.

FIGURA 35 – ESTRUTURA S_SIMBOLOS

```
struct S_Simbolos
{
    int chave; //Chave do registro
    string identificador; //Identificador
    string categoria; //Categoria do identificador
    string tipo; //Tipo de dados
    bool ponteiro; //Identifica se é um ponteiro
    bool array; //Identifica se é um array[]
    string valor; //Armazena o valor de uma constante
    int qtd_params; //Indica a quantidade de parâmetros de uma função/Procedimento
    string passby; //Passagem do parâmetro. VALUE | REF
    int pai; //Indica a chave que o registro pertence;
    int classe; //Indica a classe que o registro pertence
    Acesso access; //Indica o tipo de acesso ao membro. PRIVATE | PROTECTED | PUBLIC
    bool valido; //Indica se o registro ainda está ativo
    int linha; //Linha da declaração
    int pos_pilha; //Posição da pilha - Utilizado para a MEPA
    int pos_pilha_ini_str; //Posição inicial de uma String - MEPA
    int tam_str; //Tamanho da String
    string rotulo; //Armazena o rótulo a ser utilizado na MEPA
    string escopo; //A qual escopo o registro pertence. Utilizado em classes
    int pai_heranca; //Indica a classe pai

    //Sobrecarga do operador < para ordenar a TS pela chave
    bool operator < (const S_Simbolos& ts)
    {
        return (chave < ts.chave);
    }

}; typedef S_Simbolos S_Simbolos;
```

FONTE: o próprio autor

A classe C_Tabela_Simbolos foi criada para manipular a tabela de símbolos. O container *vector*, que é uma propriedade desta classe, foi utilizado para armazenar a

estrutura da tabela de símbolos. Alguns métodos para a manipulação da tabela de símbolos foram implementados na classe, os principais deles são:

- bool Inserir(S_Simbolos) – que insere um novo identificador na tabela de símbolos, enviando a estrutura S_Simbolos com as informações já preenchidas como parâmetro. Caso o valor seja inserido é retornado true, caso contrário false;
- bool Consultar(string); – retorna se o identificador existe ou não na tabela de símbolos; O identificador é passado como parâmetro;
- string Buscar_Tipo(string) – que retorna o tipo de um identificador existente na tabela de símbolos; O identificador é passado como parâmetro;
- string Buscar_Categoria(string) – que retorna a categoria de um identificador existentes na tabela de símbolos. O identificador é passado como parâmetro;

Além das rotinas citadas acima, outras rotinas foram criadas para manipular os dados contidos na tabela de símbolos.

Na Tabela 13 podemos ver a tabela de símbolos para o programa 5.5.1 da Figura 36, que armazena em uma classe as informações de id, nome e ano de um filme. Neste exemplo, apenas as colunas relevantes ao programa são demonstradas. Outras informações estão contidas na tabela de símbolos, mas não são demonstradas abaixo.

FIGURA 36 – PROGRAMA 5.5.1

```
1  class filmes_class
2      public:
3          var int id;
4          var string nome;
5          var int ano;
6  end_class
7
8  sub main()
9      var filmes_class f;
10
11     f.id = 1;
12     f.nome = "007-Spectre";
13     f.ano = 2015;
14
15     println(f.id);
16     println(f.nome);
17     println(f.ano);
18 end_sub
```

FONTE: o próprio autor

TABELA 13 – EXEMPLO DA TABELA DE SÍMBOLOS PARA O PROGRAMA 5.5.1

#	ID	CATEGORIA	TIPO	Pai	Classe	Acesso	Linha	Pilha	Tam Str	Escopo
1	filmes_class	CLASSE		0	0	0	1	-1	0	
2	id	VAR	TIPO_INT	1	1	0	3	0	0	filmes_class
3	nome	VAR	TIPO_STRING	1	1	0	4	15	12	filmes_class
4	ano	VAR	TIPO_INT	1	1	0	5	16	0	filmes_class
5	main	SUB		0	0	0	8	-1	0	
6	f	VAR	filmes_class	5	0	0	9	-1	0	
7	id	VAR	TIPO_INT	6	1	0	9	0	0	
8	nome	VAR	TIPO_STRING	6	1	0	9	15	12	
9	ano	VAR	TIPO_INT	6	1	0	9	16	0	

FONTE: o próprio autor

5.6 ANÁLISE SEMÂNTICA

As verificações semânticas também são feitas em conjunto com as rotinas da análise sintática e ela está fortemente ligada a tabela de símbolos. Abaixo são listados alguns exemplos de como são tratadas as validações semânticas neste trabalho.

- Identificador não declarado – Assim que a declaração de um identificador é feita, ele é introduzido na tabela de símbolos. Toda vez que o compilador encontrar um identificador, é feita uma busca na tabela de símbolos para verificar a sua existência, caso o identificador não exista na tabela de símbolos, o processo de compilação é parado e retornado um erro com a linha, o nome do identificador e a mensagem de que ele não foi declarado.
- Identificador redeclarado – Toda vez que é inserido um identificador na tabela de símbolos através do método `Inserir`, é verificado se este identificador já foi declarado para o escopo atual. Caso já tenha sido declarado, é retornado um erro com a linha, o nome do identificador e a mensagem de que ele já foi declarado.
- Atribuição à constantes – Quando o compilador encontra um comando de atribuição a um identificador, após verificar a existência do identificador na tabela de símbolos, ele verifica se o tipo do identificador é uma constante. Caso seja, o processo de compilação é abortado e é retornado um erro com a

linha, o identificador e a mensagem de que não é possível atribuir valores a constantes.

- Erro de tipo em comandos de seleção e repetição – Quando é encontrado um comando de seleção ou repetição pelo compilador, é verificado se o tipo da expressão deste comando é de valor booleano ou inteiro. Caso a expressão seja de um tipo diferente, a compilação do programa é interrompida e uma mensagem de erro avisa a linha, o nome do comando e a mensagem de incompatibilidade de tipo.
- Número de parâmetros incompatíveis com a declaração da função – Outro erro tratado pela análise semântica é a verificação das assinaturas das funções. Caso seja encontrado pelo compilador a utilização de uma função que não condiz com a assinatura de sua declaração, por exemplo, a quantidade de parâmetros está diferente, o processo de compilação é parado e a mensagem de erro é dada ao usuário, mostrando a linha da ocorrência e o motivo.
- Atribuição a tipos incompatíveis – O tipo *char* pode aceitar como atribuição um outro *char* ou um tipo inteiro, qualquer outro tipo de dado diferente desses atribuído a uma variável *char*, o compilador não reconhece e deve ser interrompido o processo de compilação, alertando o programador desta inconsistência. Esta verificação entre tipos é feita para todos os tipos de dados, um outro exemplo seria o tipo booleano, em que é aceito outro tipo booleano ou um tipo inteiro, ou uma *string*, no qual é aceito apenas atribuições do tipo *string*.

Os exemplos de verificações semânticas citados acima são apenas algumas das validações que foram tratadas neste trabalho, além dessas, outras verificações são feitas para garantir que a gramática da linguagem orientada a objetos Chimera seja a mais completa. O objetivo é alertar o usuário que está escrevendo o programa informando sempre a linha e o motivo do erro encontrado pelo validador.

5.7 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Como já foi citado anteriormente, é utilizada a MEPA para a geração do código objeto, para que isso seja possível, precisamos gerar o código intermediário para a linguagem da MEPA. A geração do código intermediário será feita em conjunto com as rotinas da análise sintática.

A classe C_Mepa foi criada para manipular as inserções do código intermediário. Foi criado um método para cada uma das instruções disponíveis da MEPA. Quando esses métodos são chamados, as informações ficam armazenadas em memória para ao fim da compilação ser gerado o arquivo com o código intermediário, que contém os comandos da MEPA. A extensão desse arquivo é *.mep*.

Uma vez gerado o arquivo com as instruções da MEPA, é necessário gerar o código objeto para podermos executar o programa. Para gerar o código objeto, precisamos compilar o código intermediário gerado pela Chimera na MEPA em um ambiente DOS de 16 bits. As instruções para a geração do executável estão detalhadas no manual do usuário.

Na Figura 37 é demonstrado como a classe para a geração da MEPA será utilizada entre as rotinas da análise sintática, no caso deste exemplo, é feita a geração do código MEPA para a estrutura seletiva de condição “*if-else*”.

FIGURA 37 – EXEMPLO DE GERAÇÃO DA MEPA

```
void C_Analise_Sintatica::Com_selecao()
{
    string tipo_exp;
    string DSVF, DSVS;

    Aceitar_Token(CONDICAO_IF, ERR_CONDICAO_IF);
    tipo_exp = Exp();
    //Avalidar Expressão
    mepa.Avaliar_Expressao(mepa.pilha_EXP, ts);
    //MEPA - Desvia se for falso
    DSVF = mepa.DSVF();
    if (tipo_exp != TIPO_INT && tipo_exp != TIPO_BOOLEANO)
        Erro(ERR_SEM_IF);
    Aceitar_Token(CONDICAO_THEN, ERR_CONDICAO_THEN);
    Bloco();
    //MEPA - Desvia sempre
    DSVS = mepa.DSVS();
    //MEPA - NADA - caso o IF tenha sido falso cai aqui
    mepa.NADA(DSVF);
    Com_selecao_1();
    //MEPA - NADA - Caso o IF tenha sido verdadeiro, cai direto aqui
    mepa.NADA(DSVS);
}
```

FONTE: o próprio autor

Para exemplificar o resultado da geração do código MEPA, na Figura 38 temos uma expressão condicional em Pascal e seu respectivo código MEPA logo abaixo. Como pode ser visto, os comandos da MEPA são criados em representação linear de ordem pós-fixa.

FIGURA 38 – EXEMPLO DE CÓDIGO MEPA

```
if q < 5 then
    a=1
else
    a=2;
```

```
CRVL q
CRCT 5
CMME
DVSF R1
CRCT 1
ARMZ a
DSVS R2
R1: NADA
CRCT 2
ARMZ a
R2: NADA
```

FONTE: o próprio autor

5.8 FUNCIONAMENTO DO COMPILADOR CHIMERA

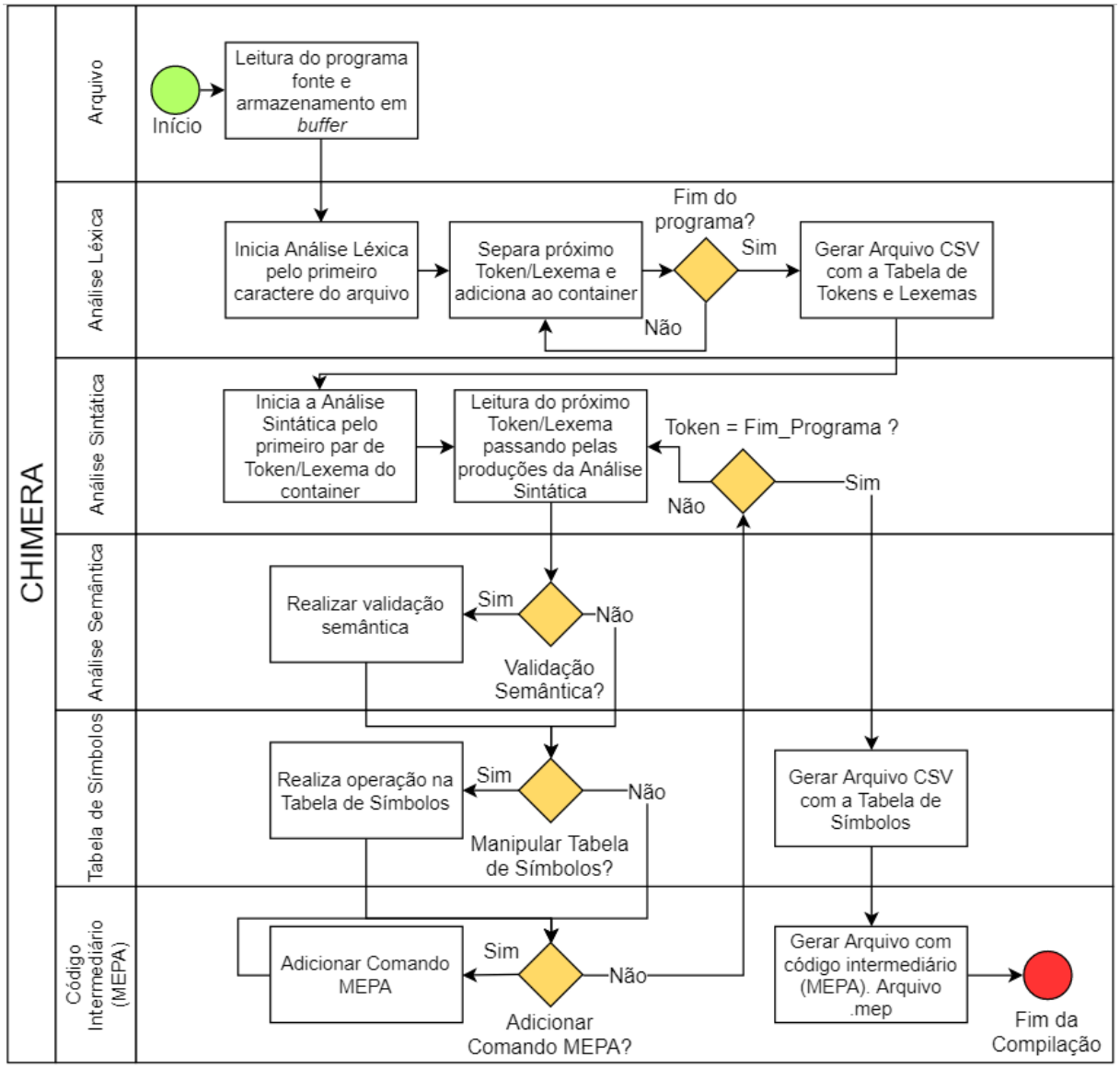
Após escrever o programa na linguagem Chimera em um arquivo com a extensão *.chi*, seguindo as instruções do manual do usuário (APÊNDICE B), o utilizador deve iniciar o processo de compilação do programa.

Quando o processo é finalizado com sucesso, são gerados os seguintes arquivos, no qual PROGRAMA é o nome do arquivo com a extensão *.chi* que está sendo compilado:

- PROGRAMA_TokLex.csv : Arquivo contendo a tabela Token/Lexema;
- PROGRAMA_TabSim.csv : Arquivo contendo a tabela de símbolos do programa compilado;
- PROGRAMA_TabSim.txt : Outro formato de arquivo para a tabela de símbolos;
- PROGRAMA.mep : Arquivo contendo o código intermediário MEPA.

Na Figura 39, podemos analisar o funcionamento do processo de compilação como um todo. Cada processo do compilador destacado abaixo representa uma fase da compilação, iniciando pela fase do arquivo, através da leitura do código fonte e finalizando com a geração do arquivo MEPA, na fase do código intermediário.

FIGURA 39 – FUNCIONAMENTO DO COMPILADOR CHIMERA



FONTE: o próprio autor

5.9 MANIPULAÇÃO DE STRINGS

Strings são cadeias de caracteres que sempre devem ser manipuladas em conjunto. Toda vez que uma String é declarada, um espaço de memória precisa ser reservado para este conjunto de caracteres. Por este motivo, a manipulação de strings na Chimera teve que ser implementada de uma forma que fosse possível determinar a quantidade de caracteres deste conjunto.

Por padrão, a Chimera trabalha com o limite de 30 caracteres. A cada String declarada, será alocado o equivalente a 30 posições de memória. Todas as operações que manipularem Strings levarão esta quantidade em conta.

Um dos grandes desafios deste trabalho foi manipular Strings nessas operações. Por exemplo, quando um ou mais parâmetros são passados para uma função, a MEPA empilha estes parâmetros antes de chamá-la. Na saída da função, é utilizada uma instrução MEPA para desalojar a quantidade de memória alocada para esses parâmetros. Quando o parâmetro é uma String, é necessário empilhar as 30 posições de memória desta String antes de executar o comando MEPA que irá chamar a função. Já dentro da função, a cada manipulação desta String, deve ser levada em conta as posições da String na pilha de parâmetros. Na saída da função, a instrução RTPR da MEPA precisa da informação com a quantidade de memória que deve desalojar para os parâmetros.

Para manipular as Strings de forma eficiente na Chimera, foram acrescentadas algumas informações importantes na tabela de símbolos. Além da posição da pilha, que controla a posição da memória em que a variável está alocada, foi também adicionada a posição inicial da pilha. Ou seja, quando se trata de String, a Chimera armazena a posição inicial da String e a posição final. Desta forma, é possível ter o controle necessário para manipularmos a String. Na Tabela 14, podemos visualizar uma parte de uma tabela de símbolos que contém a declaração de 3 Strings e um número inteiro. Neste exemplo, podemos verificar que a posição inicial de memória de uma String é registrada na propriedade Pilha_Ini_Str e a sua posição final, 30 posições de memória a mais, é armazenada na propriedade Pilha.

TABELA 14 – TABELA DE SÍMBOLOS COM DECLARAÇÃO DE STRINGS

#	ID	CATEGORIA	TIPO	Pai	Pilha	Pilha_Ini_Str
6	main	SUB		0	-1	0
7	x	VAR	TIPO_STRING	6	29	0
8	z	VAR	TIPO_STRING	6	59	30
9	y	VAR	TIPO_INT	6	60	-1
10	w	VAR	TIPO_STRING	6	90	61

FONTE: o próprio autor

Além das informações adicionais na tabela de símbolos, também foram necessários outros métodos para manipular as operações contendo Strings. Por exemplo,

métodos que controlam o empilhamento e desempilhamento das Strings quando estas são passadas como parâmetro em um procedimento ou função. Também foram criados métodos específicos para comandos MEPA que fazem a leitura, armazenamento ou impressão de Strings.

5.10 PONTEIROS

Para ser possível trabalhar com orientação a objetos na Chimera, foi necessário programar a manipulação de ponteiros. O objetivo desta seção é explicar como ela foi realizada na Chimera.

A manipulação de ponteiros na MEPA não difere muito das regras de armazenamento e leitura por valor. No entanto, são comandos diferentes que têm como objetivo manipular o endereço da memória. Por exemplo, para armazenar um valor em uma variável local, é utilizado o ARMZ, para armazenar em um valor indiretamente utilizamos o ARMI. O mesmo para carregar um valor, CRVL para uma variável local e CRVI para carregar um valor indireto. A única exceção é quando passamos um parâmetro como referência para um procedimento ou função, neste caso utilizamos o CREN, ao invés do CRVL quando o parâmetro é passado por valor.

Para a Chimera saber se deverá gerar os comandos de manipulação de ponteiros ou manipulação de valor, foi utilizada novamente a tabela de símbolos para armazenar as informações se a variável é um ponteiro ou não, ou no caso de parâmetro, se ele foi passado por referência ou por valor. A Tabela 15 traz um exemplo de tabela de símbolos com as propriedades Ponteiro e Pass By, que são utilizadas para a manipulação de ponteiros. Quando a propriedade Ponteiro contiver o valor 1, quer dizer que a variável é um ponteiro. Para a propriedade Pass By, quando estiver com o valor REF, o parâmetro é passado por referência, caso contrário, quando estiver com VALUE, é por valor.

TABELA 15 – TABELA DE SÍMBOLOS COM DADOS DE PONTEIRO

#	ID	CATEGORIA	TIPO	Ponteiro	QtdParam	Pass By	Pai
1	carrega	SUB		0	2		0
2	a1	PARAMETRO	TIPO_INT	0	0	VALUE	1
3	b1	PARAMETRO	TIPO_INT	1	0	REF	1
4	main	SUB		0	0		0

5	a	VAR	TIPO_INT	0	0		4
6	b	VAR	TIPO_INT	1	0		4

FONTE: o próprio autor

5.11 EXPRESSÕES

Conforme citado na seção 3.3.4, as expressões ambíguas consistem na dupla interpretação de uma sentença formal. Por exemplo, a sentença `a = (3 + 5 * 4) - 2;` é ambígua e dá margem a dupla interpretação. Neste caso, devemos considerar qual é o cálculo correto. Se avaliarmos primeiro a adição `(3 + 5 = 8)` e depois a multiplicação `(8 * 4 = 32)`, o resultado será `(a = 32 - 2 = 30)`. Porém, se avaliarmos primeiro a multiplicação `(5 * 4 = 20)` e depois a adição `(3 + 20 = 23)`, o resultado será `(a = 23 - 2 = 21)`. A convenção usual da matemática determina a segunda interpretação como a correta. Isso se deve a precedência da multiplicação sobre a adição.

Para solucionar este problema semântico, foram criadas funções para converter as expressões para a forma linear pós-fixa, respeitando o peso dos operadores. O compilador empilha as expressões durante a análise sintática e ao final de uma produção de expressão, essa pilha é enviada para uma função que irá reordenar essa pilha para uma ordem pós-fixa. Esta função além de verificar o peso dos operadores, também leva em consideração as expressões que estão entre parênteses, que tem precedência sobre as expressões que estão fora dos parênteses.

5.12 ESTRUTURAS

O primeiro passo para o avanço da implementação de uma classe, foi a criação do conceito de estrutura. Uma estrutura é um conjunto de variáveis de tipos distintos ou não, agrupadas sob um único nome. O seguinte exemplo da Figura 40 declara uma estrutura DiaMesAno na linguagem Chimera, assim como também demonstra como acessar os seus membros, no qual a estrutura e o membro são separados por um ponto.

FIGURA 40 – PROGRAMA 5.12.1

```
1  //Programa que imprime o valor "20/11/2017"
2  struct DiaMesAno
3      var int dia;
4      var int mes;
5      var int ano;
6  end_struct
7
8  sub main()
9      var DiaMesAno dma;
10     dma.dia = 20;
11     dma.mes = 11;
12     dma.ano = 2017;
13     print(dma.dia);
14     print("/");
15     print(dma.mes);
16     print("/");
17     println(dma.ano);
18 end_sub
```

FONTE: o próprio autor

5.13 CLASSES

Esta seção descreve a forma de implementação da orientação a objetos na Chimera.

Para se chegar neste ponto, os desafios citados nas seções 5.9 a 5.12, como manipulação de Strings, ponteiros e estruturas, tiveram que ser ultrapassados, além de garantir a estabilidade do compilador para as operações básicas de uma linguagem estruturada.

A criação do conceito de estrutura facilitou a introdução dos conceitos de orientação a objetos descritos nesta seção, pois ela foi a base para a programação dos mesmos.

5.13.1 Propriedades

A declaração de variáveis em classes se faz da mesma forma que estruturas. A diferença é que é armazenado na tabela de símbolos o acesso a esse membro, se é de tipo privado, protegido ou público.

5.13.2 Métodos

A declaração de métodos em uma classe se faz da mesma forma que uma declaração de procedimento ou função fora dela. De forma semelhante as propriedades, a diferença são as informações de encapsulamento armazenadas na tabela de símbolos.

Existe uma diferença na chamada de métodos de uma classe do que uma simples chamada de procedimento ou função fora do escopo de uma classe. Ao se chamar um método de uma classe, todos as propriedades da instância desta classe são passadas como referência para o método. Essas propriedades da classe fazem parte da contagem de parâmetros de um método, que são armazenados na pilha logo após os parâmetros existentes no método.

O programa 5.13.2.1, representado na Figura 41, imprime um valor passado por um *setter*, que é um método para alterar o valor de uma variável, normalmente privada. Na Figura 42, pode ser visto o trecho do código Mepa na chamada do procedimento `Set_Id` (rótulo R3 na Mepa) com 1 parâmetro e o outro procedimento `Imprime_Id` (rótulo R12 na Mepa) sem parâmetros. A instrução Mepa “CRCT 1” é a constante 1 sendo passada por valor para o *setter* `Set_Id(var int _id by value)` e o comando “CREN 1,0” é a propriedade `id` sendo passada por referência em ambos os métodos da classe.

FIGURA 41 – PROGRAMA 5.13.2.1

```
1  class A
2      private:
3          var int id;
4      public:
5          sub Set_Id(var int _id by value)
6              id = _id;
7          end_sub
8          sub Imprime_Id()
9              println(id);
10         end_sub
11 end_class
12
13 sub main()
14     var A a;
15
16     a.Set_Id(1);
17     a.Imprime_Id();
18 end_sub
```

FONTE: o próprio autor

FIGURA 42 – MEPA DA CHAMADA DOS MÉTODOS DO PROGRAMA 5.13.2.1

```
1  CRCT 1
2  CREN 1,0
3  ; ---------- Chama Procedimento/Funcao - R3 (Set_Id)
4  CHPR R3,1
5  CREN 1,0
6  ; ---------- Chama Procedimento/Funcao - R12 (Imprime_Id)
7  CHPR R12,1
```

FONTE: o próprio autor

Os envios dessas propriedades da classe são necessários para a utilização dos valores desses membros dentro do escopo do método chamado. Para que seja possível a alteração dessas propriedades dentro do método, elas são passadas como referência.

5.13.3 Encapsulamento

Na Chimera, as classes têm seus membros privados como padrão. Os acessos podem ser públicos, privados ou protegidos. As informações de acesso aos membros são armazenadas na tabela de símbolos na propriedade Acesso e tem os valores 0, 1 ou 2 respectivamente.

- 0-Público, não existem restrições de acesso para este membro;
- 1-Privado, o membro só pode ser acessado pela própria classe;
- 2-Protegido, o membro pode ser acessado pela própria classe ou por uma classe derivada (herança).

A propriedade Acesso é verificada quando o membro da classe é acionado em uma operação que envolva o objeto desta classe. Caso o acesso seja feito a um membro não permitido, um erro semântico é disparado no momento da compilação do programa. O método `Consultar_Acesso` da Figura 43 foi criado para verificar se uma membro pode ou não ser acessado. Caso o acesso seja permitido, o método retorna verdadeiro, caso contrário falso.

FIGURA 43 – MÉTODO QUE CONSULTA O ACESSO AO MEMBRO DA CLASSE

```
bool C_Tabela_Simbolos::Consultar_Acesso(string _identificador, int _pai)
{
    //Buscar classe
    //Utilizado para verificar acesso PRIVATE
    int classe = Buscar_Classe(_identificador, _pai);
    //Busco a classe pai, de quem é herdado o atributo?
    //Utilizado para verificar acesso PROTECTED
    int classe_pai = Buscar_Classe_pai(_pai);

    //Faço a varredura na tabela de símbolos
    for (auto it = tabela_simbolos.begin(); it != tabela_simbolos.end(); it++)
    {
        //Identificador + pai é encontrado na tabela de símbolos
        if (it->identificador == _identificador && it->pai == _pai)
        {
            //1-Se o acesso é privado
            if (it->access == PRIVATE)
            {
                //2-Se é diferente de declaração dentro da classe, ou
                //3-Se o registro está inválido, Retorno false
                if (_pai != classe || !it->valido)
                    return false;
                //4-Se for membro interno da classe e válido
                else
                    return true;
            }
            //1-Se for PROTECTED, precisa ser a classe filha para utilizar
            if (it->access == PROTECTED && classe_pai == it->classe)
                return true;
            //1-Caso seja PUBLIC, sem restrições
            if (it->access == PUBLIC)
                return true;
        }
    }
    return false;
}
```

FONTE: o próprio autor

5.13.4 Declaração de objetos

Ao se declarar um objeto na Chimera, todos os membros de uma classe são duplicados na tabela de símbolos. Isso se deve ao fato que cada membro da classe deve corresponder a uma instância única do objeto declarado, se mais de um objeto é

instanciado para a mesma classe, registros diferentes na tabela de símbolos são criados. Cada um desses registros duplicados tem a sua propriedade Pai na tabela de símbolos referenciada a chave do identificador da instância do objeto.

Para exemplificar o funcionamento da tabela de símbolos quando é feita uma declaração de objeto, foi criado o programa 5.13.4.1 como exemplo na Figura 44. Este programa possuiu uma classe chamada `animal` com uma propriedade privada `id` e outros membros públicos como, `nome`, `Set_Id()` e `Imprime()`. O objetivo do programa é apenas popular essas propriedades e imprimi-la. Neste exemplo, foram instanciados dois objetos da classe animal, `dog` e `cat`.

FIGURA 44 – PROGAMA 5.13.4.1

```
1  class animal
2      private:
3          var int id;
4      public:
5          var string nome;
6          sub Set_Id(var int _id by value)
7              id = _id;
8          end_sub
9          sub Imprime()
10             print(id);
11             print(" - ");
12             println(nome);
13         end_sub
14 end_class
15
16 sub main()
17     var animal dog;
18     var animal cat;
19
20     dog.Set_Id(1);
21     dog.nome = "Bella";
22     cat.Set_Id(2);
23     cat.nome = "Murano";
24
25     dog.Imprime();
26     cat.Imprime();
27 end_sub
```

FONTE: o próprio autor

Na Tabela 16, temos a tabela de símbolos gerada pelo programa 5.13.4.1. Podemos verificar neste exemplo, que a propriedade `id` e `nome` e também os métodos `Set_Id()` e `Imprime()` são repetidos logo após que uma instância de objeto da classe `animal` seja declarada. Neste caso, a propriedade Pai desses membros da classe na tabela de símbolos, fica vinculada a chave (coluna #) do registro que armazena os dados da instância do objeto. Exemplo, para a instância *dog*, que tem como chave (#) o valor '8', os membros dessa instância têm o valor da coluna Pai vinculados a esta chave. No caso da instância *cat*, este valor é '13'. Também é possível verificar a coluna Acesso, que contém o valor '0' para os membros públicos e '1' para os membros privados, neste exemplo a propriedade `id`.

TABELA 16 – TABELA DE SÍMBOLOS DO PROGRAMA QUE IMPRIME UM ANIMAL

#	ID	CATEGORIA	TIPO	Pai	Classe	Acesso	Pilha	Pilha_Ini_Str
1	animal	CLASSE		0	0	0	-1	0
2	id	VAR	TIPO_INT	1	1	1	0	-1
3	nome	VAR	TIPO_STRING	1	1	0	30	1
4	Set_Id	SUB		1	1	0	-1	0
5	_id	PARAMETRO	TIPO_INT	4	1	0	0	0
6	Imprime	SUB		1	1	0	-1	0
7	main	SUB		0	0	0	-1	0
8	dog	VAR	animal	7	0	0	-1	-1
9	id	VAR	TIPO_INT	8	1	1	0	-1
10	nome	VAR	TIPO_STRING	8	1	0	30	1
11	Set_Id	SUB		8	1	0	-1	0
12	Imprime	SUB		8	1	0	-1	0
13	cat	VAR	animal	7	0	0	-1	-1
14	id	VAR	TIPO_INT	13	1	1	31	-1
15	nome	VAR	TIPO_STRING	13	1	0	61	32
16	Set_Id	SUB		13	1	0	-1	0
17	Imprime	SUB		13	1	0	-1	0

FONTE: o próprio autor

5.13.5 Herança

Herança é uma das funcionalidades chaves da orientação a objetos. Com ela, é possível criar uma nova classe (classe derivada) a partir de uma classe existente (classe

base). A classe derivada herda todos os membros da classe base e pode ser complementada com elementos adicionais.

Na Chimera, todo o controle é feito a partir da tabela de símbolos. Quando ocorre a declaração de uma classe derivada, o registro que indica esta classe na tabela de símbolos tem a propriedade “Pai_Heranca” atribuída com o valor da chave da classe base. Para referenciar os membros da classe base, todos os registros referentes a esses membros são duplicados. Quando ocorre essa duplicação, os registros duplicados têm a propriedade “Pai” alterados para referenciar o novo escopo que eles pertencem, neste caso a chave da classe derivada. Essas referências são necessárias para quando um membro da classe derivada precisar acessar os membros de uma classe base, podendo buscar o endereço de memória de uma propriedade ou um rótulo de um método.

Para se declarar uma classe derivada, é preciso adicionar o nome da classe base após a declaração do nome da classe e o caractere ‘:’ (dois pontos).

Como exemplo, são supostos dois caracteres, um professor e um atleta, ambos são pessoas que podem andar e falar, porém individualmente podem ter habilidades específicas. Um professor pode ensinar matemática e um atleta pode correr uma maratona. O programa 5.13.5.1, da Figura 45, demostra como implementar este cenário na Chimera.

Neste exemplo, Pessoa é a classe base, enquanto Professor e Atleta são classes derivadas. A classe Pessoa possuiu duas propriedades, `nome` e `idade`. Ela também possui dois métodos, `Andar()` e `Falar()`. Tanto a classe Professor como a Atleta podem acessar todos os membros de Pessoa. Porém, Professor e Atleta também possuem seus próprios membros, `Ensinar()` e `Correr()` respectivamente. Esses métodos só podem ser acessados por suas próprias classes. Na função Main(), um novo objeto de Professor e Atleta são criados, populados e tem seus métodos acessados. O resultado deste programa pode ser visto na Figura 46.

FIGURA 45 – PROGRAMA 5.13.5.1

```
class Pessoa
    public:
        var string nome;
        var int idade;
        sub Andar()
            println("Eu posso andar.");
        end_sub
        sub Falar()
            println("Eu posso falar.");
        end_sub
        sub Imprime()
            print("Meu nome: ");
            println(nome);
            print("Minha idade: ");
            println(idade);
            Andar();
            Falar();
        end_sub
end_class

class Professor:Pessoa
    public:
        sub Ensinar()
            println("Eu posso ensinar matematica.");
        end_sub
end_class

class Atleta:Pessoa
    public:
        sub Correr()
            println("Eu posso correr uma maratona.");
        end_sub
end_class

sub main()
    var Professor professor;
    var Atleta atleta;

    professor.nome = "Blaise Pascal";
    professor.idade = 42;
    professor.Imprime();
    professor.Ensinar();
    println("");
    atleta.nome = "Dennis Kimetto";
    atleta.idade = 33;
    atleta.Imprime();
    atleta.Correr();
end_sub
```

FONTE: o próprio autor

FIGURA 46 – RESULTADO DO PROGRAMA 5.13.5.1

```
Meu nome: Blaise Pascal
Minha idade: 42
Eu posso andar.
Eu posso falar.
Eu posso ensinar matematica.

Meu nome: Dennis Kimetto
Minha idade: 33
Eu posso andar.
Eu posso falar.
Eu posso correr uma maratona.
```

FONTE: o próprio autor

5.14 POLIMORFISMO

Foi implementada na Chimera o polimorfismo por inclusão. Os polimorfismos de sobrecarga, coerção e paramétrico ficam sugeridos para trabalhos futuros.

5.14.1 Polimorfismo por Inclusão

A implementação do polimorfismo por inclusão foi programada também baseada nos valores registrados na tabela de símbolos. Neste caso, no momento da declaração da classe derivada, quando é feita a duplicação do registro do método da classe base, é verificado se este método também foi escrito na classe derivada. Caso o método exista em ambas as classes, o registro referente ao método da classe base tem a propriedade ‘Ativo’ da tabela de símbolos alterada para *false*, invalidando o registro do método da classe base e sobrescrevendo com o método da classe derivada.

Na Figura 47 pode ser visto o programa 5.14.1.1, no qual se tem o método `Show()` da classe base sobrescrito pelo método `Show()` da classe derivada.

FIGURA 47 – PROGRAMA 5.14.1.1

```
class Base
    public:
        sub Show()
            println("Classe Base");
        end_sub
end_class

class Derivada:Base
    public:
        sub Show()
            println("Classe Derivada");
        end_sub
end_class

sub main()
    var Base b; //Objeto da classe base
    var Derivada d; //Objeto da classe derivada

    b.Show(); //Early Biding
    d.Show();
end_sub
```

```
Resultado:
Classe Base
Classe Derivada
```

FONTE: o próprio autor

5.15 TESTES UNITÁRIOS

Um dos grandes desafios dos programadores é alterar um sistema complexo e garantir que essa alteração não cause um efeito indesejável no sistema, efeito este que é comumente conhecido como *bug*.

A construção da Chimera foi realizada em etapas e era preciso garantir que uma nova etapa não afetasse a etapa já concluída. Porém, realizar os testes manuais a cada etapa seria uma tarefa repetitiva, cansativa, demorada e sujeita a erros humanos.

Por este motivo, foi utilizado o conceito de testes unitários, que nada mais são que programas que executam os testes automaticamente. Esses programas abrem as funcionalidades do sistema, entram com dados e verificam se os resultados obtidos se encontram com o que é esperado pelo sistema. Utilizando testes unitários, foi possível

realizar a programação de cada etapa da Chimera com rapidez e segurança, executando-os a cada alteração necessária.

Com a criação de testes unitários, é possível utilizar outra técnica de programação que é de grande benefício, o desenvolvimento guiado por testes. O desenvolvimento guiado por testes, ou *Test Driven Development* (TDD) em inglês, tem como objetivo guiar o desenvolvedor a programar uma alteração, ou uma nova funcionalidade, já sabendo o resultado esperado através de casos de testes unitários criados previamente.

Tanto os testes unitários como o TDD foram utilizados na programação da Chimera. Utilizando essas técnicas foi possível programar com segurança e garantir que os conceitos do compilador já implementados não fossem alterados, caso esses conceitos necessitam de alterações, bastou adaptar os testes unitários para atender essa alteração.

Os testes unitários foram divididos em duas partes. A primeira parte é utilizada para a verificação da análise léxica e a segunda para a verificação do código intermediário.

Para a validação da análise léxica, é dada uma expressão e uma tabela contendo os *tokens* e lexemas esperados e então são comparados com a tabela *token*/lexema gerada pela análise léxica do compilador. Na Figura 48, é possível visualizar um programa que testa os dígitos e operadores que estão separados por *tabs* e espaços. Uma vez definida a expressão a ser validada, é preciso definir quais são os lexemas e *tokens* esperados dessa expressão. Esses pares de lexemas e *tokens* são empilhados em um *container* e enviados para a função que irá ativar a análise léxica. A tabela resultado da análise léxica é comparada com o *container* definido na função de teste, caso algum resultado esteja incompatível, é disparada uma mensagem com o valor esperado pela análise léxica.

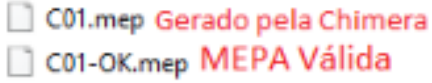
FIGURA 48 – EXEMPLO DE TESTE UNITÁRIO

```
1  //Teste de Digitos e diversos operadores com espaços e tabs
2  void TU_Analise_Lexica::Teste_003()
3  {
4      string expressao = "    587 -       4487874 / 557 * 123 + 9 % 11 -=";
5
6      this->tabelaTeste.clear();
7      this->tabelaTeste.push_back(make_pair("587", NUM_INT));
8      this->tabelaTeste.push_back(make_pair("-", OP_SUBTRACAO));
9      this->tabelaTeste.push_back(make_pair("4487874", NUM_INT));
10     this->tabelaTeste.push_back(make_pair("/", OP_DIVISAO));
11     this->tabelaTeste.push_back(make_pair("557", NUM_INT));
12     this->tabelaTeste.push_back(make_pair("*", OP_MULTIPLICACAO));
13     this->tabelaTeste.push_back(make_pair("123", NUM_INT));
14     this->tabelaTeste.push_back(make_pair("+", OP_ADICAO));
15     this->tabelaTeste.push_back(make_pair("9", NUM_INT));
16     this->tabelaTeste.push_back(make_pair("%", OP_RESTO));
17     this->tabelaTeste.push_back(make_pair("11", NUM_INT));
18     this->tabelaTeste.push_back(make_pair("-=", OP_ATRIBUICAO_SUBTRACAO));
19
20     //Ativa a análise léxica. O resultado da tabela token/lexema é
21     //comparada com a tabelaTeste definida acima.
22     Realizar_Testes(expressao, "TESTE_03");
23 }
```

FONTE: o próprio autor

Para as outras etapas do compilador, os testes estão em conjunto com os testes da geração do código intermediário (MEPA), isso se deve ao motivo da análise sintática, semântica, geração da tabela de símbolos e a geração de código intermediário estarem integrados na Chimera. Neste teste, é realizada a comparação do arquivo contendo o código da MEPA gerado pelo compilador (arquivo .mep) com o arquivo contendo a MEPA esperada (arquivo com final “-OK.mep”), conforme a Figura 49, na qual é demonstrado os arquivos .mep no diretório de compilação do programa C01. A MEPA esperada foi criada, ajustada e testada manualmente e através da geração do código objeto e execução do mesmo, foi possível validar o resultado esperado de um programa.

FIGURA 49 – ARQUIVOS .MEP QUE SÃO COMPARADOS



FONTE: o próprio autor

Na Figura 50, temos a função que realiza essa comparação. Dado um código fonte da linguagem Chimera e a MEPA esperada, a comparação com a MEPA gerada pelo compilador é realizada pela função `CompararArquivos()`, uma mensagem com o resultado da comparação e mostrada após a compilação.

FIGURA 50 – PROGRAMA QUE COMPARA OS ARQUIVOS MEPA

```
1  void TU_Programas::Testar_Programa(string _prog)
2  {
3      string f1, f2;
4      //Arquivo gerado pelo compilador
5      f1 = "Test_Files\\Programas\\" + _prog + ".mep";
6      //Arquivo com a MEPA esperada
7      f2 = "Test_Files\\Programas\\" + _prog + "-OK.mep";
8
9      //Comparar os arquivos
10     if (arquivo.CompararArquivos(f1, f2))
11         cout << _prog << " - OK" << endl;
12     else
13         cout << "*** " << _prog << " - FALHA NA COMPARACAO DOS ARQUIVOS MEPA ***" << endl;
14 }
```

FONTE: o próprio autor

6 ANÁLISE DOS RESULTADOS

A validação da metodologia proposta e desenvolvida neste trabalho deu-se através das seguintes rotinas de testes:

- Testes unitários análise léxica;
- Testes unitários código intermediário e execução do programa;

6.1 TESTES UNITÁRIOS ANÁLISE LÉXICA

Foram criados 63 casos de testes para a análise léxica da Chimera. A Tabela 18 contém o objetivo, expressão e o resultado de sua validação. Todos os *tokens* possíveis para a Chimera foram cobertos pelos testes abaixo.

TABELA 18 – RESULTADOS DA ANÁLISE LÉXICA

#	Objetivo	Expressão	Validação
1	Testar dígitos e sinal de adição	587+999	OK
2	Testar dígitos e operador de adição com espaços e tabs	587 + 999	OK
3	Testar dígitos e diversos operadores com espaços e tabs	587 - 4487874 / 557 * 123 + 9 % 11 -=	OK
4	Testar números reais e sinal de adição	5.87+9.12	OK
5	Testar números reais e sinal de adição com espaços e tabs	5.87 + 9.12	OK
6	Testar caractere	'a'	OK
7	Testar caractere vazio	''	OK
8	Testar String	“Abcdefghijklmnopqrstuvzx”	OK
9	Testar String vazia	""	OK
10	Testar operador maior	>	OK
11	Testar operador menor	<	OK
12	Testar operador maior igual	>=	OK
13	Testar operador maior	<=	OK
14	Testar operador atribuição	=	OK
15	Testar operador igualdade	==	OK
16	Testar operador negação	!	OK
17	Testar operador diferente !=	!=	OK
18	Testar operador resto %	%	OK
19	Testar operador atribuição adição	+=	OK
20	Testar operador atribuição subtração	-=	OK

21	Testar operador atribuição multiplicação	*=	OK
22	Testar operador atribuição Divisão	/=	OK
23	Testar operador atribuição Resto	%=	OK
24	Testar Endereço Elemento	&	OK
25	Testar operador Lógico E	&&	OK
26	Testar operador incremento	++	OK
27	Testar operador decremento	--	OK
28	Testar operador de seleção de elemento por identificador	objeto.propriedade	OK
29	Testar operador de seleção de elemento por ponteiro	objeto->propriedade	OK
30	Testar operador lógico ou	||	OK
31	Testar diversos operadores	"TU_Analise_LexicaTRING"+'C'555&99.99	OK
32	Testar valor do tipo booleano	true false	OK
33	Testar tipo booleano	Bool	OK
34	Testar tipo char	Char	OK
35	Testar tipo float	Float	OK
36	Testar tipo INT	Int	OK
37	Testar tipo void	Void	OK
38	Testar laço do	Do	OK
39	Testar laço for	For	OK
40	Testar laço while	while	OK
41	Testar laço break	Break	OK
42	Testar laço continue	Continue	OK
43	Testar condição IF	If	OK
44	Testar condição ELSE	Else	OK
45	Testar return	Return	OK
46	Testar goto	Goto	OK
47	Testar identificadores	Joao Maria Jose bool5	OK
48	Testar ponto de interrogação e dois pontos	? :	OK
49	Testar ponto e vírgula	;	OK
50	Testar abre e fecha parênteses ()	() ( )) (	OK
51	Testar abre e fecha colchetes []	[] [ ]] [	OK
52	Testar abre e fecha chaves {}	{} { } } {	OK
53	Testar comentário linha //	{5+5} // teste nao deve gerar token : = + ; () bool true int float	OK
54	Testar comentário /* */	{} {}/* \n \t 10 20 30 **** / /* */}{	OK
55	Teste geral, com diversos operadores e identificadores	int numero=14;/* teste diverso */void main(){//teste}	OK
56	Testar palavras reservadas	const var sub string	OK
57	Testar virgula	const, var , sub ,string	OK

58	Testar operador <>	<> < >	OK
59	Testar palavras reservadas	by value ref print println repeat return scan scanln then end_if end_sub end_function to next until loop not or div mod and	OK
60	Testar operador de seleção de elemento por ponteiro, por identificador e por ponteiro novamente	objeto->objeto.objeto->propriedade	OK
61	Testar uma estrutura	struct cliente var int id; var string nome; var bool platinium; end_struct	OK
62	Testar uma classe	class cliente private: var int id; var string nome; var bool platinium; function int calcular_id() return algum_hashing_maluco; end_function public: sub set_cliente(var string _nome by value, var bool _platinium by value) id = calcular_id(); nome = _nome; platinium = _platinium; end_sub function string get_nome() return nome; end_function end_class	OK
63	Testar operador decremento após identificador	return (fibonacci(n-1) + fibonacci(n-2));	OK

FONTE: o próprio autor

6.2 TESTES UNITÁRIOS CÓDIGO INTERMEDIÁRIO E EXECUÇÃO DO PROGRAMA

Para cada programa de teste compilado pela Chimera, foi realizada a compilação do código MEPA para o código objeto e feita a execução do programa. Se o resultado da execução for positivo e com o comportamento esperado, é feita uma cópia manual do arquivo .mep e renomeado para conter a descrição “-OK”, conforme a Figura 49 da seção 5.14.

A lista de casos de testes da Tabela 19 traz os programas que foram compilados pela Chimera, uma breve descrição do que faz o programa, o objetivo do teste e o seu resultado.

TABELA 19 – RESULTADOS EXECUÇÃO DE PROGRAMAS COMPILADOS PELA CHIMERA

#	Arquivo	Descrição do Programa	Objetivo do teste	Resultado
Teste de Programas Procedurais				
1	P01.chi	Programa que soma dois valores dados pelo usuário.	Testar leitura do teclado, operação dos dados imputados e impressão do resultado.	OK
2	P02.chi	Programa que executa uma expressão. (a = 1==1 + 3 * 5)	Testar a função que valida a ordem das expressões e o resultado de comparação entre dois valores.	OK
3	P03.chi	Programa que compara dois valores.	Testar comando de condição IF e comandos de repetição while, repeat e For.	OK
4	P04.chi	Programa que atribui valor a uma constante.	Testar constante e chamada de procedimento.	OK
5	P05.chi	Programa que imprime um valor alterado por um procedimento.	Testar chamada de procedimento com parâmetros passados por valor e por referência. Alteração da variável passada por referência para o procedimento e impressão do valor no procedimento de chamada (main).	OK
6	P06.chi	Programa que retorna o fatorial de 5.	Testar chamada de função, recursividade e retorno da função.	OK
7	P07.chi	Programa que calcula a sequência de Fibonacci para 20 números.	Testar recursividade e retorno de uma expressão.	OK
8	P08.chi	Programa que troca valores de variáveis.	Testar ponteiro.	OK
9	P09.chi	Programa que troca valores de variáveis através de um procedimento.	Testar ponteiros passados como referência.	OK
10	P10.chi	Programa que imprime um nome.	Testar String como parâmetro para um procedimento.	OK
11	P11.chi	Programa que altera valores de variáveis passadas como referência.	Testar passagem por referência para procedimentos e alteração do valor.	OK
12	P12.chi	Programa que passa uma variável por referência indireta para um procedimento.	Testar ponteiro em terceiro nível, chamada de procedimento dentro de outro procedimento com passagem de parâmetro por referência.	Erro. Devido a um não tratamento da MEPA, não foi possível alcançar o objetivo deste teste.

13	P13.chi	Programa que imprime String e inteiro.	Testar passagem de variáveis do tipo String e inteiro por parâmetro a um procedimento.	OK
14	P14.chi	Programa que imprime uma String de 30 caracteres.	Testar o limite da String.	OK
15	P15.chi	Programa que imprime String e inteiro.	Testar passagem de String e inteiro como expressão no parâmetro de um procedimento.	Ok
Teste de Programas contendo Estruturas				
1	S01.chi	Programa que preenche os registros de duas estruturas diferentes e os imprime.	Testar estruturas e o acesso aos seus registros.	OK
2	S02.chi	Programa que preenche os registros de 3 níveis de estrutura.	Testar estrutura dentro de outra estrutura.	OK
3	S03.chi	Programa que imprime dia, mês e ano.	Testar estrutura atribuindo valores e imprimindo os registros.	OK
Teste de Programas Orientado a Objetos				
1	C01.chi	Programa que preenche os atributos de uma classe e imprime.	Testar encapsulamento público, acesso e manipulação dos atributos e impressão.	OK
2	C02.chi	Programa que imprime 2 números.	Testar método e atribuição aos membros da classe.	OK
3	C03.chi	Programa que imprime 2 números.	Testar método e atribuição aos membros privados da classe e impressão dos mesmos por um método da classe.	OK
4	C04.chi	Programa que preenche as propriedades de 3 níveis de classe.	Testar declaração de classe dentro de outra classe e acessar e manipular os seus membros.	OK
5	C05.chi	Programa que dado dois números pelo usuário realiza as quatro operações básicas da matemática.	Testar métodos que retornam valores.	OK
6	C06.chi	Programa que imprime um valor da classe base e derivada.	Testar declaração de uma classe derivada.	OK
7	C07.chi	Programa acessa os métodos da classe base e derivada.	Testar herança e chamada dos métodos corretamente das classes base e derivadas.	OK
8	C08.chi	Programa que imprime uma String.	Testar herança e método da classe derivada que altera atributo da classe base.	OK
9	C09.chi	Programa que imprime duas Strings.	Testar atributo String em classes.	OK
10	C10.chi	Programa com duas instâncias que imprime uma String.	Testar múltiplas instâncias de uma mesma classe.	OK
11	C11.chi	Programa com duas instâncias que imprime número e String.	Testar múltiplas instâncias de uma mesma classe com membros privados e públicos.	OK

12	C12.chi	Programa que imprime duas Strings.	Testar passagem de múltiplas Strings para um método de uma classe.	OK
13	C13.chi	Programa que imprime Strings e Números	Testar passagem de múltiplas Strings e números alternados para um método de uma classe.	OK
14	C14.chi	Programa que imprime uma String.	Testar encapsulamento Protected.	OK
15	C15.chi	Programa que imprime duas Strings.	Testar passagem de múltiplas Strings para um método de uma classe.	OK
16	C16.chi	Programa que imprime um número e uma String.	Testar encapsulamento protegido, privado e público.	OK
17	C17.chi	Programa que imprime 10 valores.	Testar a passagem de múltiplos parâmetros de tipos diferente e atribuição de todos eles a membros da classe.	OK
18	C18.chi	Programa que imprime um inteiro.	Testar encapsulamento privado.	OK
19	C19.chi	Programa que imprime o nome, idade e a profissão.	Testar herança, acesso a atributos na classe base e encapsulamento.	Erro. Devido ao erro da MEPA não estar funcionando com passagem por referência indireta, não foi possível acessar um membro privado de uma classe base através de um *setter*.
20	C20.chi	Programa que imprime o métodos de uma classe base ou derivada.	Testar polimorfismo por inclusão. Sobrescrita de método.	OK
21	C21.chi	Programa que imprime o métodos de uma classe base ou derivada.	Testar polimorfismo por inclusão. Sobrescrita de método. Mais de uma classe derivada.	OK

7 CONCLUSÃO

Criar uma linguagem de programação orientada a objetos a partir de uma linguagem estruturada possui inúmeros desafios e, através deste trabalho, foi possível perceber que cada conceito a ser implementado levanta problemas específicos e casos especiais.

O primeiro passo, que foi a adaptação da Linguagem D+, trouxe complexidades que vieram desde a alteração da gramática para trabalhar com identificadores compostos por dois ou mais elementos até a criação dos membros de uma classe, que podem estar separados conforme a sua especificação de acesso.

O desafio de trabalhar com a MEPA também deve ser destacado pela sua complexidade. A MEPA é uma linguagem intermediária em que não é possível realizar a depuração de código, sendo extremamente custoso identificar qualquer tipo de problema no código intermediário. A leitura deste código intermediário também é difícil, principalmente quando se trabalha com *Strings* longas, no qual é necessário alocar cada caractere como um espaço de memória, ou em funções com múltiplos parâmetros, em que é necessário empilhar todos elas antes da chamada do procedimento.

Tanto os testes unitários como a técnica de desenvolvimento guiado a testes foram fundamentais para a evolução deste trabalho. A cada alteração ou implementação de novas funcionalidades os testes eram executados, garantindo um desenvolvimento mais seguro e ágil da Chimera.

Apesar de todos os desafios citados acima, foi mostrado pela análise dos resultados, que o objetivo principal deste trabalho foi alcançado. Desafio este que só foi possível porque os objetivos específicos também foram concluídos.

Diante das dificuldades encontradas no desenvolvimento do compilador Chimera, assim como os pontos fracos e a falta de implementação de alguns conceitos de orientação a objetos, foram levantadas as seguintes sugestões de melhorias futuras a serem consideradas:

- Alteração da estrutura de dados da tabela de símbolos para não ser necessária a duplicação dos registros em casos de declaração de classes e herança;

- Implementação dos conceitos de polimorfismo: Sobrecarga, Coerção e Paramétrico;
- Implementação de herança múltipla;
- Implementação de tipos genéricos;
- Possibilidade de definir o tamanho de uma *String* na declaração;
- Melhorias na manipulação de *Strings*;
- Ajustar o problema da MEPA ao passar um parâmetro por referência a uma função de segundo nível em diante;
- Criação de um Ambiente de Desenvolvimento Integrado (IDE – *Integrated Development Environment*) para a Chimera;
- Implementação de uma interface para compilação;

REFERÊNCIAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. *Compiladores: Princípios, Técnicas e Ferramentas*. Tradução de: Daniel de Ariosto Pinto. Rio de Janeiro: Editora Guanabara Koogan S.A., 1995.

APPEL, Andrew W.; GINSBURG, Maia Modern Compiler implementation in C. Cambridge, United Kingdom: Cambridge University Press, 1998.

BHOWMIK, Biswajit. *A New Approach of Compiler Design in Context of Lexical Analyzer and Parser Generation for NextGen Languages.* International Journal of Computer Applications 6.11 (2010).

CLARK, Dan. *Beginning C# Object-Oriented Programming*. Second Edition. New York, NY, USA: Apress, 2013.

COOPER, Keith D.; TORCZON, Linda *Engineering a Compiler*. Second Edition. Burlington, MA, USA: Elsevier, Inc. , 2012.

DEMAILLE, Akim; LEVILLAIN, Roland *Compiler Construction as an Effective Application to Teach Object-Oriented Programming.* (s.d.). EPITA Research and Development Laboratory (LRDE), 2008.

DENNIS, Allan; WIXOM, Barbara Haley; TEGARDEN, David *Systems Analysis and Design With UML Version 2.0 – An Object-Oriented Approach.* Third Edition. Danvers, MA, USA: John Wiley & Sons, 2009.

FURLAN, Diógenes Cogo *Gramática D+*. Curitiba: Curso de Bacharelado em Ciência da Computação da Universidade Tuiuti do Paraná, 2016. 3 f. Notas de aula. Disponível através do e-mail: *mestredidi@yahoo.com.br*

GRUNE, Dick *et al*. *Projeto Moderno de Compiladores: Implementação e Aplicações*. Tradução de: Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001.

HOLZNER, Steven *C++ Black Book*. Scottsdale, AZ, USA: The Coriolis Group, 2001.

JAIN, Mahak; SEHRAWAT Nidhi; MUNSI Neha *Compiler Basic Design and Construction.* International Journal of Computer Science and Mobile Computing 10 de Outubro de 2014: 850-852.

JOBS, Steve. *Steve Jobs in 1994: The Rolling Stone Interview*. Entrevistador: Jeff Goodell. Revista Rolling Stone, 2011. Disponível em: <http://www.rollingstone.com/culture/news/steve-jobs-in-1994-the-rolling-stone-interview-20110117>. Acesso em: 9 nov. 2016.

KOFFMAN, Elliot B.; WOLFGANF, Paul A. *Objetos, Abstração, Estruturas de Dados e Projeto Usando C++*. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora S.A., 2008.

KOWALTOWSKI, Tomasz *Implementação de Linguagens de Programação*. Rio de Janeiro: Editora Guanabara Dois, 1983

LOUDEN, Kenneth C. *Compiladores: princípios e práticas.* Tradução de: Flávio Soares Corrêa da Silva. São Paulo: Pioneira Thompson Learning, 2004

MOGENSEN, Torben E. *Introduction to Compiler Design*. Londres: Springer-Verlag London Limited, 2011.

MSDN, Microsoft. Disponível em: <https://msdn.microsoft.com/en-us/library/79b3xss3.aspx>. Acesso em: 11 nov. 2016

MUCHNICK, Steven S. *Advanced Compiler Design and Implementantion.* San Francisco, CA, EUA: Academic Press, 1997

NETO, João José *Introdução à Compilação*. Rio de Janeiro: Livros Técnicos e Científicos Editora S.A., 1987

OLIVEIRA, Hiago Borges de. *Um compilador, uma linguagem de programação e uma máquina virtual simples para o ensino de uma lógica de programação, compiladores e arquitetura de computadores*. Alfenas, 10 de fevereiro de 2014.

PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo *Implementação de Linguagens de Programação: Compiladores Série Livros Didáticos – Número 9 Instituto de Informática da UFRGS.* Porto Alegre, RS, Brasil: Editora Sagra Luzzato, 2001

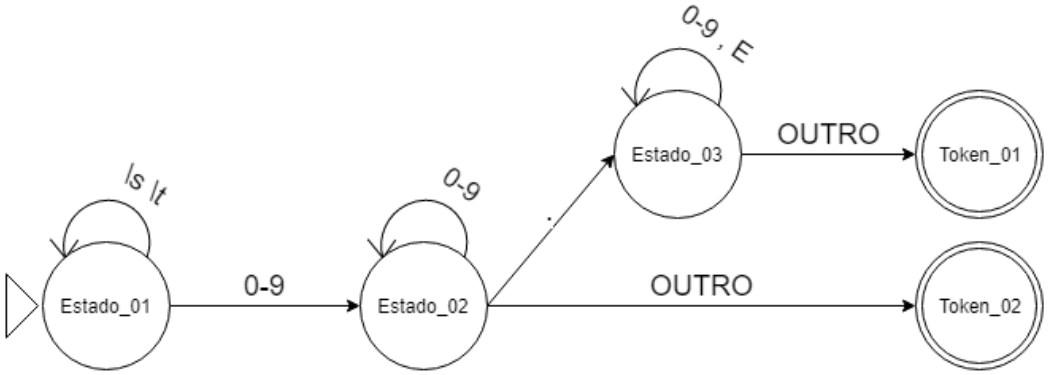
SINGH, Aastha; SINHA Sonam; PRIYADARSHI Archana *Compiler Construction*. International Journal of Scientific and Research Publications 3.4 (2013).

STEFANOV, Stoyan *Object-Oriented JavaScript: Create scalable, reusable high-quality JavaScript applications, and libraries*. Birmingham, United Kingdom: Packt Publishing, 2008.

APÊNDICE A

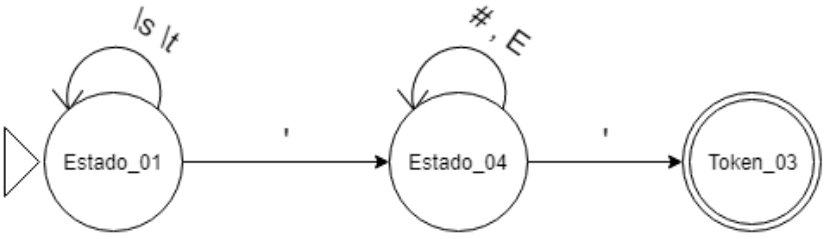
Todos os Autômatos finitos determinísticos (*Deterministic Finite Automaton* - DFA) utilizados neste projeto para definir a análise léxica estão contidos neste apêndice. A legenda com todos os *tokens* gerados pelas transições abaixo estão contidos na Tabela 20.

FIGURA 51 – DFA PARA NÚMEROS INTEIROS E REAIS



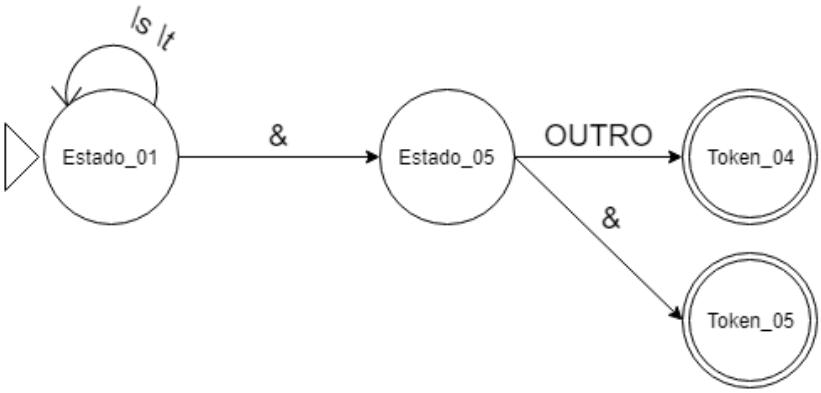
FONTE: o próprio autor

FIGURA 52 – DFA PARA CARACTERES



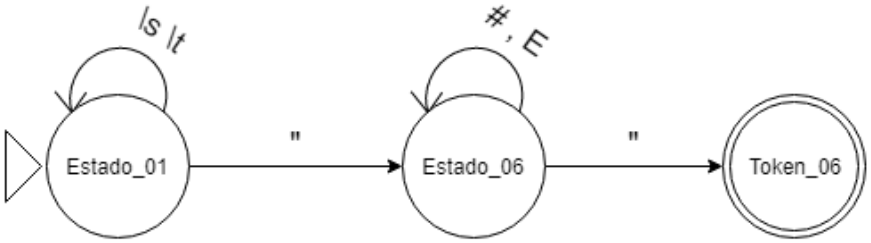
FONTE: o próprio autor

FIGURA 53 – DFA PARA OPERADOR LÓGICO E ENDEREÇO DE ELEMENTO



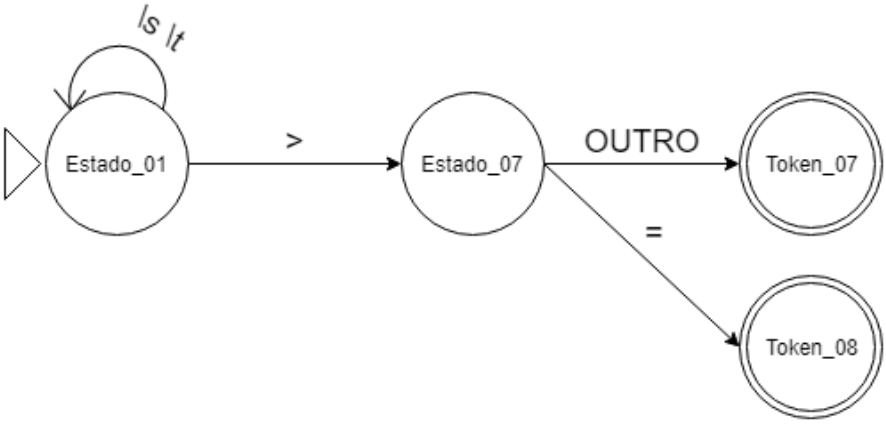
FONTE: o próprio autor

FIGURA 54 – DFA PARA STRING



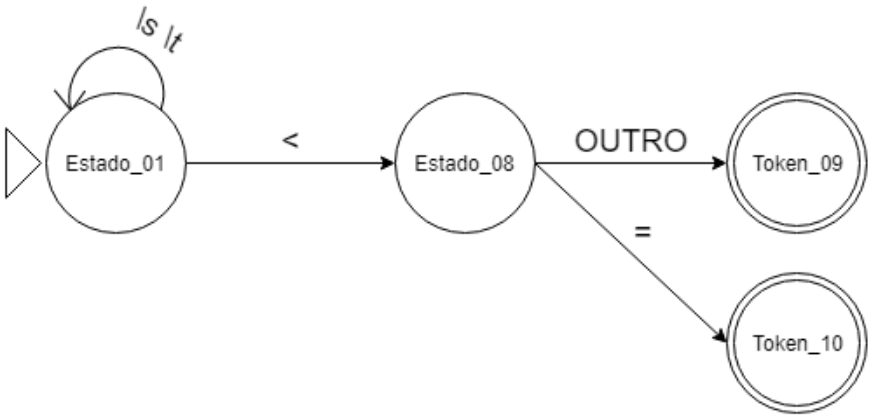
FONTE: o próprio autor

FIGURA 55 – DFA PARA OPERADORES: MAIOR E MAIOR IGUAL



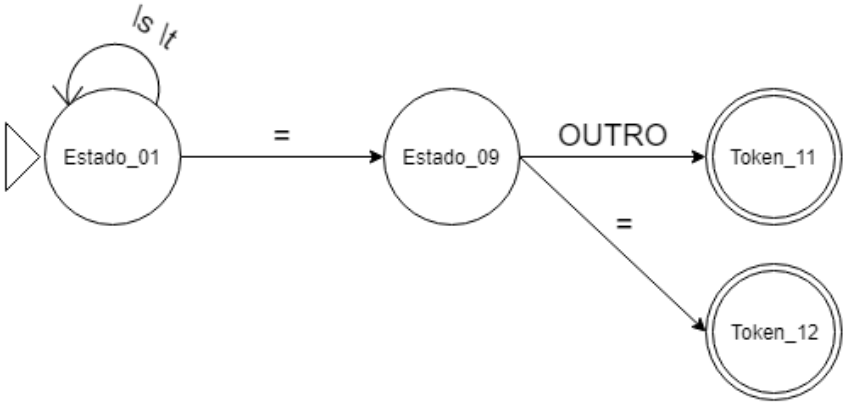
FONTE: o próprio autor

FIGURA 56 – DFA PARA OPERADORES: MENOR E MENOR IGUAL



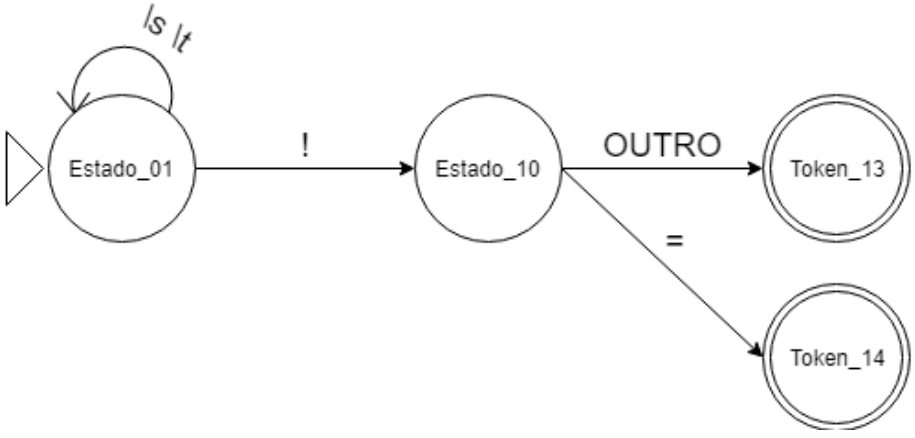
FONTE: o próprio autor

FIGURA 57 – DFA PARA OPERADORES: ATRIBUIÇÃO E IGUALDADE



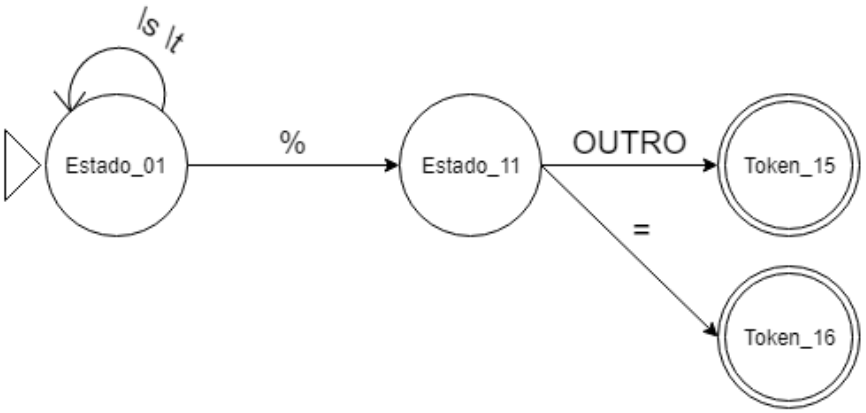
FONTE: o próprio autor

FIGURA 58 – DFA PARA OPERADORES: NEGAÇÃO E NEGAÇÃO IGUALDADE



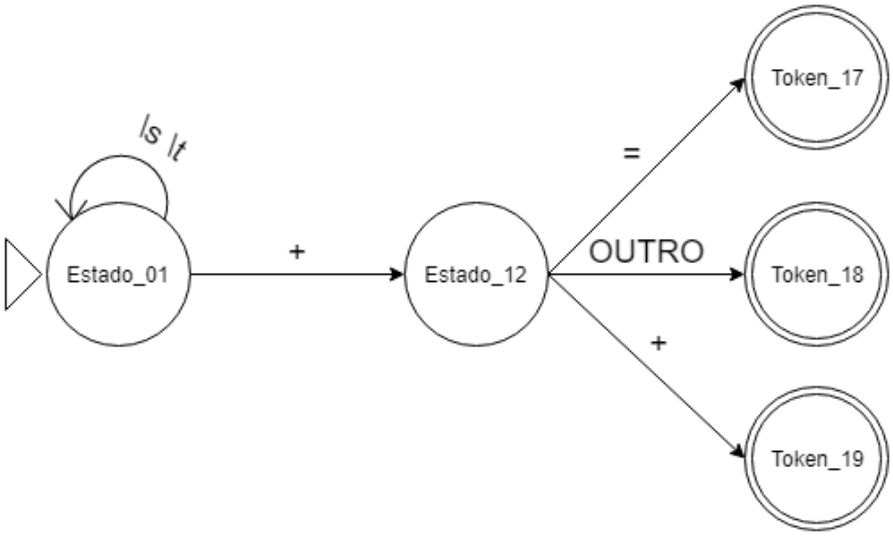
FONTE: o próprio autor

FIGURA 59 – DFA PARA OPERADORES: RESTO E ATRIBUIÇÃO RESTO



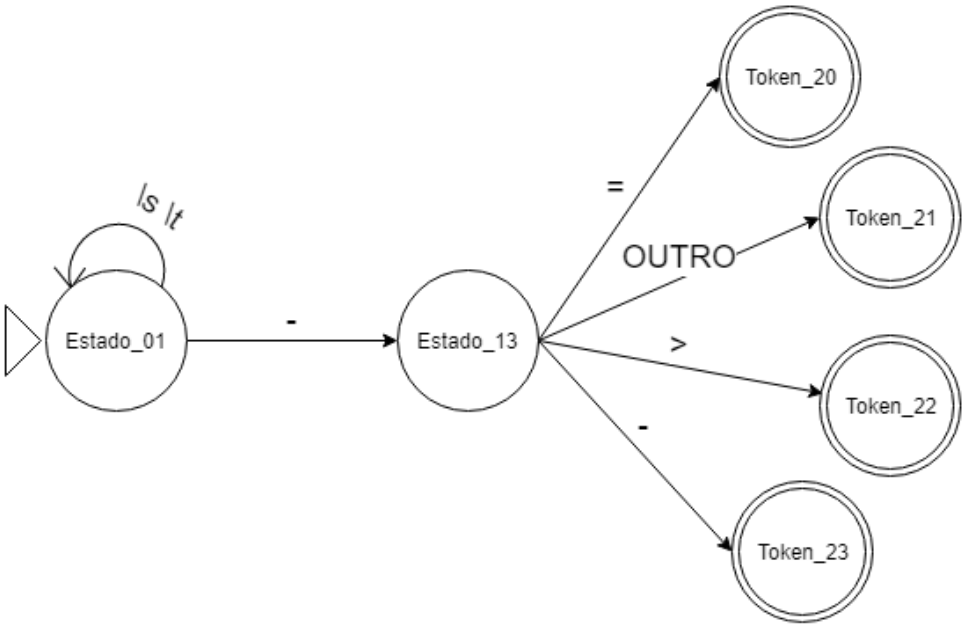
FONTE: o próprio autor

FIGURA 60 – DFA PARA OPERADORES: ATRIBUIÇÃO ADIÇÃO, ADIÇÃO E INCREMENTO



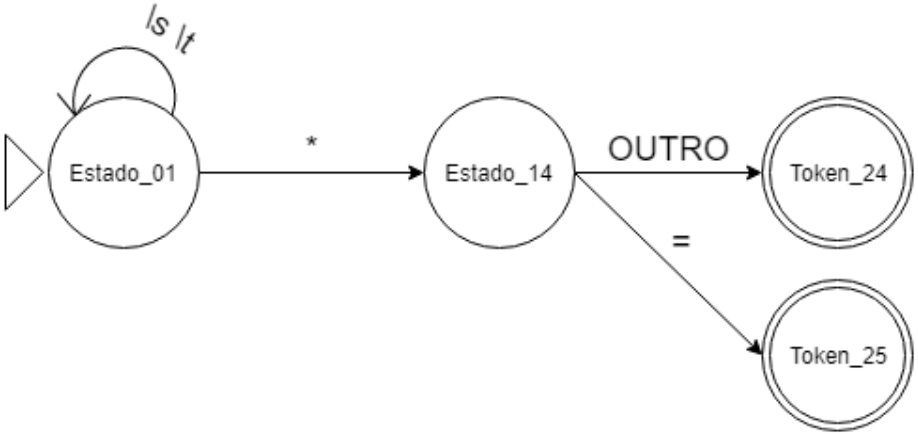
FONTE: o próprio autor

FIGURA 61 – DFA PARA OPERADORES: ATRIBUIÇÃO SUBTRAÇÃO, SUBTRAÇÃO, SELEÇÃO PONTEIRO E DECREMENTO



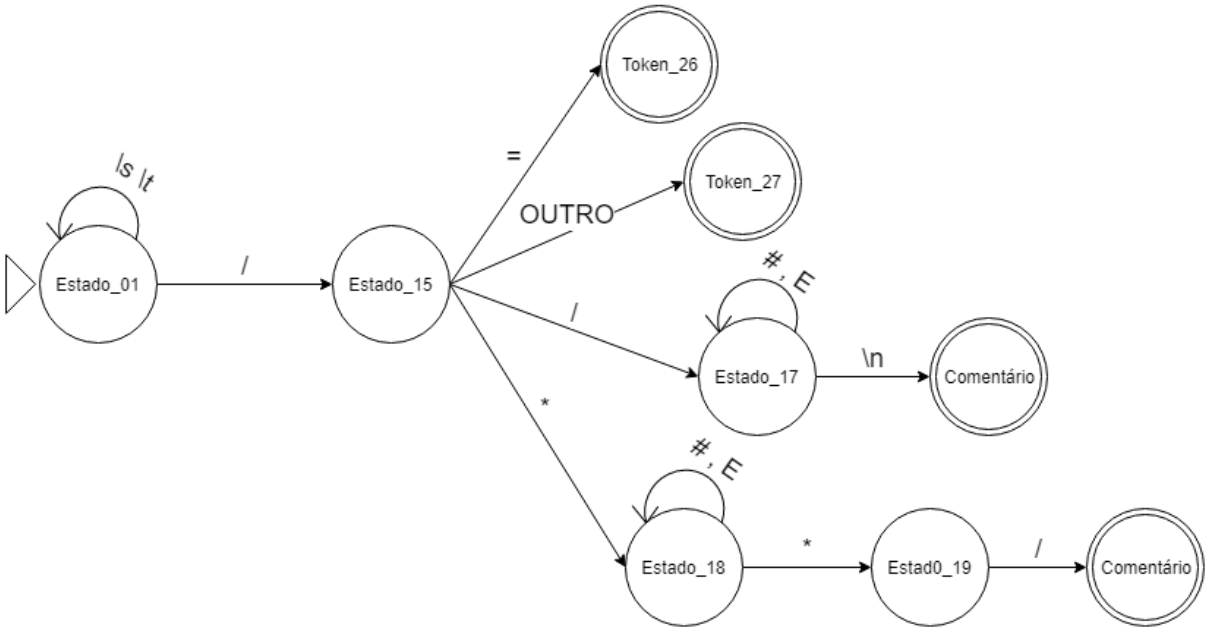
FONTE: o próprio autor

FIGURA 62 – DFA PARA OPERADORES: MULTIPLICAÇÃO E ATRIBUIÇÃO MULTIPLICAÇÃO



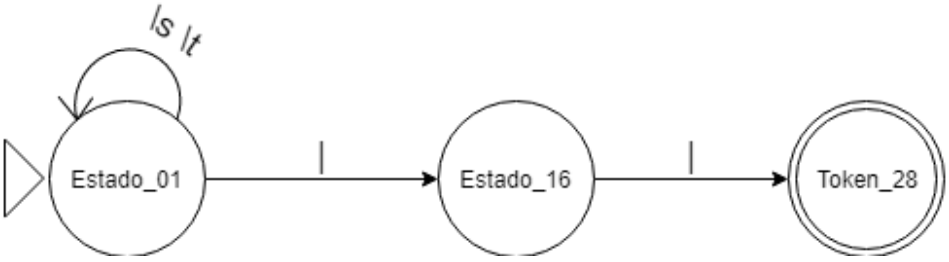
FONTE: o próprio autor

FIGURA 63 – DFA PARA OPERADORES: ATRIBUIÇÃO DIVISÃO E DIVISÃO



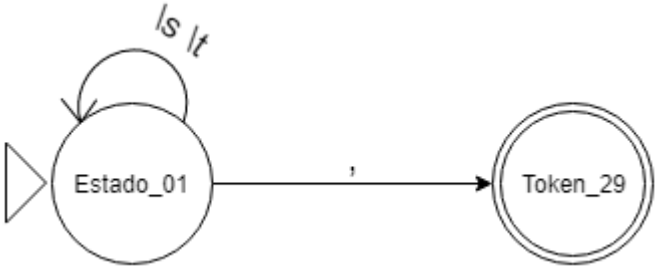
FONTE: o próprio autor

FIGURA 64 – DFA PARA OPERADOR LÓGICO OU



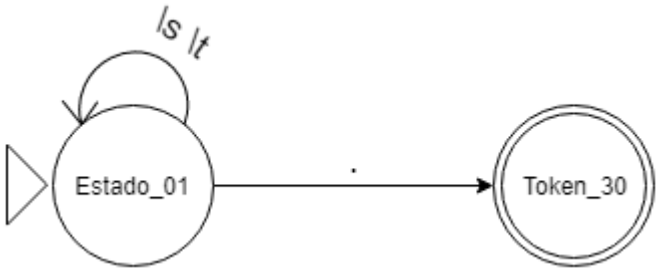
FONTE: o próprio autor

FIGURA 65 – DFA PARA VÍRGULA



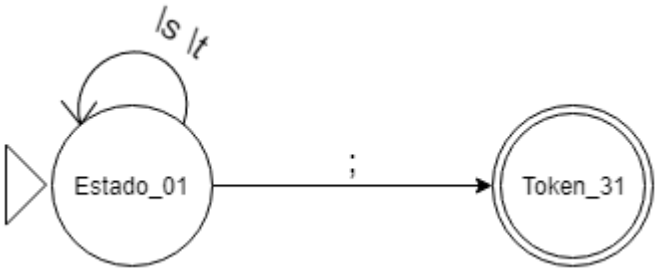
FONTE: o próprio autor

FIGURA 66 – DFA PARA OPERADOR DE SELEÇÃO DE IDENTIFICADOR



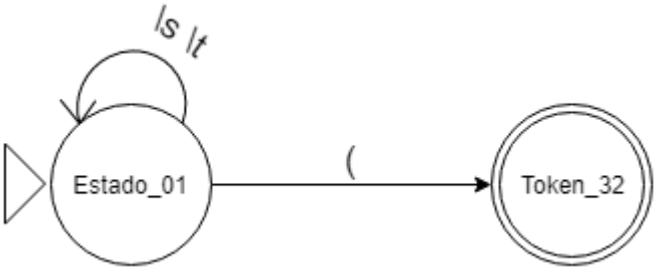
FONTE: o próprio autor

FIGURA 67 – DFA PARA PONTO E VÍRGULA



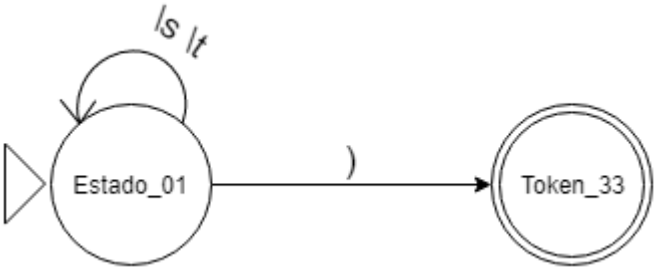
FONTE: o próprio autor

FIGURA 68 – DFA PARA ABRE PARÊNTESES



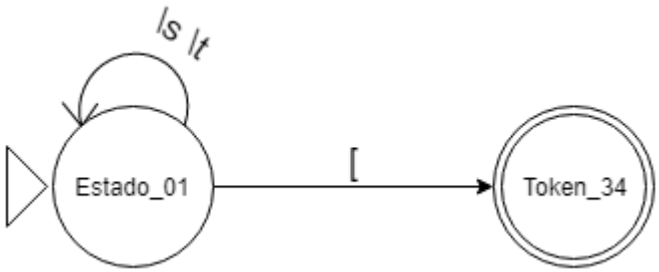
FONTE: o próprio autor

FIGURA 69 – DFA PARA FECHA PARÊNTESES



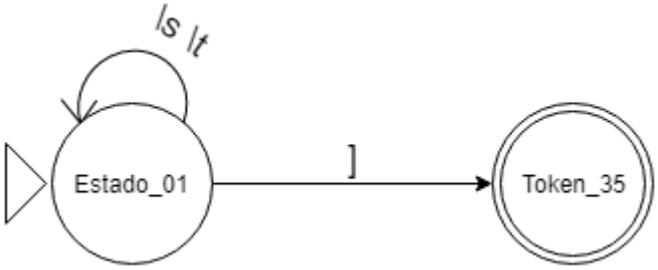
FONTE: o próprio autor

FIGURA 70 – DFA PARA ABRE COLCHETE



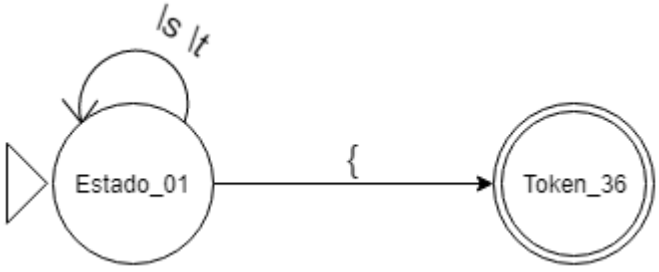
FONTE: o próprio autor

FIGURA 71 – DFA PARA FECHA COLCHETE



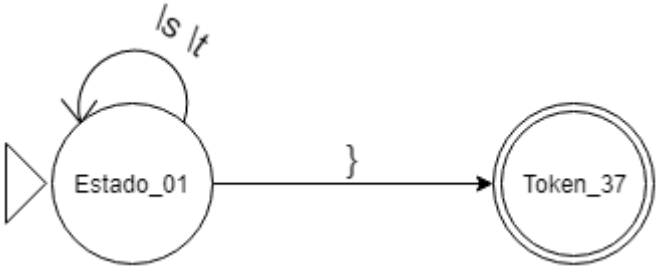
FONTE: o próprio autor

FIGURA 72 – DFA PARA ABRE CHAVES



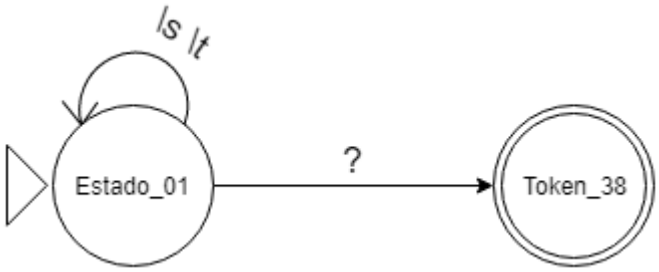
FONTE: o próprio autor

FIGURA 73 – DFA PARA FECHA CHAVES



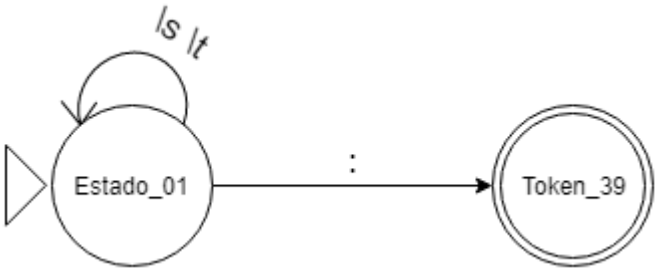
FONTE: o próprio autor

FIGURA 74 – DFA PARA PONTO INTERROGAÇÃO



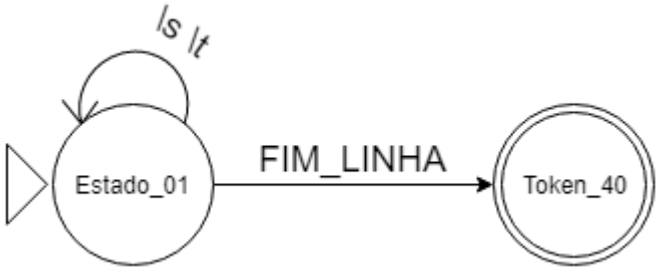
FONTE: o próprio autor

FIGURA 75 – DFA PARA DOIS PONTOS



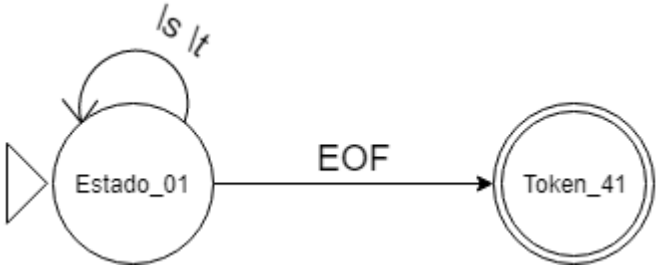
FONTE: o próprio autor

FIGURA 76 – DFA PARA FIM DE LINHA



FONTE: o próprio autor

FIGURA 77 – DFA PARA FIM DO PROGRAMA



FONTE: o próprio autor

TABELA 20 – TABELA DE LEGENDA DOS TOKENS UTILIZADOS NA CHIMERA

Código	Token
T01	NUM_INT
T02	NUM_REAL
T03	CARACTERE
T04	ENDERECO_ELEMENTO
T05	OP_LOGICO_E
T06	STRING
T07	OP_MAIOR
T08	OP_MAIOR_IGUAL
T09	OP_MENOR
T10	OP_MENOR_IGUAL
T11	OP_ATRIBUICAO
T12	OP_IGUALDADE
T13	OP_NEGACAO
T14	OP_NEGA_IGUALDADE
T15	OP_RESTO
T16	OP_ATRIBUICAO_RESTO
T17	OP_ATRIBUICAO_ADICAO
T18	OP_ADICAO
T19	OP_INCREMENTO
T20	OP_ATRIBUICAO_SUBTRACAO
T21	OP_SUBTRACAO
T22	OP_SELECAO_PONTEIRO
T23	OP_DECREMENTO
T24	OP_MULTIPLICACAO
T25	OP_ATRIBUICAO_MULTIPLICACAO
T26	OP_ATRIBUICAO_DIVISAO
T27	OP_DIVISAO
T28	OP_LOGICO_OU
T29	VIRGULA
T30	OP_SELECAO_IDENTIFICADOR
T31	PONTO_VIRGULA
T32	ABRE_PARENTESES
T33	FECHA_PARENTESES
T34	ABRE_COLCHETES
T35	FECHA_COLCHETES
T36	ABRE_CHAVES
T37	FECHA_CHAVES
T38	PONTO_INTERROGACAO
T39	DOIS_PONTOS
T40	FIM_LINHA
T41	FIM_PROGRAMA

FONTE: o próprio autor

APÊNDICE B

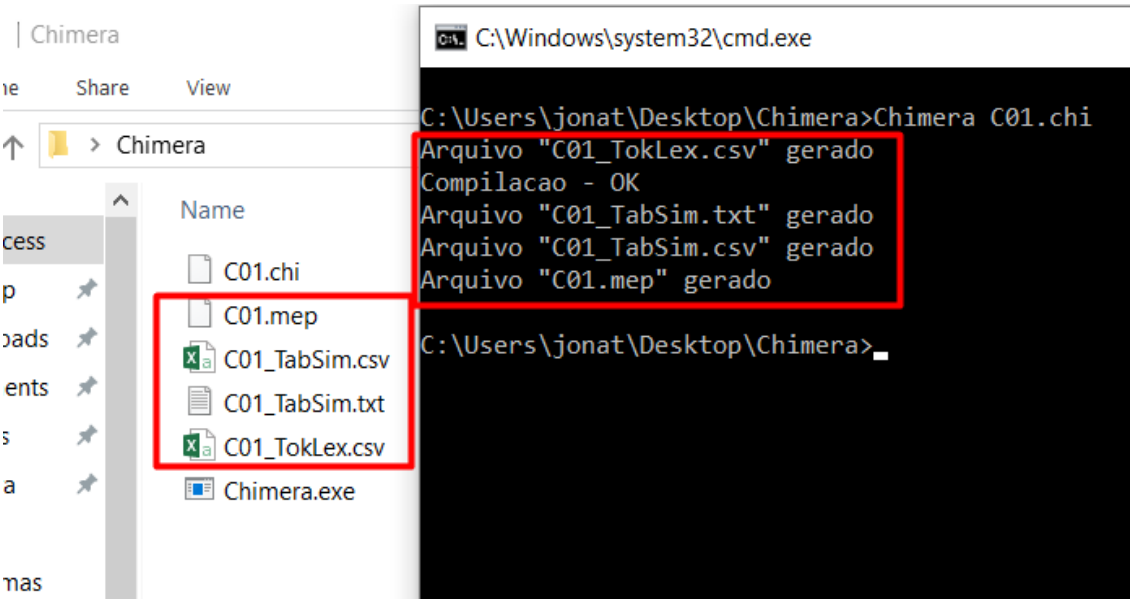
Neste apêndice será demonstrado o passo a passo para a compilação de um programa pela Chimera. Após a compilação, é gerado um arquivo com a extensão .mep no diretório do programa. Este arquivo .mep deve ser compilado pela MEPA através do aplicativo DOSBox, que é uma máquina de 16 bits do DOS. O DOSBox 0.74 está incluso no CD em anexo a este trabalho.

Na pasta do CD \Programas, pode ser encontrado todos os códigos fontes dos programas testados neste projeto. Para o nosso exemplo, utilizaremos o programa C01.chi.

COMPILAR UM PROGRAMA PELA CHIMERA

1) Copiar a pasta D:\Chimera para um ambiente local, no qual D: é o volume do disco;
2) Copiar o arquivo .chi com o programa na linguagem Chimera para dentro deste diretório;
3) Compilar o programa pelo compilador Chimera a partir do Prompt de Comando do Windows, conforme passos descritos abaixo:
    a. Abrir o cmd.exe através do Executar do Windows;
    b. Acessar o diretório que contém o executável Chimera.exe e o arquivo .chi a ser compilado;
    c. Executar o seguinte comando: Chimera nome_arquivo.chi
    d. Verificar se a compilação foi bem-sucedida e se os arquivos foram gerados no diretório, conforme o exemplo da Figura 78, no qual foi compilado o arquivo C01.chi.

FIGURA 78 – RESULTADO DA COMPILAÇÃO DE UM PROGRAMA PELA CHIMERA

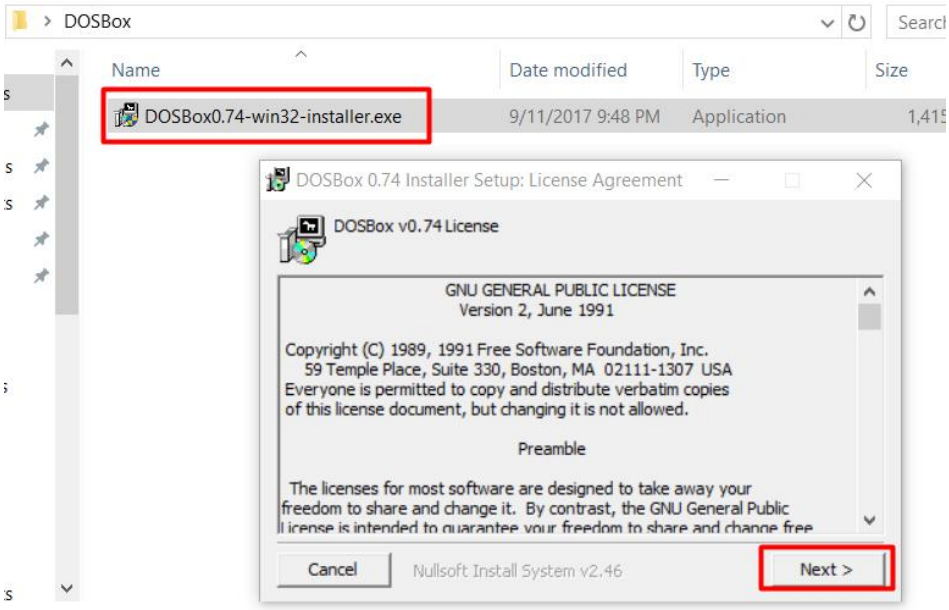


FONTE: o próprio autor

ISNTALAR O DOXBOX 0.74

1) Instalar o programa DOSBox 0.74 que se encontra no diretório D:\DOSBox, conforme Figura 79.

FIGURA 79 – INSTALANDO O DOSBOX 0.74

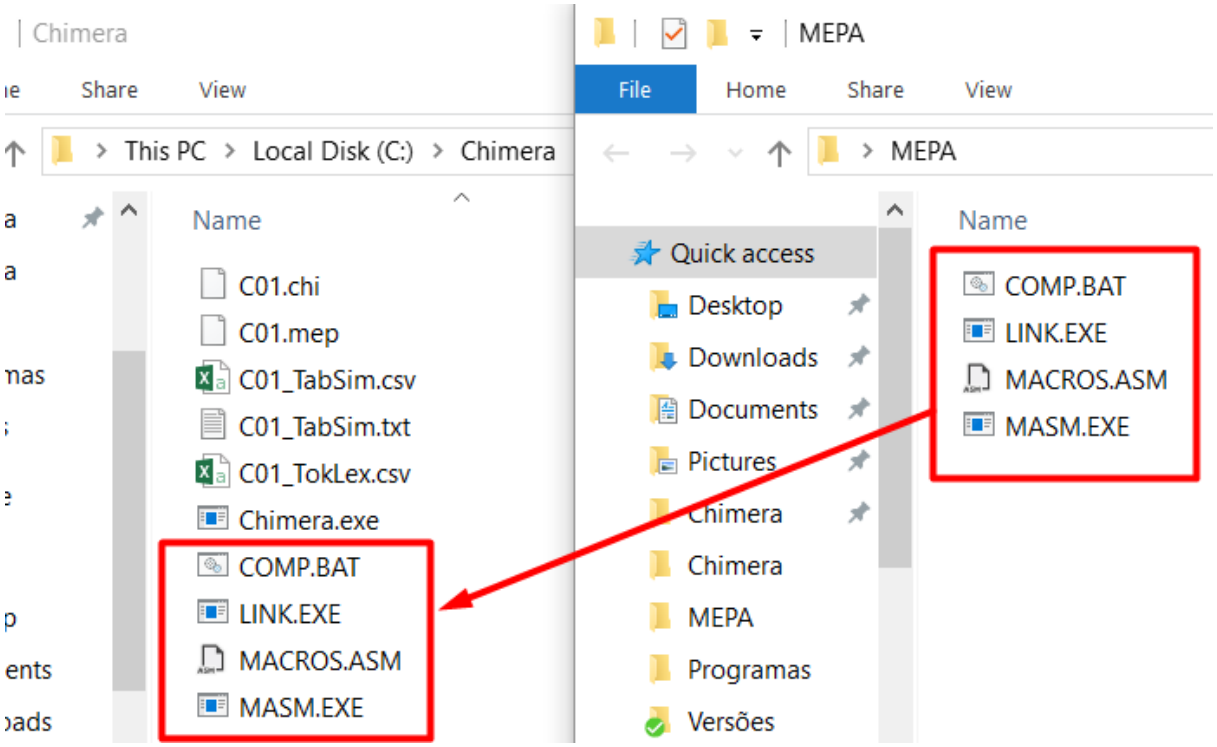


FONTE: o próprio autor

COMPILAR O ARQUIVO C01.MEP E EXECUTAR O PROGRAMA

1) Copiar o conteúdo do diretório do CD D:\MEPA para o diretório local da Chimera, conforme Figura 80;

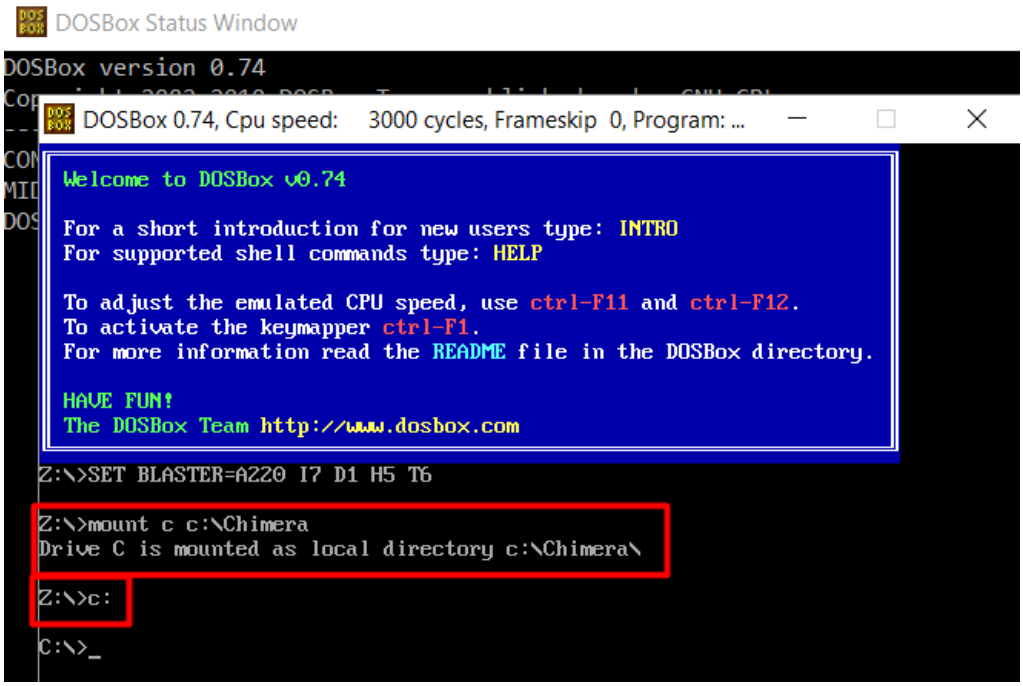
FIGURA 80 – ARQUIVOS DA MEPA



FONTE: o próprio autor

2) Executar o programa DOSBox.exe e realizar os seguintes passos:
    a. Conforme a Figura 81, montar uma pasta no DOSBox através do comando: mount c c:\Chimera. O comando intro mount ensina a montar as pastas;
    b. Acessar a pasta montada através do comando c:

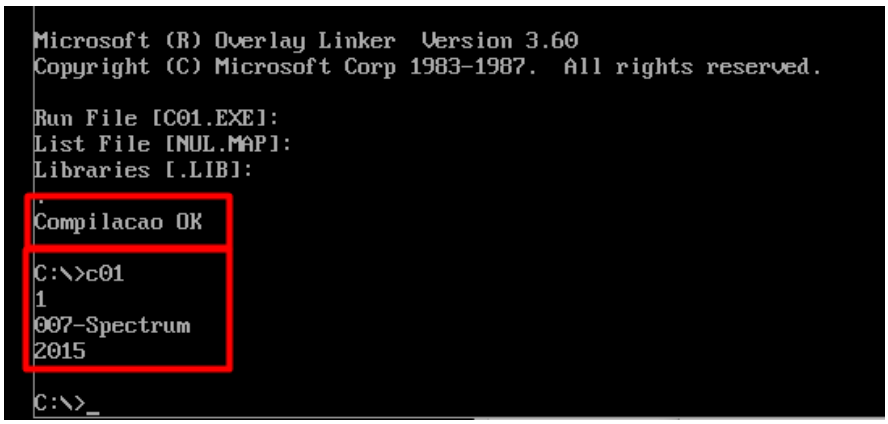
FIGURA 81 – MONTANDO UMA PASTA NO DOSBOX



FONTE: o próprio autor

c. Executar o comando comp nome_arquivo para compilar o arquivo .mep. No nosso exemplo, comp C01
d. Apertar a tecla <enter> no processo de geração;
e. Execute o arquivo EXE gerado: C01 + <enter>
f. O resultado da compilação e da execução do programa podem ser vistos na Figura 82;

FIGURA 82 – COMPILAÇÃO E EXECUÇÃO DO PROGRAMA C01



FONTE: o próprio autor

APÊNDICE C

A sintaxe da Chimera e exemplos de utilização dos comandos da linguagem podem ser consultados neste apêndice como um manual para o usuário.

CLASS

Use class para definir uma classe.

Sintaxe:

```
class IDENTIFICADOR
        ACESSO_MEMBRO
        #Bloco classe
end_class
```

Exemplo declaração:

```
class filmes
        public:
                var int id;
                var string nome;
                var int ano;
end_class
#Declaração e acesso aos membros
sub main()
        var filmes f;
        f.id = 1;
        f.nome = "007-Spectrum";
        f.ano = 2015;
        println(f.id);
        println(f.nome);
        println(f.ano);
end_sub
```

Exemplo com métodos:

```
class filmes
```

```
    private:
        var int id;
    public:
        var int ano;
        sub set_filme(var int _ano by value)
            id = 1;
            ano = _ano;
        end_sub
        sub imprime()
            println(id);
            println(ano);
        end_sub
end_class
#Main
sub main()
    var filmes f;
    f.set_filme(2017);
    f.imprime();
end_sub
```

Exemplo herança:

```
class base
    public:
        var int id_p;
end_class
class derivada : base
    public:
        var int id_c;
end_class
```

FOR

Use o comando for para repetir um loop de instruções até se cumprir uma condição verdadeira**.**

Sintaxe:

```
for EXPRESSÃO to EXPRESSÃO do
        #lista de instruções
next
```

Exemplo:

```
for a=1 to 5 do
        #instruções executadas enquanto a menor ou igual a 5
next
```

FUNCTION

Use essa função para chamar uma sub-rotina executada com suas próprias variáveis locais e que retornam um valor. Os parâmetros podem ser por referência ou por valor, desta forma, os parâmetros passados por referência podem ser modificados pela sub-rotina.

Sintaxe:

```
function TIPO_RETORNO IDENTIFICADOR (LISTA_PARAMETROS)
        #lista de instruções
        return EXPRESSÃO
end_function
```

Exemplo:

```
function int Fatorial(var int n by value)
        if(n==0) then
                return 1;
        else
                return Fatorial (n-1)*n;
        end_if
end_function
```

```
#Chamada
fat = Fatorial(5);
```

Exemplo 2:

```
function int Trocar(var int n by ref)
        n = 1;
end_function
#Chamada
function (&x);
```

IF E ELSE

O comando IF e ELSE são utilizados para verificar uma condição.

Sintaxe:

```
if (EXPRESSÃO) then
        #lista de instruções
else
        #lista de instruções
end_if
```

Exemplo:

```
if (I==1) then
        #instruções executadas se I=1
else
        #instruções executadas se I!=1
end_if
```

PRINT

Use essa função para imprimir um valor.

Sintaxe:

```
print(EXPRESSÃO);
```

Exemplos:

```
Print(“Ola Mundo”);
```

Print(5);

PRINTLN

Use essa função para imprimir um valor e pular uma linha.

Sintaxe:

println(EXPRESSÃO);

Exemplos:

Print(“Ola Mundo”);

Print(5);

REPEAT

Use o comando repeat para repetir um loop de instruções até se cumprir uma condição verdadeira**.**

Sintaxe:

```
repeat
        #lista de instruções
until
        (EXPRESSÃO);
```

Exemplos:

```
repeat
        #instruções a serem executadas se a menor que 5
until
        (a<5);
```

SCAN

Use essa função para ler um valor de entrada.

Sintaxe:

scan(VAR);

Exemplos:

```
var int x;
scan(x);
```

STRUCT

Use struct para criar um conjunto de registros.

Sintaxe:

```
struct IDENTIFICADOR
    #Declarações variáveis
end_struct
```

Exemplos:

```
struct DiaMesAno
    var int dia;
    var int mes;
    var int ano;
end_struct
#Declaração e acesso aos registros
sub main()
    var DiaMesAno dma;
    dma.dia = 20;
    dma.mes = 11;
    dma.ano = 2017;
    print(dma.dia);
    print("/");
    print(dma.mes);
    print("/");
    println(dma.ano);
end_sub
```

SUB

Use essa função para chamar uma sub-rotina executada com suas próprias variáveis locais. Os parâmetros podem ser por referência ou por valor, desta forma, os parâmetros passados por referência podem ser modificados pela sub-rotina.

Sintaxe:

```
sub IDENTIFICADOR (LISTA_PARAMETROS)
        #lista de instruções
end_sub
```

Exemplo:

```
sub Trocar(var int x by value, var int y by ref)
        y = x;
end_sub
#Chamada
Trocar(a, &b);
```

VAR

Use VAR para a declaração de variáveis.

Variáveis podem ser do tipo int, string, char, bool e float.

Sintaxe:

```
var TIPO IDENTIFICADOR;
var TIPO IDENTIFICADOR_1, IDENTIFICADOR_2, IDENTIFICADOR_N;
var pointer TIPO IDENTIFICADOR;
var pointer TIPO IDENTIFICADOR_1, IDENTIFICADOR_2, IDENTIFICADOR_N;
```

Exemplos:

```
var int x;
var String a, b, c;
var pointer int y;
#Exemplo atribuição ponteiro
y = &x;
```

WHILE

Use o comando while para repetir um loop de instruções enquanto uma condição for verdadeira.

Sintaxe:

```
while (EXPRESSÃO) do
        #lista de instruções
loop
```

Exemplos:

```
while (a<5) do
        #Instruções executadas enquanto o valor de a é menor que 5
loop
```