Compiladores

Roselene Henrique Pereira Costa
Alfredo Johnson Rodríguez
Analeia Gomes
João Paulo Resende Rosa
Leandro Ferreira Monteiro
Simone Elza dos Santos Teodoro
Valeria Aparecida Mattar Vilas Boas

Editora Científica

Dados Internacionais de Catalogação na Publicação (CIP)
Selma Alice Ferreira Ellwein – CRB 9/1558

C87c Costa, Roselene Henrique Pereira, et al.

Compiladores. / Roselene Henrique Pereira Costa, Alfredo Johnson Rodríguez, Analéia Gomes, João Paulo Resende Rosa, Leandro Ferreira Monteiro, Simone Elza dos Santos Teodoro, Valeria Aparecida Mattar Vilas Boas. – Londrina: Editora Científica, 2023.

ISBN 978-65-00-68043-0

1. Programação. 2. Linguagens. 3. Ensino Superior. I. Título. II. Autores.

CDD 660

SUMÁRIO

UNIDADE 1 – ESTRUTURA E FUNCIONAMENTO DE UM COMPILADOR

SEÇÃO 1 – ESTRUTURA DE UM COMPILADOR

De acordo com Price e Toscani (2001), a **linguagem de programação** é o meio mais eficaz de comunicação entre pessoas na programação de computadores. Ela serve como uma ponte entre o pensamento humano e a precisão necessária para processamento pela máquina. É mais fácil desenvolver um programa quando a linguagem de programação utilizada está próxima do problema a ser resolvido (linguagem de alto nível), incluindo construções que refletem a terminologia e elementos usados na descrição do problema.

Ainda segundo Price e Toscani (2001) os computadores digitais só entendem sua própria **linguagem de máquina**, que é composta normalmente por sequências de zeros e uns (linguagem de baixo nível). Para tornar os programas escritos em **linguagem de alto nível** operacional, é necessário **traduzi-los** para linguagem de máquina, ou que é feito por meio de **compiladores** ou **interpretadores**.

Tucker e Noonam (2009) dividem os princípios de um projeto de linguagem de programação em três categorias: sintaxe, nomes e tipos e semântica.

A **sintaxe** descreve se o programa foi escrito corretamente, quais palavras e símbolos podem ser usados e como organizá-los. As regras de sintaxe são especificadas pelo formalismo das gramáticas livres de contexto. Os **tipos** existentes em uma linguagem mostram ao programador como implementar as estruturas para armazenar dados e executar cálculos. Há tipos simples e complexos, como listas, árvores, funções e classes. As regras para definições de **nomes** na linguagem compõem um conjunto de normas a serem seguidas. A **semântica** em um programa é o efeito que aquele comando tem sobre os valores envolvidos, por exemplo (Figura 1).

Figura 1 – Exemplo de semântica

```
a = b ;  // para as variáveis  a   e   b   a semântica deve analisar se os tipos
         // das variáveis a   e   b   são compatíveis com as regras da LP
calcule ( n1, n2 ) ;  /*  é uma chamada para a  função calcule
                          a semântica, aqui, deve analisar se método calcule
                          aceita dois parâmetros, se o tipo dos parâmetros
                          e o tipo do valor retornado são compatíveis.
                       */
```

Fonte: Fedozzi (2018.

De acordo com Fedozzi (2018), ao analisar o histórico das linguagens de programação, é possível identificar um padrão comum em alguns deles para solucionar problemas, que são conhecidos como paradigmas de programação. Os quatro paradigmas de programação clássicos (que são descritos no Quadro 1) são:

- Imperativo;
- Orientado a objetos;
- Funcional;
- Lógico.

Quadro 1 – Paradigmas de Programação

Paradigma	Descrição	Exemplo
(1) Imperativo	Variáveis e programas armazenados juntos na memória, além dos comandos e das atribuições, que são usados para cálculos, entradas e saídas. A estrutura e os recursos da linguagem permitem uma transcrição quase direta da solução algorítmica.	FORTRAN, COBOL, C, Basic, ALGOL, Pascal, PHP e Java
(2) Orientada a Objeto	Abstração dos dados e tipos. Pensar em uma solução orientada a objeto é modelar o mundo como peças (objetos) de forma abstrata, sem atribuir valores, e somente depois desenvolver (implementar) a solução, relacionando os diversos objetos e comportamentos.	Smalltalk, C++, C#, Java e Python
(3) Funcional	Linguagens funcionais em crescimento em virtude das aplicações na área de inteligência artificial, pois facilita a programação para sistemas, baseando-se em regras e processamento de linguagem natural.	LISP, ELIXIR, HASKELL
(4) Lógico	Desenvolver a solução declarando qual resultado o programa deve alcançar, em vez de como o programa deve alcançar tal resultado.	Prolog

Fonte: dados da pesquisa.

Tucker e Noonam (2009) acrescentam que esses paradigmas evoluíram ao longo das últimas três décadas.

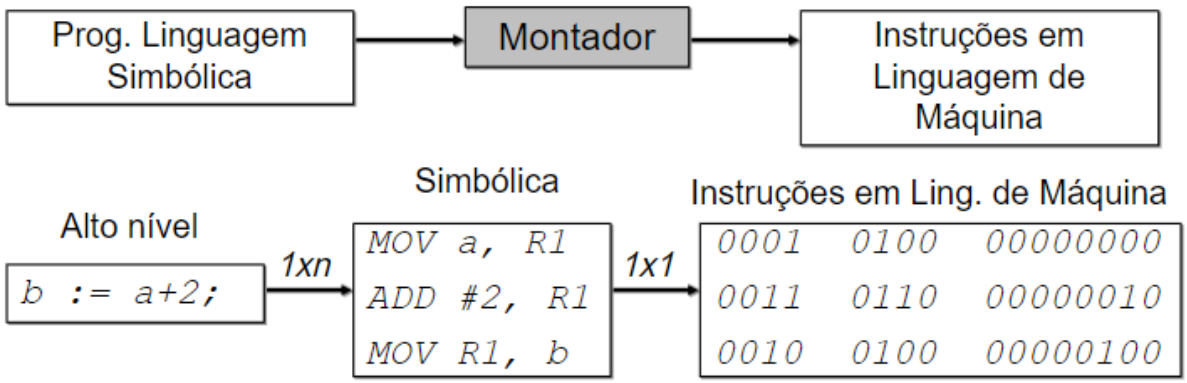
Price e Toscani (2001) explicam que um tradutor é um sistema que recebe como entrada um programa escrito em uma linguagem de programação (linguagem fonte) e gera como resultado um programa equivalente em outra linguagem (linguagem objeto). A evolução das linguagens de programação está diretamente

relacionada aos tipos de tradutores utilizados, os quais podem ser classificados em:

- Montadores;
- Compiladores;
- Interpretadores;
- Compiladores Híbridos.

Os **Montadores** (Figura 2), também conhecidos como Assemblers, são tradutores que convertem instruções escritas em linguagem assistida (Assembly) em instruções na linguagem de máquina correspondente.

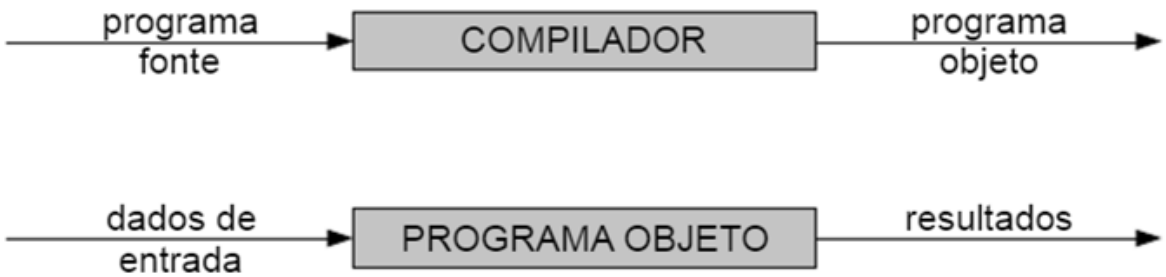
Figura 2 – Montador



Fonte: dados da pesquisa.

Os **Compiladores** (Figura 3) são tradutores que transformam programas escritos em linguagens de alto nível em programas equivalentes em linguagem expressa ou linguagem de máquina. De acordo com Aho (2007), um compilador é um programa que lê um programa escrito em uma linguagem de origem e traduz em um programa equivalente em outra linguagem, a linguagem de destino.

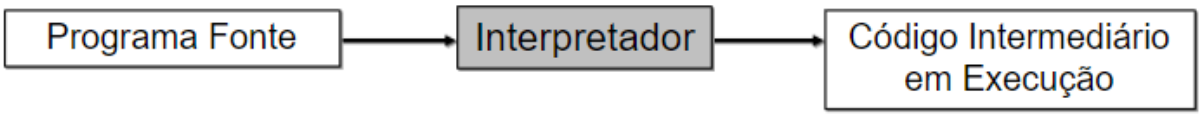
Figura 3 – Compilador



Fonte: dados da pesquisa.

Os **interpretadores** (Figura 4) são tradutores que recebem como entrada o código intermediário de um programa previamente traduzido e com o efeito de execução do algoritmo original sem convertê-lo em linguagem de máquina.

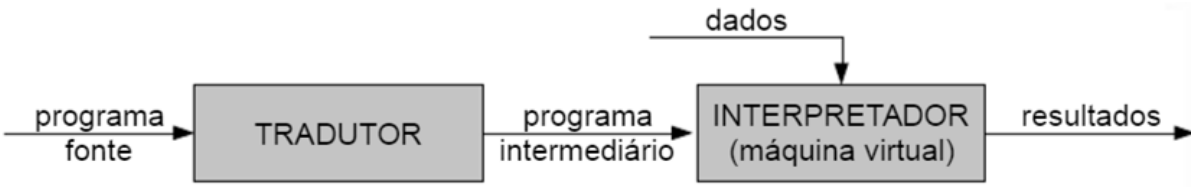
Figura 4 – Interpretador



Fonte: dados da pesquisa**.**

Os compiladores híbridos (Figura5) seguem todos os passos de um compilador tradicional, mas em vez de gerar código executável diretamente, geram um código intermediário que é executado por uma máquina virtual. A principal vantagem desse tipo de compilador é a portabilidade.

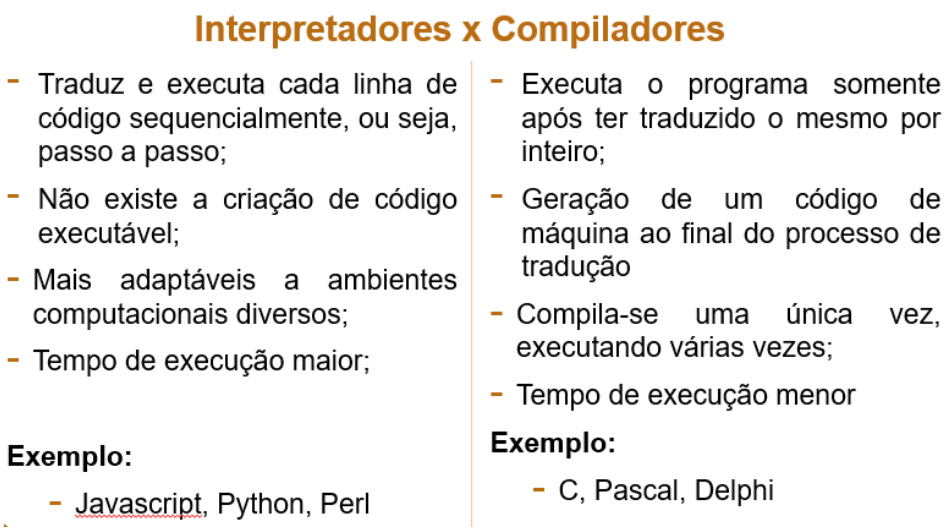
Figura 5 – Compiladores hibridos



Fonte: dados da pesquisa**.**

A Figura 6 apresenta a diferenciação dos interpretadores e dos compiladores.

Figura 6 – Interpretadores x Compiladores

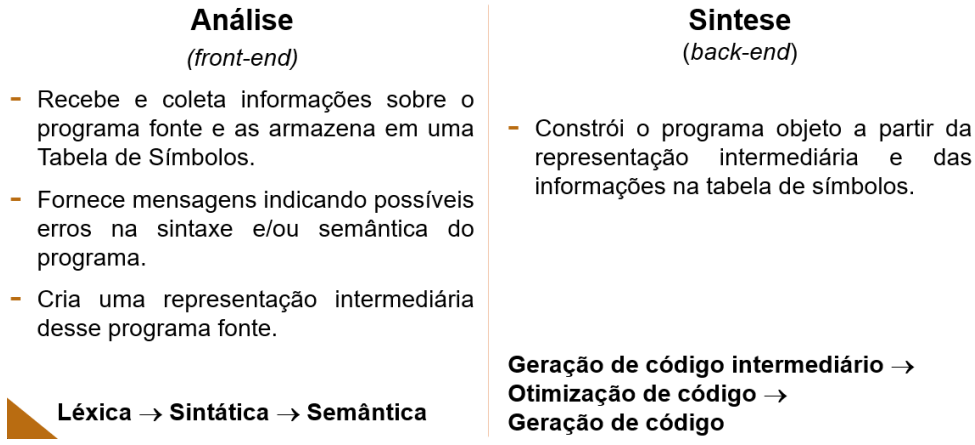


Fonte: dados da pesquisa**.**

.

De acordo com Fedozzi (2018), ao pensarmos nas funções que um compilador deve executar, podemos dividi-las em duas etapas distintas: a fase de análise e a fase de síntese. A Figura 7 apresenta a definição.

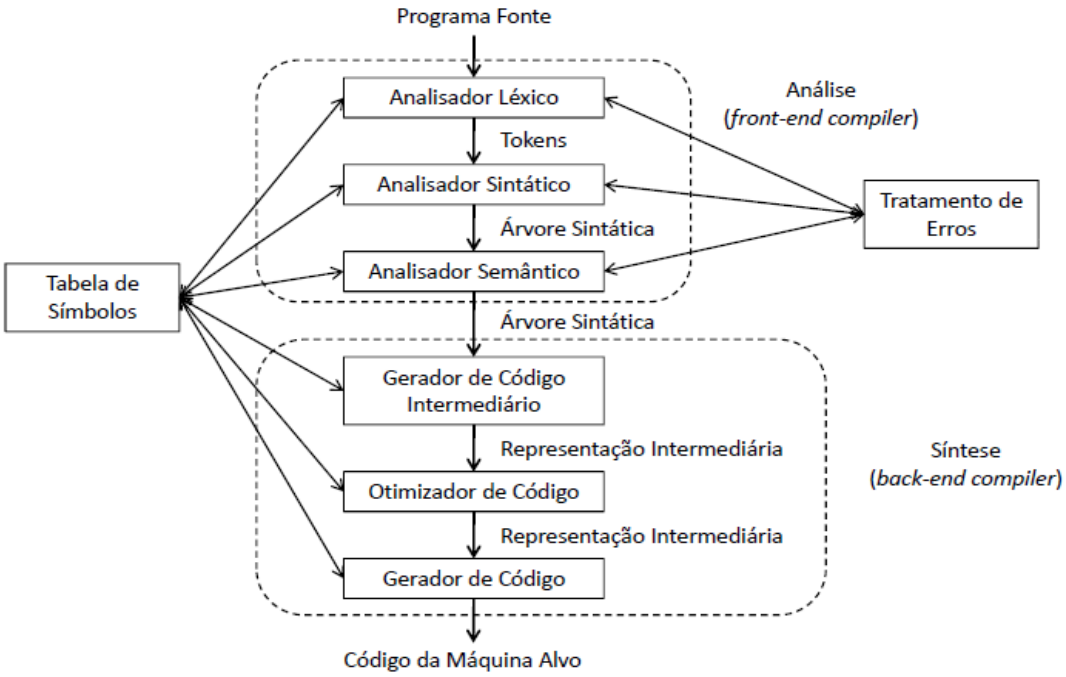
Figura 7 – Análise e Síntese



Fonte: dados da pesquisa**.**

O processo de compilação envolve várias etapas, como pode ser vista na Figura 8, que permitem a transformação do código-fonte de uma linguagem de programação de alto nível em um executável programável.

Figura 8 – Estrutura do compilador



Fonte: adaptado de Price e Toscani (2001).

Na **Análise léxica**, o compilador lê o código-fonte caractere a caractere e identifica os diferentes componentes léxicos, como palavras-chave, símbolos, identificadores, números e literais de strings. A **Análise sintática** envolve a construção da árvore sintática do programa, que representa a estrutura gramatical do código-fonte de acordo com as regras da gramática da linguagem de programação. Essa fase também inclui a detecção de erros de sintaxe, como uma falta de ponto e vírgula ou um parêntese mal fechado. Na fase de **Análise semântica**, o compilador verifica se as instruções do programa estão corretas em relação às regras da linguagem de programação, tais como tipos de dados, escopos de variáveis, entre outros. O compilador também pode realizar otimizações nesta fase, como simplificar expressões matemáticas.

Na etapa de síntese temos a **Geração de código intermediário que** envolve a tradução do código-fonte em um formato intermediário, como código de três circuitos ou árvore de expressão. Este código intermediário é mais fácil de ser processado e otimizado pelo compilador, e pode ser usado como entrada para outras ferramentas de compilação. Na **Otimização de código**, o compilador analisa o código intermediário e aplica várias técnicas para melhorar o desempenho e reduzir o

tamanho do programa. Essas técnicas incluem eliminação de código redundante, simplificação de expressões, reordenação de instruções e uso de registradores para armazenar variáveis. E por fim, a **Geração de código** o compilador gera o código objeto ou código de máquina a partir do código intermediário otimizado. Este código pode ser executado diretamente pelo processador da máquina alvo. O compilador também pode gerar informações de depuração que ajudam os programadores a identificar problemas no código executado.

UNIDADE 1 – ESTRUTURA E FUNCIONAMENTO DE UM COMPILADOR

SEÇÃO 2 – FUNDAMENTOS DE LINGUAGENS FORMAIS

Segundo Fedozzi (2018), é essencial compreender a definição de uma **linguagem formal**, o que envolve o estudo dos seus elementos básicos. Toda **linguagem** tem sua origem em um **alfabeto**, que consiste em um conjunto finito e não vazio de símbolos. A **concatenação** dos símbolos desse alfabeto gera uma **palavra**, e um conjunto de palavras formadas a partir desse alfabeto é o que compõe uma **linguagem**.

Linguagens formais são um conjunto de regras que definem uma estrutura de dados abstratos, ou seja, são uma representação formal de uma linguagem. As linguagens formais são importantes na Ciência da Computação e na Teoria da Linguagem, sendo utilizadas para modelar diversas aplicações, como a criação de compiladores e interpretadores, processamento de linguagens naturais, reconhecimento de padrões, entre outras.

As **gramáticas** formais são um tipo de linguagem formal que define uma sintaxe para uma determinada linguagem. Elas são compostas por um conjunto de regras que especificam como palavras ou símbolos podem ser combinadas para formar sentenças válidas na linguagem. Existem vários tipos de gramáticas, que são classificados de acordo com as suas regras e sua capacidade de gerar certos tipos de linguagens.

Uma **gramática** é uma construção que consiste em regras de produção ou regras de reescrita, que permitem gerar todos os elementos de uma linguagem a partir de um símbolo inicial. Formalmente, uma gramática G é definida como {N, Σ, P, S}, onde:

- N é um conjunto de símbolos auxiliares, chamados de símbolos não-terminais.
- Σ é um conjunto de símbolos terminais, que são os elementos básicos da linguagem.
- P é um conjunto de regras de produção ou regras de reescrita que especificam como os símbolos não-terminais e terminais podem ser combinados para gerar elementos da linguagem.
- S é o símbolo inicial, que é um dos símbolos não-terminais e a partir do

qual se inicia a geração da linguagem.

A **derivação** é um processo de substituição de símbolos de acordo com as regras definidas por uma gramática. A partir de um símbolo inicial, a derivação consiste em aplicar as regras da gramática para produzir novas sequências de símbolos, até que se obtenha uma sentença válida na linguagem definida pela gramática. Pode ser utilizado o símbolo de relação $\Rightarrow$ (deriva em um passo).

Vamos utilizar a gramática como exemplo: G = {N, $\sum$, P, S } = onde:

- N = { S } é o conjunto de não terminais,
- $\sum$ = { 0, 1 } é o conjunto de terminais,
- P = { S $\rightarrow$ 0S1, S $\rightarrow \varepsilon$ } = { S $\rightarrow$ 0S1 | ε } é o conjunto de regras.

Para mostrar que a cadeia 000111 faz parte da linguagem associada à gramática, seguimos, a partir de S, os seguintes passos intermediários: S, 0S1, 00S11, 000S111, 000111.

Quanto a derivação, podemos representar:

S $\Rightarrow$ 0S1

0S1 $\Rightarrow$ 00S11

00S11 $\Rightarrow$ 000S111

000S111 $\Rightarrow$ 000111

Podemos também representar de forma mais compacta:

S $\Rightarrow$ 0S1 $\Rightarrow$ 00S11 $\Rightarrow$ 000S111 $\Rightarrow$ 000111

A **linguagem** gerada por uma gramática é o conjunto de todas as sentenças que podem ser geradas a partir das regras da gramática. Essa linguagem pode ser finita ou infinita, dependendo das regras da gramática. As linguagens geradas por gramáticas são importantes para a compreensão e construção de compiladores, interpretadores e outras aplicações relacionadas à teoria da computação.

Definimos a linguagem da gramática G = {N, $\sum$, P, S} por **L(G) = { x $\rightarrow \sum^*$ | S $\Rightarrow^*$ x }**.

Podemos afirmar que a linguagem gerada pela gramática G é formada por sequências x compostas exclusivamente de símbolos terminais, as quais podem ser iniciadas a partir do símbolo inicial S em um número indefinido de etapas de derivação, utilizando-se as regras de produção presentes em P Em outras palavras, G determina

uma linguagem por meio das suas regras de derivação, a partir de um conjunto finito de símbolos terminais e não terminais.

A **Hierarquia de Chomsky** é uma classificação das gramáticas formais de acordo com sua capacidade de gerar certos tipos de linguagens. Ela é composta por quatro tipos, que podem ser vistos no Quadro 1.

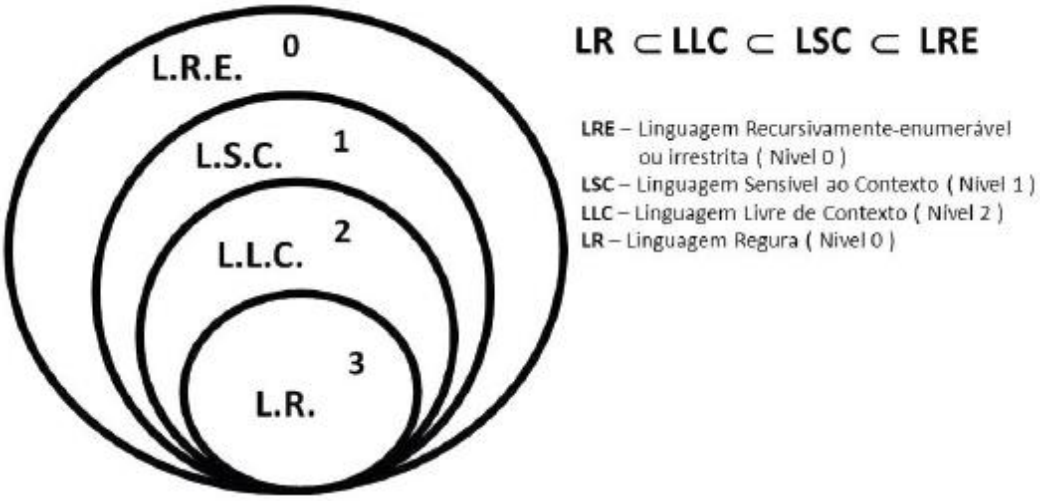
Quadro 1 – Hierarquia de Chomsky

Tipo	Gramática	Linguagem
0	Gramática Irrestrita	Linguagens Recursivamente Enumeráveis
1	Gramática Sensível ao Contexto	Linguagens Sensíveis ao Contexto
2	Gramática Livre de Contexto	Linguagens Livres de Contexto
3	Gramática Regular	Linguagens Regular

Fonte: elaborada pela autora.

Segundo Fedozzi (2018) o trabalho desenvolvido pela teoria generativa, a **Hierarquia de Chomsky** e a tese de **Turing-Church** proporcionaram aos futuros projetistas de linguagens e compiladores a possibilidade de criar linguagens novas e desenvolver programas que analisem sua sintaxe, sabendo se elas são escritas de acordo com o conjunto de regras gramaticais definidas. A maioria das linguagens de programação pode ser desenvolvida com linguagens de nível 2, não sendo necessário passar para os níveis 1 e 0, respectivamente, Linguagem Sensível ao Contexto (LSC) e Linguagem Recursivamente Enumerável (LRE). A Hierarquia Chomsky define as estruturas das linguagens como um processo crescente entre os tipos de línguagens, sendo: **LR ⊂ LLC ⊂ LSC ⊂ LRE**, veja a Figura 9.

Figura 9 – Gráfico representativo da Hierarquia de Chomsky



Fonte: Fedozzi (2018).

Segundo Fedozzi (2018), a **Hierarquia de Chomsky** mostra que as linguagens vão desde as mais simples até as mais complexas. A **linguagem** mais simples é composta basicamente de **palavras**. Quando começamos o processo de aprender a escrever, um dos primeiros passos é reconhecer o alfabeto. Após essa etapa, passamos à grafia das palavras, que nada mais é do que a concatenação (justaposição) dos símbolos do alfabeto. Dando continuidade ao processo de escrita, analisamos as palavras que obedecem às regras definidas pela **gramática**, o que, nas línguas naturais, costumamos chamar de grafia correta. A esse nível da **linguagem**, Chomsky associou o **nível 3** da hierarquia e o chamou de **Linguagem Regular**. Após a grafia correta das palavras, avançamos um pouco mais no domínio de uma língua, construindo sentenças onde a ordem das palavras deve ser analisada, análise denominada **análise sintática** da sentença. Noam conseguiu demonstrar que esse é um padrão na maioria das linguagens formais e designou esse nível como 2, chamado de **Linguagens Livres de Contexto**.

Conforme Fedozzi (2018) quando atingimos o **nível 2** de uma linguagem formal, somos capazes de escrever uma "redação" e, se associarmos essa analogia às **linguagens de programação**, poderíamos dizer que um programa de computador é uma "redação". Afinal, um programa de computador nada mais é do que um conjunto de comandos (frases) sintaticamente corretos. As **Linguagens Sensíveis ao**

Contexto (LSC), designadas como **nível 1**, geram especificações mais longas e complexas, o que dificulta seu uso prático, ou seja, sua implementação. Portanto, quando criamos **especificações gramaticais** com o objetivo de construir compiladores, na prática, adotamos o LLC e utilizamos a análise dirigida à sintaxe para a implementação de dependências de contexto no processo de **compilação**, essa verificação consiste na **análise semântica**. Os idiomas de **nível 0** são as **linguagens recursivamente enumeráveis** (LRE), esse nível abrange todos os tipos de linguagem e é conhecido como estrutura frasal ou irrestrito.

De acordo com Fedozzi (2018), a especificação da gramática antes do colaborador da BNF (Backus-Naur Forms) era pouco legível e prática. Para solucionar esse problema, no final dos anos 50, dois investigadores, Jonh Backus e Peter Naur, desenvolveram uma notação formal para LLC, denominada BNF. Conforme Tucker (2009), a BNF foi adaptada da teoria de Chomsky e se tornou simbólica de gramática livre de contexto, sendo generalizada para especificação de linguagens de programação.

A **EBNF** (Extended Backus-Naur Form) é uma notação puxada da forma de Backus-Naur, que é uma notação formal usada para descrever a sintaxe de linguagens de programação. A notação EBNF é uma extensão da BNF, que adiciona recursos para expressar ações semânticas e restrições na estrutura da linguagem. Ela é frequentemente utilizada na construção de gramáticas para linguagens de programação, em que as regras definidas especificam como as construções da linguagem podem ser combinadas para formar programas válidos.

A notação EBNF é composta por símbolos terminais, não terminais e operadores, que são usados para definir regras de produção. Os símbolos terminais são letras, números ou símbolos que aparecem literalmente em um programa escrito em uma determinada linguagem. Símbolos não-terminais representam construções sintáticas mais complexas, como expressões ou declarações, que podem ser construídas a partir de símbolos terminais e outros símbolos não-terminais. Os operadores, por sua vez, são usados para descrever a relação símbolos entre terminais e não-terminais, como alternativas, repetições e agrupamentos.

A utilização da notação EBNF é fundamental no processo de construção de compiladores, já que a gramática de uma linguagem de programação é a base para a construção de um compilador. Ao definir a gramática de uma linguagem em EBNF,

é possível criar uma especificação formal para a linguagem, permitindo que sejam escritos programas que sigam essa especificação. Além disso, o compilador pode usar a gramática definida em EBNF para verificar se um programa escrito em uma determinada linguagem está sintaticamente correto, ou seja, se ele segue as regras definidas pela gramática. Por meio da notação EBNF, é possível especificar a estrutura de uma linguagem de programação, desde construções simples como números e letras, até construções mais complexas como declarações de variáveis, expressões e comandos. Além disso, um EBNF permite especificar a precedência e associatividade dos operadores, a ordem de avaliação de expressões, as regras para aninhamento de blocos e muitos outros aspectos importantes da sintaxe de uma linguagem de programação.

A Figura 10 ilustra a gramática do nome de variáveis em C++ na notação EBNF.

Figura 10 – Gramática para nome de variáveis em C++, usando notação EBNF

Solução EBNF	Comentários
<*var*> ::= ('_' | <*letra*>) { '_' | <*letra*> | <digito> }	Palavra inicia por '_' ou <*letra*>
	Os demais símbolos poderão ser '_' ou <*letra*> ou <*digito*>
	Por estarem entre chaves '{ }' podem repetir 0 ou *n* vezes.
<*letra*> ::= 'a' | 'b' | 'c' | ... | 'z'	<*letra*> representa símbolos terminais, no caso as letras do alfabeto
<*digito*> ::= '0' | '1' | '2' | '3' | ... | '9'	<*digito*> produção que representa símbolos terminais, no caso os números de 0 a 9

Fonte: Fedozzi (2018).

Dessa forma, a EBNF é uma ferramenta essencial na construção de compiladores e interpretadores, pois permite especificar de forma clara e precisa a estrutura sintática de uma linguagem de programação, permitindo a construção de programas que sigam essa estrutura.

UNIDADE 1 – ESTRUTURA E FUNCIONAMENTO DE UM COMPILADOR

SEÇÃO 3 – PLANEJAMENTO DA CONSTRUÇÃO DE UM COMPILADOR E A SELEÇÃO DE FERRAMENTAS

O planejamento da construção de um compilador envolve várias etapas, desde a análise dos requisitos da linguagem a ser compilada até a implementação e teste do compilador. A seguir, é apresentado alguns exemplos das etapas envolvidas nesse processo:

Definição da linguagem de programação: Antes de começar a construir o compilador, é preciso definir a linguagem de programação que será compilada. Por exemplo, pode-se definir uma linguagem de programação para jogos em que os programadores possam especificar regras de jogo, gráficos e física do jogo. Outro exemplo, se desejamos criar um compilador para a linguagem de programação C, precisamos estudar a sintaxe e as regras da linguagem para entender como ela deve ser interpretada e transformada em código de máquina.

Especificação da gramática: A próxima etapa é a especificação da gramática da linguagem fonte. Isso envolve a definição de todas as regras sintáticas de linguagem. Por exemplo, uma regra sintática da linguagem C é uma declaração de variável, que pode ser especificada na gramática da seguinte forma:

<declaration> ::= <type> <identifier> "=" <expression>

<type> ::= "int" | "float" | "char"

<identifier> ::= <letter> { <letter> | <digit> }

<expression> ::= <identifier> | <literal> | <expression> "+" <expression>

Essa especificação indica que uma declaração deve começar com um tipo, seguido de um identificador, seguido do sinal de igual e uma expressão. O tipo pode ser "int", "float" ou "char". O identificador é uma sequência de letras e dígitos. Uma expressão pode ser um identificador, um literal (como "10" ou "3.14") ou uma expressão aritmética envolvendo dois operandos e o operador "+".

Implementação do analisador léxico: A próxima etapa é a implementação do analisador léxico, que é responsável por ler o código fonte e gerar uma sequência de tokens. Por exemplo, se o código fonte for: int x = 10;

O léxico analisador deve gerar os seguintes tokens:

TOKEN_TYPE_INT

TOKEN_IDENTIFIER (valor: "x")

TOKEN_ASSIGN

TOKEN_LITERAL_INT (valor: 10)

TOKEN_SEMICOLON

Implementação do analisador sintático: A próxima etapa é a implementação do analisador sintático, que é responsável por verificar se a sequência de tokens gerados pelo analisador léxico está de acordo com a gramática especificada. Por exemplo, se a entrada do compilador for: int x = 10;

O analisador sintático deve reconhecer essa sequência de tokens como uma declaração válida de uma variável inteira chamada "x" com o valor inicial de 10.

Implementação do gerador de código: A próxima etapa é a implementação do gerador de código, que é responsável por gerar o código objeto ou código intermediário a partir da sequência de tokens reconhecidos pelo analisador sintático. Por exemplo, para uma declaração "int x = 10;", o gerador de código pode gerar o seguinte código intermediário:

ALLOCATE x, 4

STORE_CONST 10, x

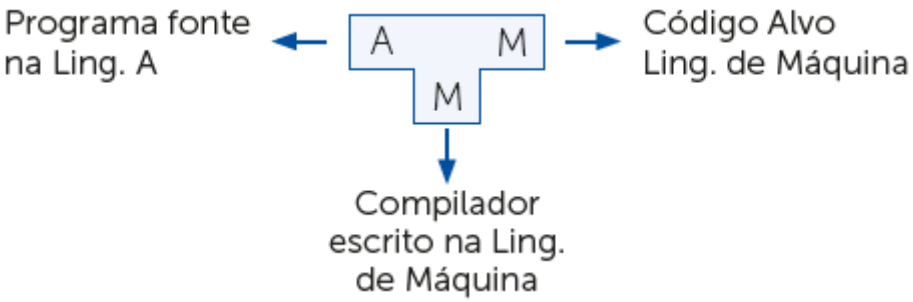
Essas são apenas algumas das etapas envolvidas na construção de um compilador. O processo completo pode ser muito mais complexo, dependendo da fonte da linguagem e das funcionalidades do compilador.

De acordo com Fedozzi (2018), o primeiro compilador foi escrito em código de máquina, o que tornava o processo de desenvolvimento trabalhoso, mas tinha como vantagem a rapidez e servia como base para todos os outros por meio do processo de bootstrapping. Esse processo consiste em escrever um compilador na linguagem de máquina M para a linguagem A, e depois, com o compilador escrito em M, escrever um novo compilador para A. Dessa forma, seguindo esses passos, é possível obter o código da máquina a partir de um compilador feito na própria linguagem.

Segundo Fedozzi (2018), os **compiladores** construídos com essa técnica são conhecidos como **autocompiláveis**, pois em alguma fase da sua construção utilizam a mesma linguagem de programação na qual foram implementados. Além disso, existem também os **cross-compilers**, que usam a técnica de **bootstrapping** e são escritos em um ambiente, mas rodam em outro.

Fedozzi (2018) exemplifica o processo de **bootstrapping** com o uso de um diagrama em T (Figura 11). No exemplo, o primeiro compilador para a linguagem A foi escrito em código de máquina.

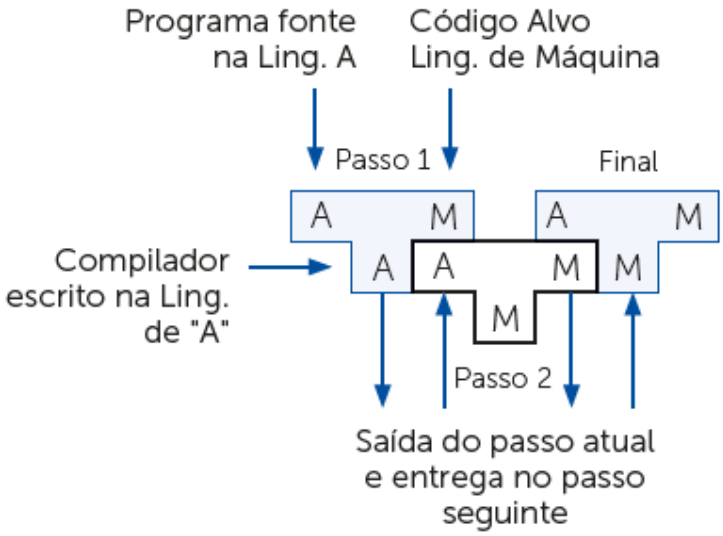
Figura 11 – Primeiro compilador para linguagem 'A'



Fonte: Fedozzi (2018).

Em seguida, deseja-se escrever um compilador para A na própria linguagem A. Ao aplicar o processo de bootstrapping, é possível obter o código alvo M a partir do compilador para A escrito em M. Logo, é possível obter o código alvo M a a partir da entrada do programa fonte A e do compilador escrito na própria linguagem A, com o auxílio do primeiro compilador criado para a linguagem A, desenvolvido em M. Veja o passo a passo do processo na Figura 12.

Figura 12 – Exemplo de bootstrapping



Fonte: Fedozzi (2018).

É importante considerar que quanto mais passos forem necessários para alcançar o código alvo, mais demorada será a compilação.

As ferramentas de apoio para o desenvolvimento dos compiladores estão divididas em:

- Geradores de analisadores léxicos (scanners)
- Geradores de analisadores sintáticos (parsers)
- Geradores de código

Os **geradores de analisadores léxicos**, também conhecidos como scanners, são ferramentas que auxiliam na construção do analisador léxico de um compilador. Esses geradores recebem uma descrição formal da linguagem em que o compilador está sendo desenvolvido e geram o código fonte do analisador léxico em uma linguagem de programação escolhida pelo desenvolvedor. Essa descrição formal pode ser feita utilizando a notação regular ou a EBNF, por exemplo. Exemplos de geradores de analisadores léxicos incluem o Flex e o JFlex.

Os **geradores de analisadores sintáticos**, também conhecidos como parsers, auxiliam na construção do analisador sintático de um compilador. Esses geradores recebem uma descrição formal da gramática da linguagem em que o compilador está sendo desenvolvido e geram o código fonte do analisador sintático em uma linguagem de programação escolhida pelo desenvolvedor. Essa descrição formal pode ser feita utilizando a notação BNF, a EBNF ou outra linguagem formal de especificação de gramáticas. Exemplos de geradores de analisadores sintáticos incluem o Yacc, o Bison e o ANTLR.

Os **geradores de código** auxiliar na geração do código de saída de um compilador. Eles recebem como entrada uma descrição da árvore de sintaxe abstrata (AST) gerada pelo analisador sintático e gerado o código em uma linguagem de programação alvo. Essa descrição da AST pode ser feita utilizando alguma linguagem de especificação de AST, como o XML ou o JSON. Exemplos de geradores de código incluem o LLVM e o GCC.

UNIDADE 2 – ESPECIFICAÇÃO DA ANÁLISE LÉXICA E TÉCNICAS DE IMPLEMENTAÇÃO

SEÇÃO 1 – ANÁLISE LÉXICA

A **análise léxica** é a primeira fase do processo de compilação em que a entrada do compilador, normalmente um programa em uma linguagem de programação, é conduzida e transformada em uma sequência de tokens (ou lexemas) que representam unidades léxicas com significado da linguagem.

Essa fase é realizada pelo analisador léxico, também conhecido como scanner, que percorre o programa de entrada e identifica cada lexema presente. Um lexema pode ser um identificador, uma palavra-chave, um operador, um número, um caractere especial, entre outros elementos da linguagem. Por exemplo, considere a Figura 13 com o trecho de código em C++.

Figura 13 – Código em C++

```
int main() {
    int x = 10;
    std::cout << "O valor de x é: " << x << std::endl;
    return 0;
}
```

Fonte: dados da pesquisa.

O analisador léxico irá transformar cada elemento significativo em um token, como por exemplo: int, main, (,), {, int, x, =, 10, ;, std, ::, cout, <<, "O valor de x é: ", <<, x, <<, std, ::, , endl, ;, return, 0, , ;.}

Esses **tokens** serão usados pelo analisador sintático para verificar a conformidade do código com a gramática da linguagem e produzir a árvore sintática. A análise léxica é uma etapa importante no processo de compilação, pois garante que o compilador entenda corretamente a estrutura do programa e possa produzir um código executável.

De acordo com Fedozzi (2018) a análise léxica lê os caracteres de entrada, gera um fluxo de dados, os tokens e os disponibiliza para o analisador sintático. O analisador sintático analisa o fluxo de dados gerado de acordo com a GLC e, quando necessário, aciona a análise semântica para verificar se os elementos presentes na

estrutura sintática são compatíveis. Se em cada uma dessas etapas não for encontrado erro de escrita no programa fonte, o fluxo de dados será enviado para a fase seguinte, a de síntese, caso contrário, os erros de cada etapa da análise serão reportados ao programador, e o processo de compilação será abortado.

Conforme Aho et al. (2007), uma análise léxica é responsável por transformar o fluxo de caracteres do código fonte em uma sequência de tokens que serão utilizados pela análise sintática.

É importante ressaltar, segundo Christensson (2009), que em programação um token é um elemento individual da linguagem de programação. Como visto na unidade anterior, os tokens são identificados pelos nomes dados às tradições da gramática, ou seja, pelos padrões reconhecidos pela gramática.

Conforme Fedozzi (2018) a função exata do analisador léxico é reconhecer os **tokens** associados às expressões regulares, ou seja, o subconjunto da linguagem livre de contexto pertencente apenas às linguagens regulares, composto pelos elementos básicos, tais como identificadores, operadores, constantes, comentários, caracteres especiais e tipos compostos. Na prática, é o analisador sintático (parser) que aciona o analisador léxico (lexer) para saber se a palavra lida na frase é válida para a linguagem, logo o analisador sintático depende da resposta do analisador léxico.

De acordo com Aho et al. (2007), uma técnica comum para construir um analisador léxico é criar um diagrama que represente a estrutura dos tokens da linguagem fonte e, em seguida, implementar um programa que seja capaz de identificar esses tokens. Essa implementação pode ser feita diretamente ou por meio da utilização de padrões para especificação léxica, juntamente com geradores de analisadores léxicos, como o Yacc ou o JFLEX.

Conforme Fedozzi (2018) quando tratamos de análise léxica, devemos encontrar no programa fonte os padrões correspondentes ao par (tipo do token, lexema). Observe a o exemplo da Figura 14, as linhas 01 e 02 foram escritas na linguagem C:

Figura 14 – trecho de código

```
1   int x = 0 ;
2   x = 10 + 1b * x
```

Fonte: elaborada pela autora.

De acordo com o código apresentado, identificamos os seguintes padrões para cada par (tipo do token, lexema):

(<tipo de dado>, int)

(<variável>, x)

(<símbolo de atribuição>, =)

(<constante numérica>, 0)

(<terminador>, ;)

(<constante numérica>, 10)

(<operador aritmético>, +)

(Não reconhecido, 1b)

(<operador aritmético>, *)

Cada par (token, lexema) deve ser armazenado em uma tabela dinâmica de símbolos para futuras consultas pelas demais fases da compilação.

Conforme Fedozzi (2018) antes de iniciar a análise dos tokens, é possível remover os espaços em branco necessários. Porém, é necessário identificar todos os símbolos terminais para que os espaços em branco não sejam excluídos de forma ilimitada. Observe o código da Figura 15.

Figura 15 – Exemplo de uma classe em Java

```
1 public class Exemplo {
2     public static void main(String[] args){
3             int a = 10;
4             int b = 20;
5             int soma ;
6             soma = a - ( b * 8 + 56 ) ;
7             System.out.println( "soma = " + soma);
8     }
9 }
```

Fonte: Fedozzi (2018).

A remoção de espaços em branco do código fonte em Java pode tornar mais fácil a identificação dos tokens pelo analisador léxico, conforme ilustrado na Figura 16.

Figura 16 – Classe em Java sem espaços, escrita em uma mesma linha

```
public class Exemplo2{public static void
main(String[] args){int a=10;int b=20;int
soma;soma=a-(b*8+56);System.out.println("soma =
"+soma);}}
```

Fonte: Fedozzi (2018).

Conforme Fedozzi (2018) o segundo passo do analisador é o reconhecimento dos tokens. Nessa fase é realizada a leitura caractere a caractere, o que consome um tempo razoável, e há momentos em que o analisador léxico precisar verificar vários caracteres à frente para concluir o reconhecimento do token. Esse processo de armazenar dados e depois retroceder impõe uma sobrecarga no processamento, então a técnica de bufferização foi desenvolvida para obter uma melhor eficiência neste o processo.

A **técnica de bufferização** é uma técnica utilizada na análise léxica de compiladores para otimizar o processo de leitura de caracteres de entrada. Em vez de ler um caractere por vez da entrada e analisá-lo imediatamente, a técnica de bufferização permite que uma sequência de caracteres seja lida e armazenada em um buffer antes que o analisador léxico comece a identificar os tokens.

Isso é feito por meio da criação de um buffer, que é uma região da memória que armazena uma quantidade pré-definida de caracteres da entrada. O analisador léxico lê essa sequência de caracteres do buffer e, em seguida, identifica os tokens correspondentes, o que reduz o número de leituras da entrada.

A técnica de bufferização pode ser integrada de diversas formas. Uma forma comum é a utilização de uma função de leitura de caracteres, que lê uma quantidade fixa de caracteres do arquivo de entrada e armazena-os em um buffer. Em seguida, o analisador léxico lê os caracteres do buffer e, ao atingir o final do buffer, a função de leitura é chamada novamente para preencher o buffer com mais caracteres.

Essa técnica é útil em linguagens de programação que possuem

sequências de caracteres longos e complexos, como identificadores, literais e comentários, que podem ser lidos de uma vez e armazenados em um buffer. Isso reduz a sobrecarga de leitura da entrada e aumenta a eficiência do compilador.

Segundo Fedozzi (2018) os elementos básicos de qualquer linguagem são os identificadores, as palavras-chaves, as constantes e os operadores. Podemos identificar os elementos básicos da linguagem Pascal como:

Identificadores: sequências de caracteres alfanuméricos que começam com uma letra e podem conter também sublinhados (_). Por exemplo: x, soma_total, variavel_1.

Palavras-chave: palavras reservadas da linguagem que possuem um significado específico e não podem ser usadas como identificadores. Exemplos: program, var, begin, end, if, while, etc.

Constantes: valores fixos que aparecem no código fonte, podendo ser numéricos, booleanos ou caracteres. Exemplos: 42, true, 'a', 3.14, etc.

Operadores: símbolos que representam operações matemáticas, relacionais ou lógicas. Exemplos: +, -, *, /, =, <>, >, <, and, or, not, etc.

A especificação desses elementos básicos na notação EBNF seria:

- Identificador: letra (letra | dígito | _)*
- Palavra-chave: 'programa' | 'var' | 'começar' | 'fim' | 'se' | 'enquanto' | ...
- Constante numérica: dígito+
- Constante booleana: 'true' | 'falso'
- Caractere constante: ' (caractere exclusivo aspas simples) '
- Operador: '+' | '-' | '*' | '/' | '=' | '<>' | '>' | '<' | 'e' | 'ou' | 'não' | ...

Descrevendo melhor, ficaria:

(1) <letra>::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v |w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | W | V | W | X | Y | Z

(2) <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(3) <identificador>::= <letra>{ [<letra> | <dígitos>] }

Estes começam com uma letra e os demais símbolos podem ser letras ou dígitos. Vale lembrar, que { } indica repetições de 0 a n, e [ ] indica que pode aparecer 0 ou 1 vez, de acordo a notação EBNF.

(4) <constante> ::= <const_numerica> | <const_literal>

As produções <const_numerica> e <const_literal> precisam ser criadas, e quanto mais simples for cada produção mais fácil será a implementação.

Vamos construir as produções auxiliares:

(5) <inteiro> ::= <digito> { <digito> }

(6) <decimal> ::= <inteiro> '.' <inteiro>

(7) <const_numerica> ::= <inteiro> | <decimal>

(8) <aspasDupla> ::= "

Essa produção, mesmo representando apenas um símbolo, é útil para a clareza na especificação da gramática.

(9) <caracterqq> ::= {[<alfabeto>]}

É uma produção com todos os símbolos possíveis do teclado

(10) <const_literal> ::= <aspasDupla> {[<caracterqq>]}<aspasDupla>

Agora, vamos às produções lexicais finais:

(11) <operadoresAritmeticos> ::= + | - | * | /

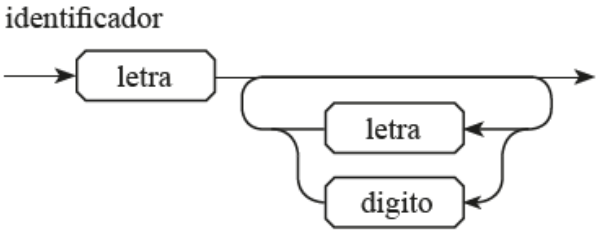
(12) <operadoresComparação> ::= <> | < | > | <= | >=

(13) <sombolosEspeciais> ::= (|) | [|] | := | . | , | :

(**14) <palavrasChaves> ::=** div | or | and | not | if | then | else | of | while | do | begin | end| read | write | var | array | function | procedure | program | true | false | char | integer | boolean

Para ajudar nesse processo, podemos construir um diagrama que represente a produção (3), conforme representada na Figura 17.

Figura 17 – Diagrama de sintaxe da produção (3) <identificador>

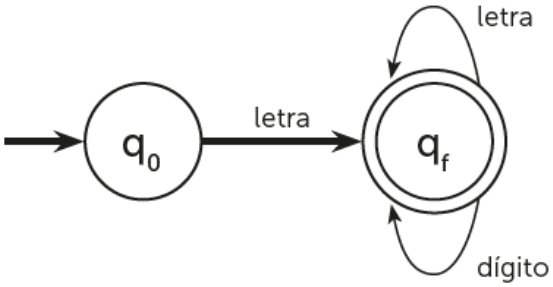


Fonte: Fedozzi (2018).

Segundo Fedozzi (2018) os autômatos finitos determinísticos (AFD) são reconhecedores e facilmente implementados. O JFLEX gera métodos que fazem isso.

Para a produção (3) <identificador> ::= <letra>{[<letra>|<dígitos]}, o AFD é representado conforme a Figura 18.

Figura 17 – AFD para <identificador>



Fonte: Fedozzi (2018).

UNIDADE 2 – ESPECIFICAÇÃO DA ANÁLISE LÉXICA E TÉCNICAS DE IMPLEMENTAÇÃO

SEÇÃO 2 – CONSTRUÇÃO DE UM ANALISADOR LÉXICO

Segundo Fedozzi (2018), um gerador de analisador léxico recebe como entrada uma especificação com um conjunto de expressões regulares e as ações correspondentes. Ele produz um programa, conhecido como lexer, que lê a entrada e combina com as expressões regulares na especificação, executando uma ação correspondente caso haja uma correspondência. Esses lexers geralmente são a primeira etapa em compiladores, identificando palavras-chave, comentários, operadores, entre outros, e gerando um fluxo de tokens de entrada para análise sintática. Além disso, os lexers também podem ser utilizados para diversos outros propósitos.

De acordo com Klein (2015), um gerador de **analisador léxico**, ou scanner, recebe como entrada uma especificação contendo um conjunto de expressões regulares. Ele produz um programa, chamado de **lexer**, que lê a entrada do programa escrito na linguagem (o código-fonte) e retorna o padrão de cada tipo de token identificado, ou indica que o padrão não é válido, ou seja, não é reconhecido pela gramática.

Segundo Fedozzi (2018), embora o JFlex tenha sido projetado para trabalhar em conjunto com o gerador de analisador sintático CUP, ele também pode ser utilizado em conjunto com outros geradores de parser, como o ANTLR.

Segue um exemplo de uma especificação JFlex que reconhece identificadores, palavras-chave (if, else, while), números inteiros e operadores aritméticos (+, -, *, /):

```
%{
// código java para ser incluído na geração do lexer
%}
// definição de tokens e expressões regulares
%class Lexer
%unicode
%public
%type int
```

```
// definição dos estados
%state IN_COMMENT
// definição de macros
DIGIT = [0-9]
LETTER = [a-zA-Z]
ID = {LETTER}({LETTER}|{DIGIT})*
// definição dos tokens
%token IF <IF>
%token ELSE <ELSE>
%token WHILE <WHILE>
%token INT_CONST {DIGIT}+
%token ADD_OP "+"
%token SUB_OP "-"
%token MUL_OP "*"
%token DIV_OP "/"
// definição das regras
%%
// regra para ignorar espaços em branco e quebras de linha
[\t\n\f\r ]+
// regra para tratar comentários
"/*"              { yybegin(IN_COMMENT); }
<IN_COMMENT>"*/"      { yybegin(YYINITIAL); }
<IN_COMMENT>.
// regra para identificar palavras-chave e identificadores
{ID}              {
                  if (yytext().equals("if")) {
                     return IF;
                  } else if (yytext().equals("else")) {
                     return ELSE;
                  } else if (yytext().equals("while")) {
                     return WHILE;
                  } else {
                     return ID;
```

```
                    }
                }
    // regra para identificar números inteiros
    {DIGIT}+            { return INT_CONST; }
    // regra para identificar operadores aritméticos
    "+"                 { return ADD_OP; }
    "-"                 { return SUB_OP; }
    "*"                 { return MUL_OP; }
    "/"                 { return DIV_OP; }
    // regra para tratamento de erros
    .                   {
                         System.out.println("Erro léxico: " + yytext());
                         return -1;
                        }
```

Neste exemplo, a especificação define os tokens usando a diretiva %token, como expressões regulares usando a diretiva %regex, como regras usando a sintaxe %%e macros usando a sintaxe %name = regex. Uma diretiva %classdefine uma classe do lexer que será gerada e %statedefine um estado de processamento adicional para tratar comentários. A implementação das ações correspondentes aos tokens e regras é escrita em código Java incluída na seção %{ ... %}.

Para implementar um analisador léxico é necessário:

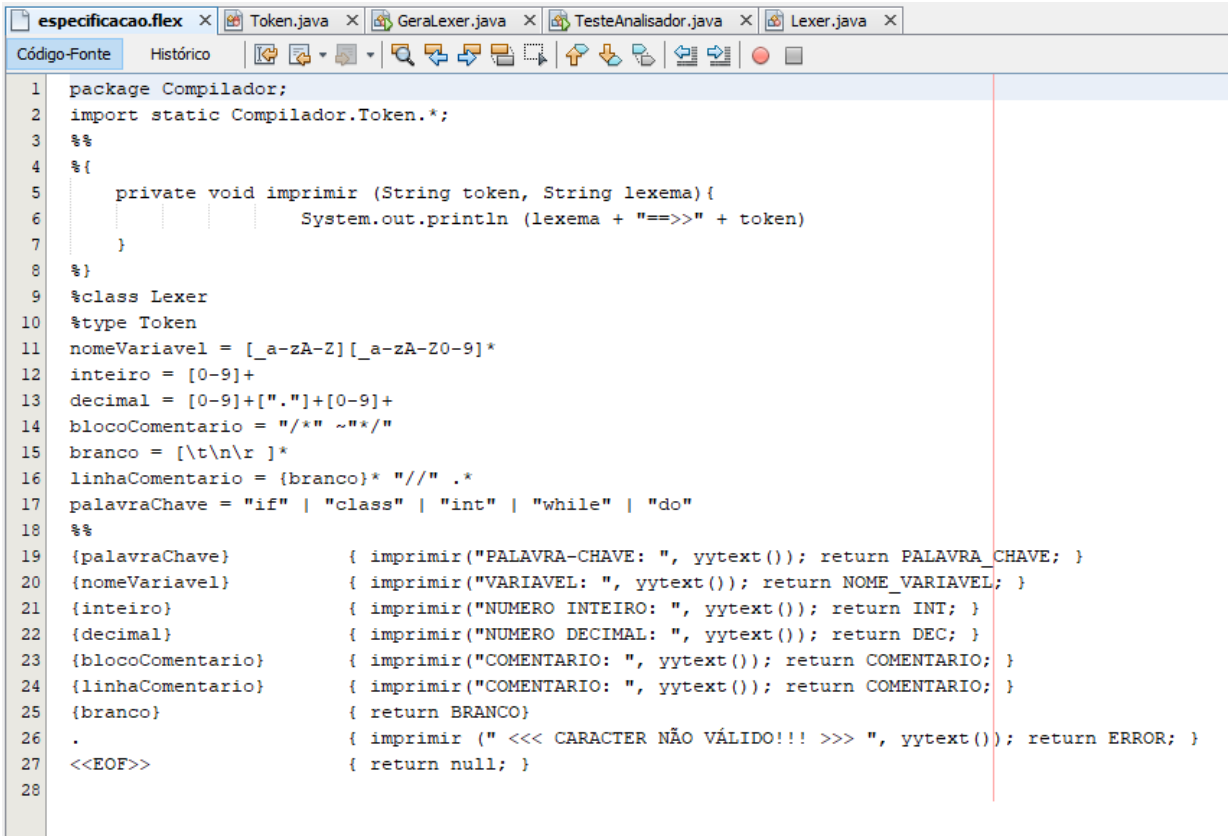
1. Do kit JDK 8.0 ou superior para o JAVA SE.
   http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html
2. Da IDE NetBeans
   https://netbeans.org/community/releases/81/index_pt_BR.html
3. Do gerador JFlex
   http://jflex.de/download.html

Conforme Fedozzi (2018), depois de baixados os softwares indicados, eles devem ser instalados nesta ordem: primeiro o JDK, depois o Netbeans. Siga as instruções recomendadas no site de cada ferramenta baixada. Depois é necessário o plug-in JFlex para o Netbeans. Após baixar o JFLEX.zip, faça a descompactação e abra o Netbeans.

Para criar um analisador léxico, abra o Netbeans e siga os passos a seguir:

1. Criar um Projeto: Unidade2
2. Criar um Pacote: Compilador
3. Criar um arquivo vazio java: especificação.jflex (Figura 18).

Figura 18 – especificação.flex

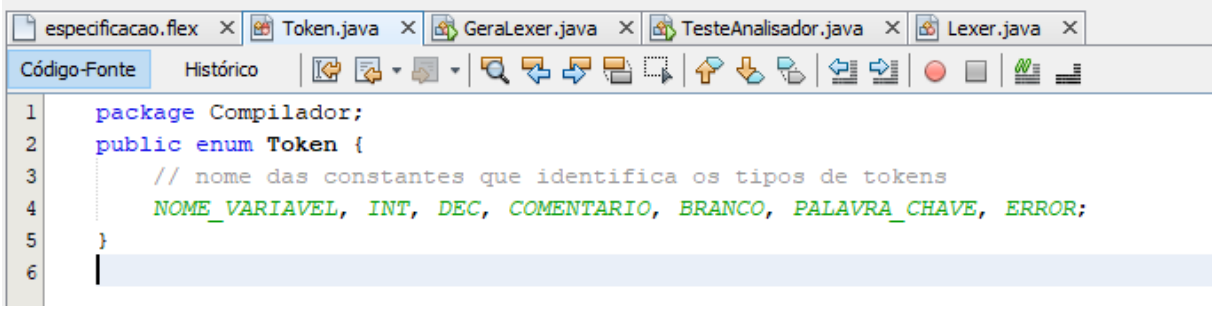



```
1  package Compilador;
2  import static Compilador.Token.*;
3  %%
4  %{
5      private void imprimir (String token, String lexema){
6                  System.out.println (lexema + "==>>" + token)
7      }
8  %}
9  %class Lexer
10 %type Token
11 nomeVariavel = [_a-zA-Z][_a-zA-Z0-9]*
12 inteiro = [0-9]+
13 decimal = [0-9]+["."]+[0-9]+
14 blocoComentario = "/*" ~"*/"
15 branco = [\t\n\r ]*
16 linhaComentario = {branco}* "//" .*
17 palavraChave = "if" | "class" | "int" | "while" | "do"
18 %%
19 {palavraChave}          { imprimir("PALAVRA-CHAVE: ", yytext()); return PALAVRA_CHAVE; }
20 {nomeVariavel}          { imprimir("VARIAVEL: ", yytext()); return NOME_VARIAVEL; }
21 {inteiro}               { imprimir("NUMERO INTEIRO: ", yytext()); return INT; }
22 {decimal}               { imprimir("NUMERO DECIMAL: ", yytext()); return DEC; }
23 {blocoComentario}       { imprimir("COMENTARIO: ", yytext()); return COMENTARIO; }
24 {linhaComentario}       { imprimir("COMENTARIO: ", yytext()); return COMENTARIO; }
25 {branco}                { return BRANCO}
26 .                       { imprimir (" <<< CARACTER NÃO VÁLIDO!!! >>> ", yytext()); return ERROR; }
27 <<EOF>>                 { return null; }
28
```

Fonte: os autores.

4. Criar uma classe: Token.java (Figura 19).

Figura 19 – Token.java

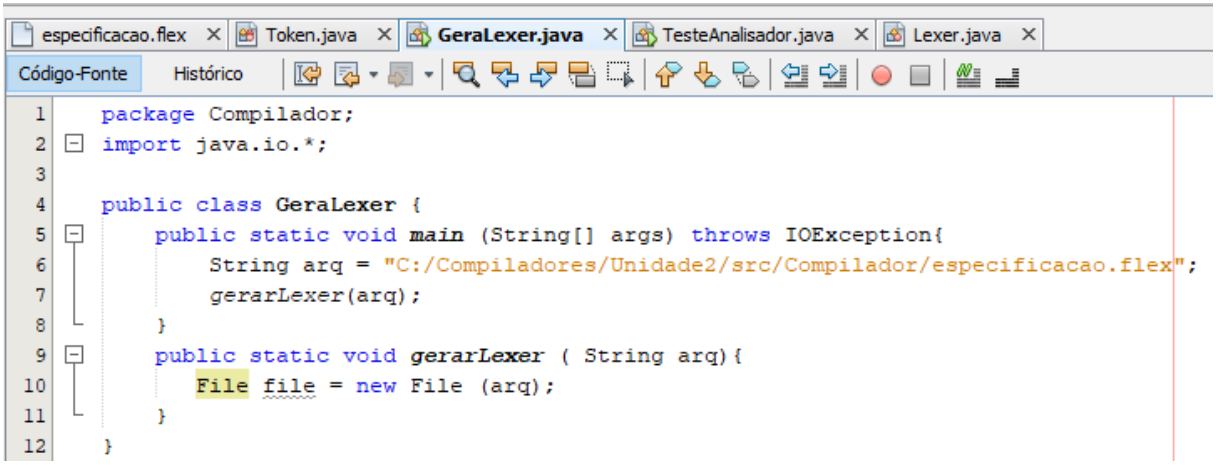



```
1  package Compilador;
2  public enum Token {
3      // nome das constantes que identifica os tipos de tokens
4      NOME_VARIAVEL, INT, DEC, COMENTARIO, BRANCO, PALAVRA_CHAVE, ERROR;
5  }
6
```

Fonte: elaborada pela autora.

5. Criar uma classe: GeraLexer.java (Figura 20).

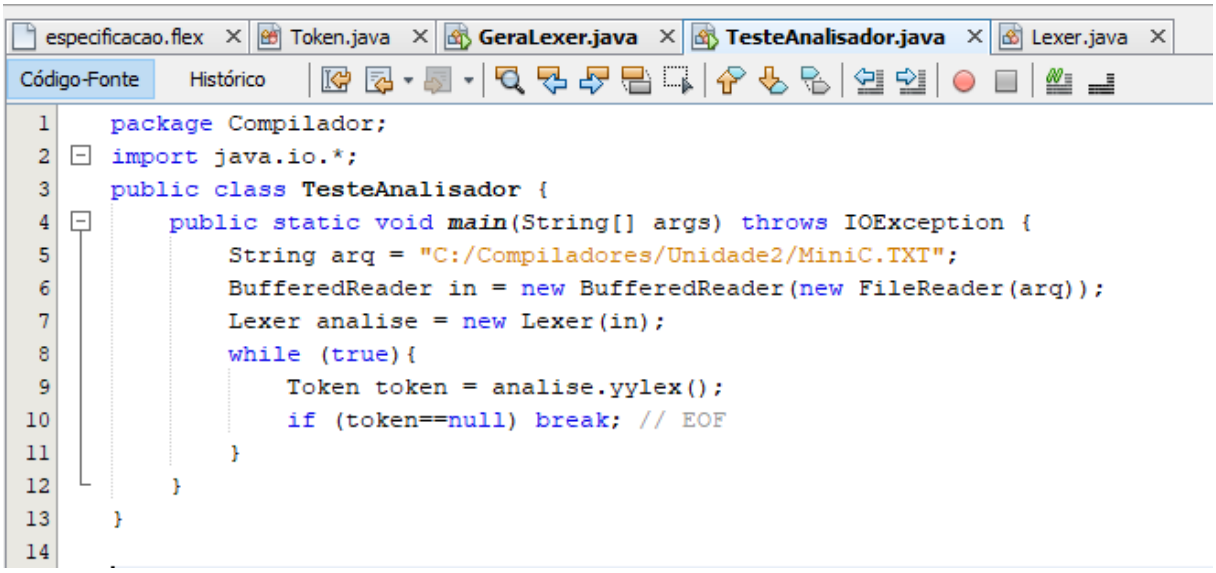
Figura 20 – GerarLexer.java



Fonte: elaborada pela autora**.**

6. Agora você pode executar o projeto pressionando a tecla <F6>, e o resultado será uma nova classe no seu projeto, denominada Lexer.
7. Criar uma classe: TesteAnallisador.java (Figura 21).

Figura 21 – TesteAnalisador.java

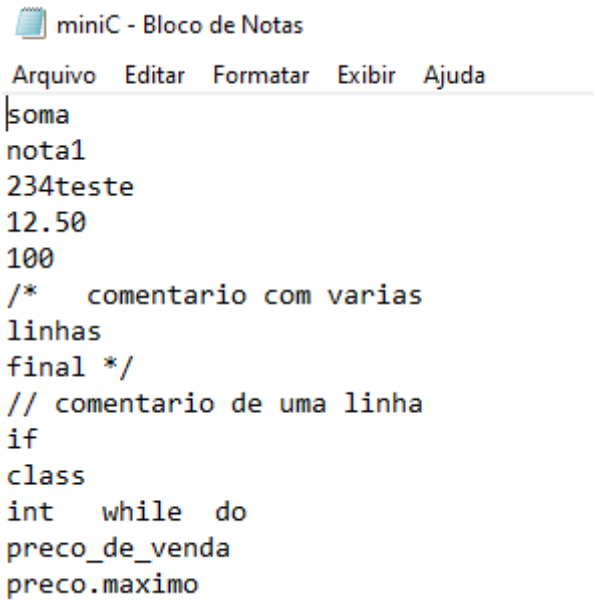


Fonte: elaborada pela autora.

8. Criar um arquivo para testar a aplicação: MiniC.txt

   A Figura 22 apresenta o arquivo de teste (a entrada).

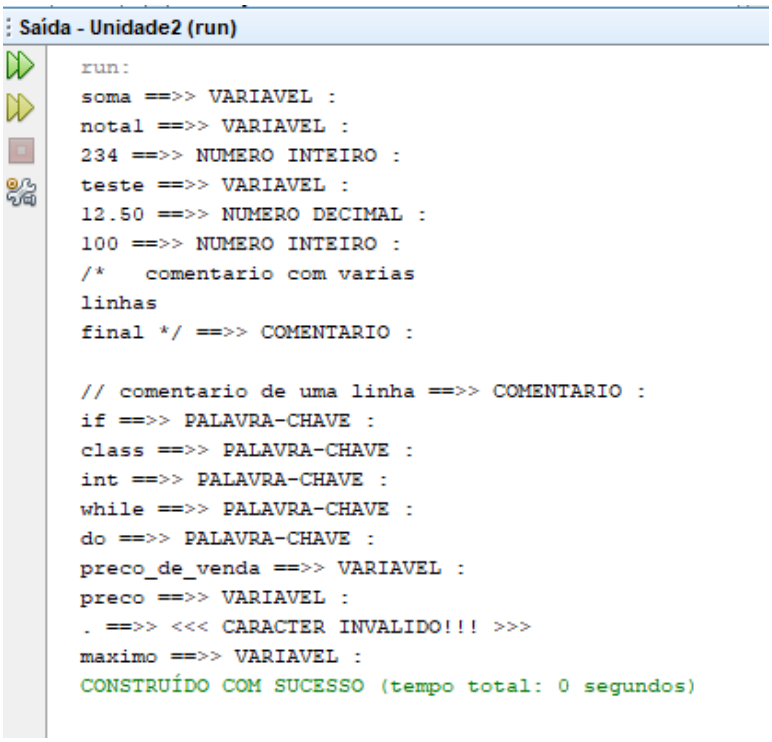
Figura 22 – Arquivo de entrada (teste)



Fonte: os autores.

Feito o arquivo de teste, podemos finalmente executar o projeto, observe a Figura 23 A saída mostra o lexema e o tipo do token, à sua frente.

Figura 23 – Saída



Fonte: elaborada pela autora.

UNIDADE 2 – ESPECIFICAÇÃO DA ANÁLISE LÉXICA E TÉCNICAS DE IMPLEMENTAÇÃO

SEÇÃO 3 – ANÁLISE SINTÁTICA

Após a análise léxica, o **analisador sintático** (parser) é responsável por receber os tokens gerados pelo analisador léxico e construir a estrutura de árvore que representa a estrutura gramatical do código fonte. Essa etapa verifica se a sequência de tokens recebida pode ser derivada a partir das regras definidas na gramática da linguagem, verificando assim se o código fonte está sintaticamente correto.

Segundo Aho (2007), a **estrutura sintática** de cada linguagem de programação (LP) é definida por um conjunto de regras, que pode ser descrita por meio de linguagens livres de contexto (LLC) ou pela notação BNF (Backus-Naur Forms).

Segundo Fedozzi (2018), o **analisador sintático** tem como função verificar se a fonte do programa está escrita de acordo com as regras definidas pela sua gramática, ou seja, se as estruturas sintáticas estão corretas. Para alcançar esse objetivo, ele solicita ao analisador léxico a análise do fluxo dos tokens de entrada e, após receber o retorno do léxico, verifica se o fluxo, ou seja, o conjunto de tokens, pode ser gerado pela gramática da linguagem. Assim, o agrupamento de tokens forma uma "frase gramatical" e analisador sintático irá gerar uma representação hierárquica chamada de árvore sintática.

O parser pode ser implementado de diversas formas, sendo a mais comum a utilização de técnicas de **análise descendente** (top-down) ou **análise ascendente** (bottom-up).

A **análise descendente**, também conhecida como análise **top-down**, é uma técnica de análise sintática que começa com o símbolo inicial da gramática e tenta derivar a cadeia de entrada seguindo as produções da gramática. Em outras palavras, a análise começa do topo da árvore de derivação e desce até a cadeia de entrada. Essa técnica utiliza um conjunto de procedimentos recursivos para derivar as sentenças da gramática a partir do símbolo inicial. O analisador começa examinando o símbolo inicial da gramática e, em seguida, utiliza as regras de produção para derivar cada símbolo da cadeia de entrada até que a cadeia completa seja gerada ou um erro seja encontrado.

Existem dois tipos de análise descendente: a **análise descendente recursiva** e a **análise descendente com retrocesso** (backtracking).

Na **análise descendente recursiva**, cada gramática não-terminal na gramática é associada a um procedimento que tenta gerar a sentença da gramática a partir dessa não-terminal. O procedimento chama outros procedimentos recursivamente até que a cadeia de entrada seja gerada ou um erro seja encontrado. A análise descendente recursiva é fácil de implementar e entender, mas pode ser ineficiente em algumas gramáticas ambíguas ou recursivas.

Já a **análise descendente com retrocesso** tenta gerar a sentença da gramática de forma não determinística, explorando todas as possíveis alternativas de produção. Caso a derivação atual não leve à cadeia de entrada, o analisador retorna e tenta outra produção. Essa técnica é mais poderosa do que a análise descendente recursiva, mas pode ser mais lenta e consumir mais recursos computacionais.

Em resumo, a técnica de **análise descendente** é uma abordagem para análise sintática que começa com o símbolo inicial da gramática e tenta derivar a cadeia de entrada seguindo as regras da gramática. Essa técnica pode ser integrada de forma recursiva ou com retrocesso, dependendo da gramática e das necessidades do desenvolvedor.

A **análise ascendente**, também conhecida como análise bottom-up, é uma técnica utilizada pelos analisadores sintáticos para construir uma árvore sintática a partir dos tokens de entrada. Nessa técnica, a análise começa com os símbolos da gramática e, a partir deles, vai construir a árvore sintática até chegar ao símbolo inicial.

A **análise ascendente** é chamada de bottom-up porque ela começa a partir dos terminais, que estão no "fundo" da gramática, e vai "subindo" até chegar ao símbolo inicial. Para isso, os analisadores sintáticos bottom-up utilizam uma pilha (stack) para armazenar os símbolos e, a cada passo, vão desempilhando os símbolos que formam uma produção e empilhando o símbolo não-terminal correspondente a essa produção.

Existem várias **técnicas de análise ascendente**, como a análise LR, a análise LALR e a análise SLR. Essas técnicas se diferenciam pelo modo como identificam qual produção deve ser aplicada em cada passo da análise. A análise LR é a técnica mais geral e poderosa, mas também é a mais complexa e de implementação mais difícil. A análise LALR é uma técnica mais simples e eficiente,

que consegue lidar com a maioria das gramáticas usadas em linguagens de programação. A análise SLR é ainda mais simples, mas é menos poderosa e não consegue lidar com todas as gramáticas.

De acordo com Aho (2007), o tratamento de erros em um analisador sintático tem três metas simples:

1. relatar com clareza e precisão a presença de erros;
2. ser capaz de se recuperar do erro para continuar identificando os próximos erros, se houver;
3. não afeta a eficiência do processo de análise sintática para programas corretos.

Conforme Fedozzi (2018), a segunda meta de um tratar de erros em um analisador sintático é ser capaz de se recuperar após a detecção de um erro e continuar a análise do programa. Essa é uma das tarefas mais exigentes no projeto de um compilador, uma vez que o objetivo é identificar todos os erros a cada processo de compilação, o que nem sempre é possível. Dessa forma, foram desenvolvidas estratégias para identificar o máximo de erros em cada construção. A seguir, apresentamos algumas das estratégias mais comuns utilizadas para lidar com erros:

A **modalidade do desespero** se refere a uma técnica de recuperação de erros em analisadores sintáticos que consiste em descartar tokens até encontrar um que possa ser usado para reiniciar a análise. Essa técnica é chamada de "modalidade do desespero" porque é considerada a última opção quando todas as outras estratégias de recuperação de erro falharam.

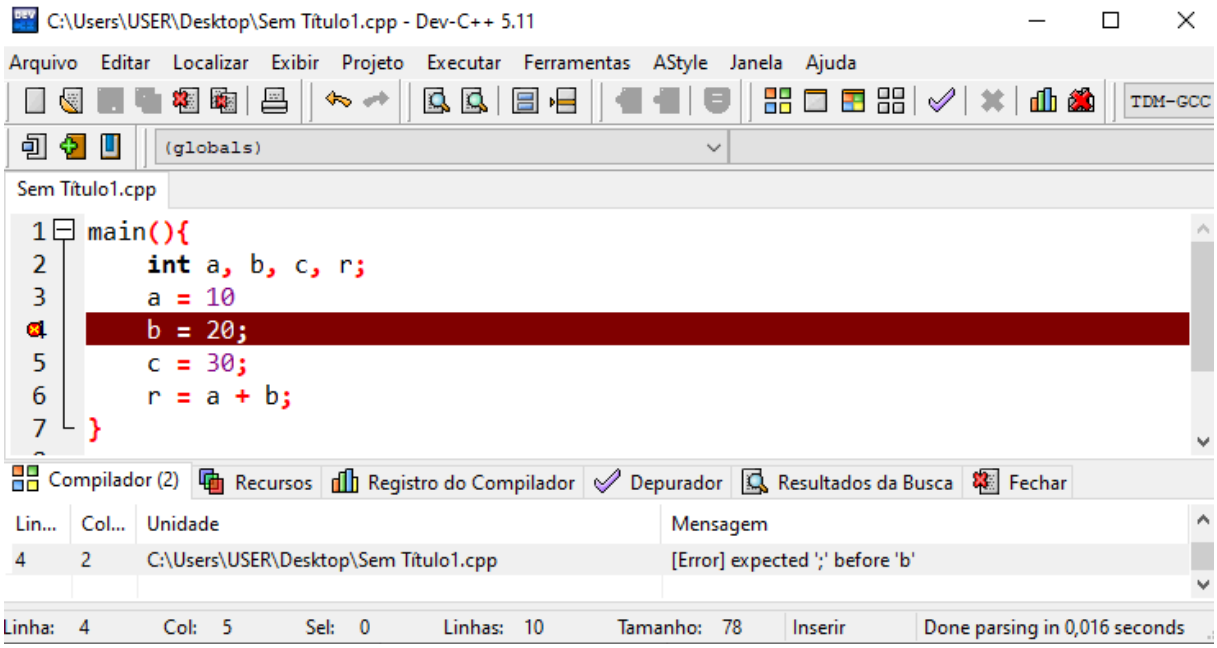
O **nível de frase** se refere ao nível de análise no qual os erros são detectados. Em uma análise de frase, o analisador sintático detecta erros em cada frase (ou comando) individual do programa. Por outro lado, em uma análise global, o analisador sintático verifica todo o programa antes de relatar qualquer erro.

Produções de erros são produções adicionais que são adicionadas à gramática de uma linguagem de programação para lidar com erros sintáticos. Essas produções são usadas pelo analisador sintático para tentar corrigir o programa quando um erro é detectado.

A **correção global** é uma estratégia de recuperação de erros em que o analisador sintático continua a análise do programa, mesmo após detectar um erro, e tenta identificar o máximo de erros possível em uma única passagem. Essa abordagem geralmente envolve uma combinação de técnicas de recuperação de

erros, como a modalidade do desespero e produção de erros, para tentar minimizar o impacto dos erros no processo de compilação.

Figura 24 – Programa exemplo em C



Fonte: elaborada pela autora.

Na situação presente, a estratégia utilizada para recuperação de erros foi a modalidade do desespero, ou seja, uma estratégia que não realiza correções no código fonte. Como ilustrado na Figura 24, o compilador identificou o erro na linha 3, em que faltou o ponto e vírgula, mas só indicou o erro na linha 4 após detectar o símbolo terminal (;). A mensagem de erro privativa pelo compilador foi clara e indicou que era esperado um ponto e vírgula antes do símbolo 'b'.

Porém, caso o programador tivesse escrito apenas '= 20;' na linha 4, o compilador localizaria um novo erro e não faria referência ao erro na linha 3, já que as linhas 3 e 4 são vistas como uma única "frase" pelo compilador. Nesse caso, a recuperação de erros sem correção é rápida e eficiente, mas não é perfeita. Infelizmente, ainda não existe uma estratégia definitiva para lidar com todos os tipos de erros em um compilador.

UNIDADE 3 – TABELA DE SÍMBOLOS, ANÁLISE SEMÂNTICA E TRADUÇÃO DIRIGIDA POR SINTAXE

SEÇÃO 1 –ANÁLISE SEMÂNTICA

Após a verificação sintática, a **análise semântica** é responsável por verificar se as informações contidas no código fonte estão semanticamente corretas. Nessa etapa, é realizada a verificação de tipos, checagem de escopos, resolução de sobrecarga de operadores, entre outras verificações.

Por exemplo, se um código fonte contém uma expressão 5 + "string", a análise semântica deve detectar que os tipos dos operandos são incompatíveis e gerar uma mensagem de erro informando que a operação não é permitida.

A **análise semântica** é uma fase do processo de compilação que verifica se as construções sintaticamente corretas do programa estão de acordo com as restrições semânticas da linguagem de programação. Em outras palavras, é a verificação se o programa faz sentido de acordo com as regras da linguagem.

Essa fase inclui várias tarefas, como a verificação de tipos, a verificação de escopos, a geração de código intermediário e a verificação de erros semânticos. As tarefas variam de acordo com a linguagem de programação, mas em geral, a análise semântica é responsável por garantir que o programa seja seguro, eficiente e livre de erros.

Um exemplo de **análise semântica** é uma verificação de tipos. Em linguagens de programação tipadas, a análise semântica deve verificar se cada operação é aplicada a operandos do tipo correto. Por exemplo, em uma linguagem que suporta operações aritméticas, a análise semântica deve garantir que a soma seja realizada apenas entre operandos numéricos. Caso contrário, um erro semântico deve ser reportado.

Outro exemplo é a verificação de escopo. Em linguagens que suportam escopo, a análise semântica deve garantir que as variáveis sejam declaradas antes de serem usadas e que as variáveis tenham visibilidade no escopo em que estão sendo usadas. Caso contrário, um erro semântico deve ser reportado.

Em resumo, a **análise semântica** é uma parte crucial do processo de compilação que garante que o programa seja correto e eficiente de acordo com as restrições semânticas da linguagem de programação.

Seja uma estrutura simples, que represente uma expressão matemática, do tipo: a + b * c.

O nosso alfabeto será ∑ = {a, b, c, +, *}

A gramática no padrão EBNF, a qual iremos denominar de L, será:

<expr> ::= <expr> <op> <expr> | <id>

<id> ::= a | b | c

<op> ::= + | *

Processo de derivação: Seja w uma palavra (sentença) que pertença à linguagem L definida, então w ∈ L . Seja w1 = a + b e desejamos verificar se w1 □ L .

Conforme Fedozzi (2018) uma forma de verificar isso é por meio do processo de derivação, que consiste em substituir a sentença, ou parte dela, no lado direito da produção da gramática repetidas vezes, até alcançar os símbolos terminais, e, segundo Aho (2007), podemos aplicar repetidamente as produções em qualquer ordem, a fim de obtermos uma sequência de substituições.

Segundo Fedozzi (2018), Para ajudar a compreensão do processo de derivação, iremos substituir por letra MAIÚSCULA cada símbolo não-terminal da gramática, e adotaremos para os símbolos terminais, aqueles que pertencem ao alfabeto, neste caso " S " , a grafia em negrito para diferenciá-los dos demais. A seta " → " indicará a definição da produção, equivalente à notação "::=" no padrão EBNF, e " ⇒ " indicará uma derivação.

Assim, de acordo com essas convenções, antes de aplicarmos o processo de derivação à gramática no padrão EBNF, a qual estamos analisando, vamos reescrever a gramática equivalente de acordo com os padrões convencionados aqui, assim teremos a Figura 25.

Figura 25 – Gramática

Gramática padrão EBNF	Gramática equivalente
<*expr*> ::= <*expr*> <*op*> <*expr*>	E → E OP E | ID
<*id*> ::= **a** | **b** | c	ID → **a** | **b** | **c**
<*op*> ::= + | *	OP → + | *

Fonte: Fedozzi (2018).

A cadeia que desejamos derivar é w1= a + b, assim, vamos aplicar as derivações sucessivas até chegarmos aos símbolos terminais, conforme a Figura 26, Fedozzi (2018).

Figura 26 – Derivação

⇒ E → E OP E	1º passo deriva a regra mais à esquerda, pois a cadeia ω_1 tinha um **OP**
⇒ E → ID OP E	
⇒ E → a + E	2º passo derivou o **E** mais à esquerda
	3º passo derivou o **ID**. Chegamos ao símbolo terminal **a**
⇒ E → a OP E	4º passo derivou **OP**.
⇒ E → a + E	5º passo derivou o **ID**. Chegamos ao símbolo terminal **+**
⇒ E → a + ID	6º passo derivou o **E**
⇒ E → a + b	7º passo derivou **ID**. Chegamos ao símbolo terminal **b**.

Fonte: Fedozzi (2018).

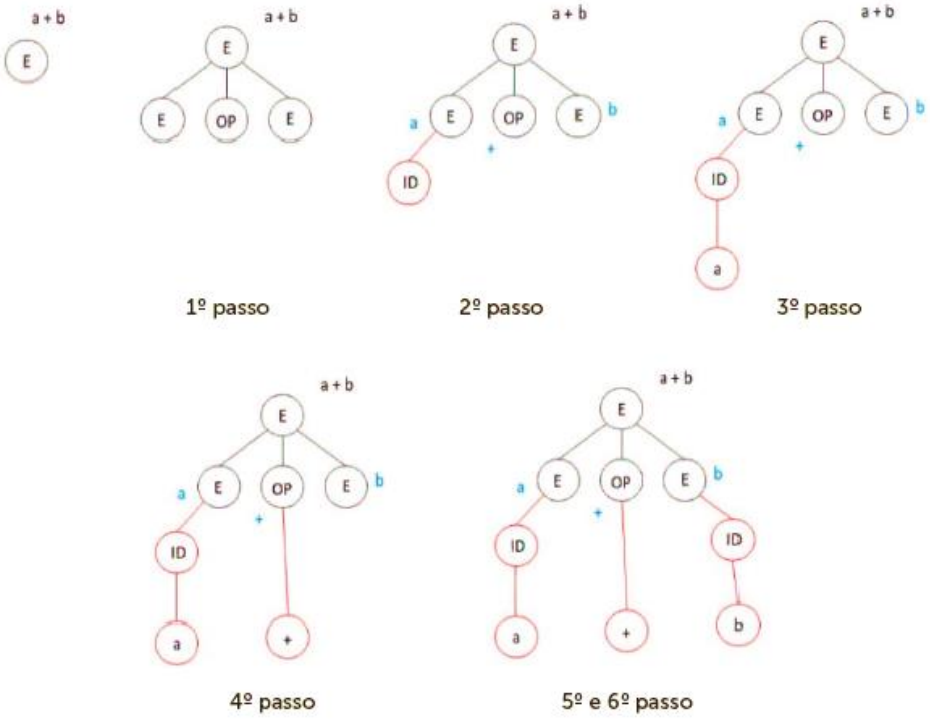
Portanto, podemos afirmar que a sentença "**a+b**" é uma cadeia gerada pela linguagem L, pois, ao fazermos as derivações sucessivas, segundo a gramática L, chegamos a uma cadeia de símbolos terminais, logo se w1= a+b então w1 ∈ L, conforme Fedozzi (2018),

A derivação da árvore sintática, ou gramatical, segundo afirma Aho (2007) pode ser vista como a representação gráfica da derivação. Os **nós** tem sucessores, logo são símbolos **não terminais**, e as **folhas** são os símbolos **terminais**.

Ao construirmos a árvore de derivação de uma sentença de acordo com a gramática de uma linguagem, aplicarmos sucessivamente as regras e alcançarmos as folhas para todos os nós, então a sentença é gerada pela gramática.

Veja na Figura 27 a construção da árvore de derivação para a sentença '**a + b**', pertencente a gramática da linguagem L.

Figura 27 – Passo a passo da árvore de derivação para a sentença “a+b”



Fonte: Fedozzi (2018).

Conforme Fedozzi (2018), tanto a derivação como a árvore gramatical foram derivadas da esquerda para a direita (Left-to-Right), mas poderíamos ter feito isso da direita para esquerda (Right-to-Left). No caso da sentença “a+b”, tanto a derivação quanto a árvore seriam iguais se tivéssemos realizado a derivação mais à direita, mas nem sempre é assim. Se sentença a ser analisada for “a + b * c” para a gramática definida, as árvores de derivações serão diferentes se fizermos a derivação mais à esquerda ou mais à direita, e isso não é desejável para uma gramática, pois indica que a gramática é ambígua.

UNIDADE 3 – TABELA DE SÍMBOLOS, ANÁLISE SEMÂNTICA E TRADUÇÃO DIRIGIDA POR SINTAXE

SEÇÃO 2 – TRADUÇÃO DIRIGIDA PELA SINTAXE

Segundo Aho (2007), existem duas abordagens para associar informações semânticas a uma árvore sintática:

- definições dirigidas pela sintaxe;
- esquemas de tradução.

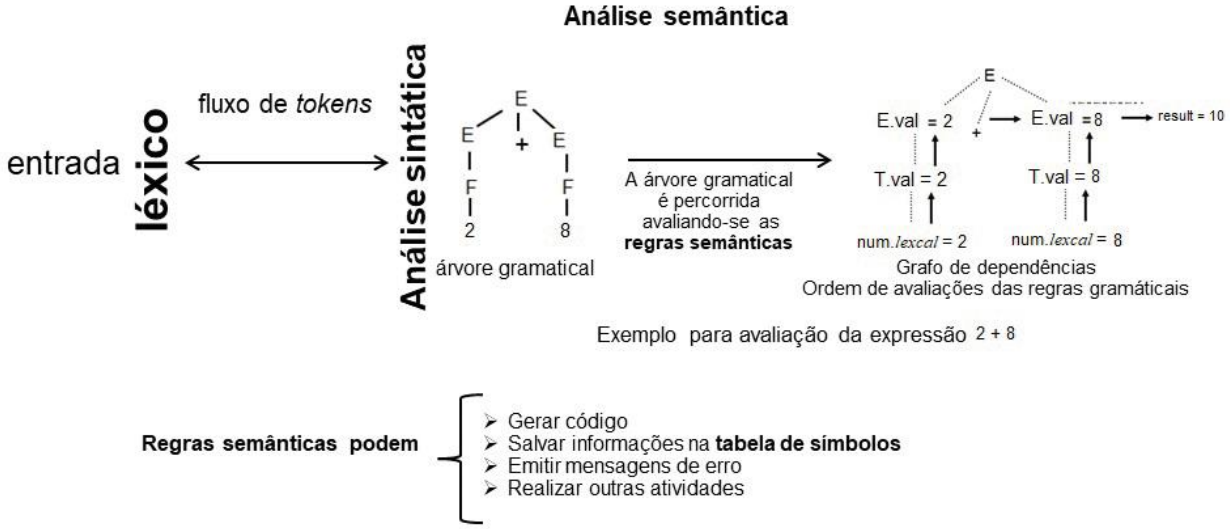
Conforme Fedozzi (2018), na definição direcionada pela sintaxe, as especificações são de alto nível e para cada token associamos um conjunto de atributos. Com base na árvore de derivação, esses atributos podem ser classificados em:

- **Sintetizados:** quando é anotada a avaliação da regra semântica de cada nó, de baixo para cima, ou seja, da folha para a raiz.
- **Herdados:** quando o valor semântico do nó é definido pelo nó superior (pai) e lateral (irmão), como no caso de uma variável aparecer do lado esquerdo e direito de uma recebida.

De acordo Fedozzi (2018) nos esquemas de tradução, insira-se as ações semânticas no lado direito das produções e é possível determinar a ordem em que as ações e as estimativas dos atributos ocorrerão.

A definição dirigida pela sintaxe é um método mais geral e pode ser usado para gerar código ou fazer outras transformações sobre a entrada. Por outro lado, os esquemas de tradução são geralmente usados para gerar código-fonte. A Figura 28 apresenta um esquema do processo de tradução dirigido pela sintaxe.

Figura 28 – Processo da tradução dirigida pela sintaxe



Fonte: Fedozzi (2018).

A árvore gramatical representada na Figura 28 corresponde à sentença 2 + 8, e, à direita, temos uma **árvore gramatical anotada** por meio das regras semânticas. Para melhor compreensão da árvore anotada, vamos comparar as regras de produção e as regras semânticas para este caso, na Figura 29.

Figura 29 – Regras semânticas

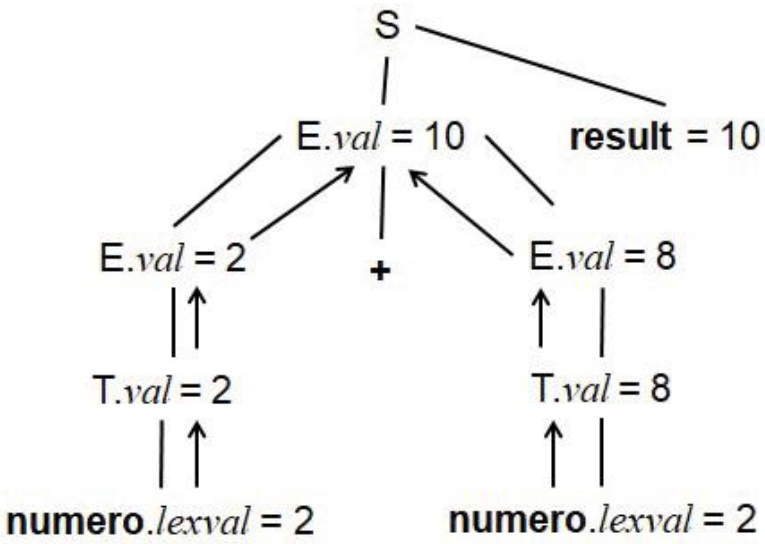
Regras de produção	Regras semânticas	Comentários
S :=**result**	Resultado (E.*val*)	**result** é um terminal que representa o resultado do cálculo.
E := E + E | T	E.*val* = E.*val* + E.*val* E.*val* = T.*val*	Quando realizamos a soma de valores, há dependência entre os termos, assim, a definição dirigida por sintaxe associa um atributo a cada termo. Neste caso, ***val*** indica o tipo e está associado aos símbolos não terminais
T := **numero**	T = **numero**. *lexval*	**numero** é um símbolo terminal e está associado a um atributo sintetizado, pois, neste caso, não há regra semântica para o mesmo, já que o valor deverá ser fornecido pelo léxico.

Fonte: Fedozzi (2018).

A árvore gramatical anotada correspondente às produções e respectivas regras

semânticas é apresentada na Figura 30.

Figura 30 – Regras semânticas



Fonte: Fedozzi (2018)

UNIDADE 3 – TABELA DE SÍMBOLOS, ANÁLISE SEMÂNTICA E TRADUÇÃO DIRIGIDA POR SINTAXE

SEÇÃO 3 – TABELA DE SÍMBOLOS

A **tabela de símbolos** (TS) é uma estrutura de dados utilizada pelos compiladores para armazenar informações sobre os identificadores (nomes) que aparecem na fonte do programa, tais como variáveis, funções, constantes e tipos. Ela é ocupada durante a análise léxica e sintática do programa, e serve como um repositório central para o gerenciamento dessas informações ao longo do processo de compilação.

Cada entrada na tabela de símbolos representa um identificador único no programa e contém informações como o tipo de dados associados ao identificador, a localização (endereço de memória) onde ele será armazenado, o escopo em que foi declarado, entre outros atributos. Um TS é utilizado para garantir que o uso correto e consistente dos identificadores no programa e ajudar a detectar erros de sintaxe ou semântica relacionados a esses identificadores.

A estrutura de dados utilizada para implementar um TS pode variar dependendo do compilador e da linguagem de programação, sendo comum o uso de tabelas hash, árvores de busca binária ou listas temporárias. A eficiência e rapidez do acesso à tabela de símbolos são importantes para garantir um processo de compilação rápido e eficiente.

Segundo Fedozzi (2018), a Tabela de Símbolos é iniciada no analisador léxico com a inserção do par (token, lexema) quando o token é reconhecido. Outros atributos do token são identificados na análise sintática, que consulta a TS para adicionar, modificar ou incluir novos elementos na tabela.

Aho, Sethi e Ullman (2007) afirmam que o uso de **tabela hash** é fundamental na construção de um compilador, pois o número de elementos a serem inseridos na tabela é grande, com muitos tokens em um programa. Além disso, os elementos não possuem a mesma estrutura e há uma grande quantidade de operações a serem realizadas na TS que favoreciam uma resposta rápida. As ações semânticas definem todas essas características.

Segundo Fedozzi (2018), existem na linguagem Java diversas classes que fornecem suporte para tabelas hash, como a ConcurrentHasMap, apresentam na API

Collections java.util.concurrent, e as classes HashMap e Hashtable, disponíveis na java.util. Essas aulas implementam os elementos fundamentais das mesas hash, como a **função hash** e o tratamento de colisões, além de oferecerem métodos para inserção, busca e exclusão de elementos na mesa.

Segue um exemplo de como utilizar uma classe **Hashtable** em Java para manipular uma tabela de símbolos:

```
import java.util.Hashtable;
public class TabelaSimbolos {
    Hashtable<String, String> tabela;
    public TabelaSimbolos() {
        tabela = new Hashtable<String, String>();
    }
    public void inserirSimbolo(String token, String lexema) {
        tabela.put(token, lexema);
    }
    public String buscarSimbolo(String token) {
        return tabela.get(token);
    }
    public void removerSimbolo(String token) {
        tabela.remove(token);
    }
}
```

Neste exemplo, uma classe **Tabela Simbolos** utiliza uma classe **Hashtable** para implementar as operações básicas de uma tabela de símbolos: inserção, busca e remoção.

- O **método inserirSimbolo** recebe um parâmetro token que representa o token a ser inserido e um parâmetro lexema que representa o seu respectivo lexema. Em seguida, o método utiliza o método put da Hashtable para adicionar o par (token, lexema) na tabela.
- O **método buscarSimbolo** recebe um parâmetro token que representa o token a ser buscado e utiliza o método getda Hashtable para retornar o lexema associado a este token.
- O **método removerSimbolo** recebe um parâmetro token que representa o

token a ser removido e utiliza o método remove da Hashtable para remover o par (token, lexema) da tabela.

Esses são os métodos básicos que são usados na manipulação da tabela de símbolos. Claro que, dependendo das necessidades da linguagem que está sendo compilada, pode ser necessário implementar outras operações específicas.

Conforme Fedozzi (2018) o JFlex é o gerador de analisadores léxicos, e é possível integrar o léxico com o sintático, utilizando a ferramenta CUP. Com o uso integrado do JFlex e do CUP, será possível implementar ações semânticas e a tabela de símbolos.

Já o CUP é um gerador de analisadores sintáticos que possui uma sintaxe semelhante ao YACC e implementa a maioria das facilidades deste último. O **YACC** é o primeiro gerador de analisadores sintáticos e gera o código em C, enquanto o **CUP** foi criado em **JAVA** e gera código em JAVA, de acordo com Fedozzi (2018), o analisador sintático gerado pelo **CUP** implementa um analisador **LALR** (Look-Ahead Left-to-Right), que é um analisador bottom-up, ou seja, analisa as produções de baixo para cima (das folhas para a raiz) com análise preditiva (análise de símbolos à frente).

Assim como o JFlex, o CUP também possui algumas regras para a montagem do arquivo de especificação da gramática.

Conforme Fedozzi (2018) a estrutura do arquivo de especificação está dividida em quadro partes, na ordem a seguir:

- Diretivas do CUP e código Java que se deseja incluir na classe que será gerada;
- Definição dos símbolos terminais e não-terminais;
- Definição das precedências;
- Definição das produções gramaticais.

A Para criar um analisador léxico com o analisador sintático, utilizando respectivamente o JFlex e o CUP, abra o Netbeans e siga os passos a seguir:

1. Criar um Projeto: Unidade3
2. Criar um Pacote: ExemploCUP
3. Criar um arquivo vazio: sintatico.cup (Figura 31).

Figura 31 – sintático.cup

Página Inicial | sintatico.cup | lexico.flex

Código-Fonte | Histórico

```
1  package ExemploCUP;
2  import java_cup.runtime.*;
3  import java.util.*;
4  import java.io.*;
5  parser code {:
6    public void report_error(String message, Object info)   {
7        System.out.println("AVISO - " + message);
8        System.out.println(info.toString());
9  }
10   public void report_fatal_error(String message, Object info)   {
11         System.out.println("ERRO - " + message);
12         System.exit(-1);
13     }
14 :};
15 terminal PROGRAMA, INICIO, FIM, TIPO_INTEIRO, TIPO_CHAR;
16 terminal PTVG, SIMBOLO_ATRIB, CHAR, INTEIRO, VARIAVEL;
17
18 non terminal program, comandos, comando, decl_atrib, expr;
19 non terminal decl_variavel, tipo_dado;
20
21 start with program;
22 //declaracao das producoes da gramatica
23 program  ::= PROGRAMA INICIO comandos FIM;
24 comandos ::= comando comandos | comando;
25 comando  ::= decl_atrib | decl_variavel;
26
27 decl_atrib ::= VARIAVEL SIMBOLO_ATRIB expr PTVG;
28 expr ::= VARIAVEL | INTEIRO | CHAR;
29
30 decl_variavel ::= tipo_dado VARIAVEL PTVG;
31 tipo_dado ::= TIPO_INTEIRO | TIPO_CHAR;
```

Fonte: dados da pesquisa.

4. Criar um arquivo vazio: lexico.flex (Figura 32).

Figura 32 – léxico.flex

```
package ExemploCUP;
import java_cup.runtime.*;
%%
%cup
%public
%class Lexer
%line
%column
%{
  private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }
  private Symbol symbol(int type, Object value) { return new Symbol(type, yyline, yycolumn, value);  }
%}
DIGITO = [0-9]
LETRA = [a-zA-Z_]
LITERAL =  \"[^\"]*\"
INTEIRO = {DIGITO}+
VARIAVEL = {LETRA}+
IGNORE = [\n|\s|\t\r]
%%

<YYINITIAL> {
    "programa"      {return new Symbol(Sym.PROGRAMA); }
    "inicio"        {return new Symbol(Sym.INICIO); }
    "fim"           {return new Symbol(Sym.FIM); }
    "int"           {return new Symbol(Sym.TIPO_INTEIRO); }
    "caracter"      {return new Symbol(Sym.TIPO_CHAR); }
    ";"             {return new Symbol(Sym.PTVG); }
    "<-"            {return new Symbol(Sym.SIMBOLO_ATRIB); }
    {LITERAL}       {return new Symbol(Sym.CHAR); }
    {INTEIRO}       {return new Symbol(Sym.INTEIRO); }
   {VARIAVEL}      {return new Symbol(Sym.VARIAVEL); }
   {IGNORE}        {}
}
<<EOF>> { return new Symbol( Sym.EOF ); }
[^] { throw new Error("caracter inválido: "+yytext()+" na linha "+(yyline+1)+", coluna "+(yycolumn+1) ); }

```

Fonte: dados da pesquisa.

5. Criar um arquivo vazio: gera_analisador_sintatico.bat (Figura 33).

Figura 33 – gera_analisador_sintatico.bat

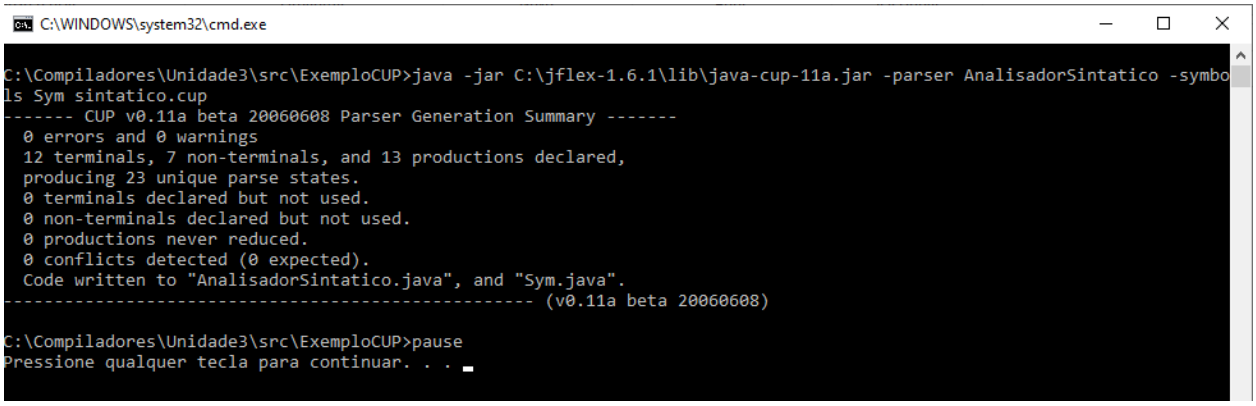
```
java -jar C:\jflex-1.6.1\lib\java-cup-11a.jar -parser AnalisadorSintatico -symbols Sym sintatico.cup
pause

```

Fonte: dados da pesquisa.

6. Ir na pasta Compiladores – Unidade 3 – src – ExemploCUP e executar o arquivo: gera_analisador_sintatico.bat, irá abrir o console e vai executar (Figura 34).

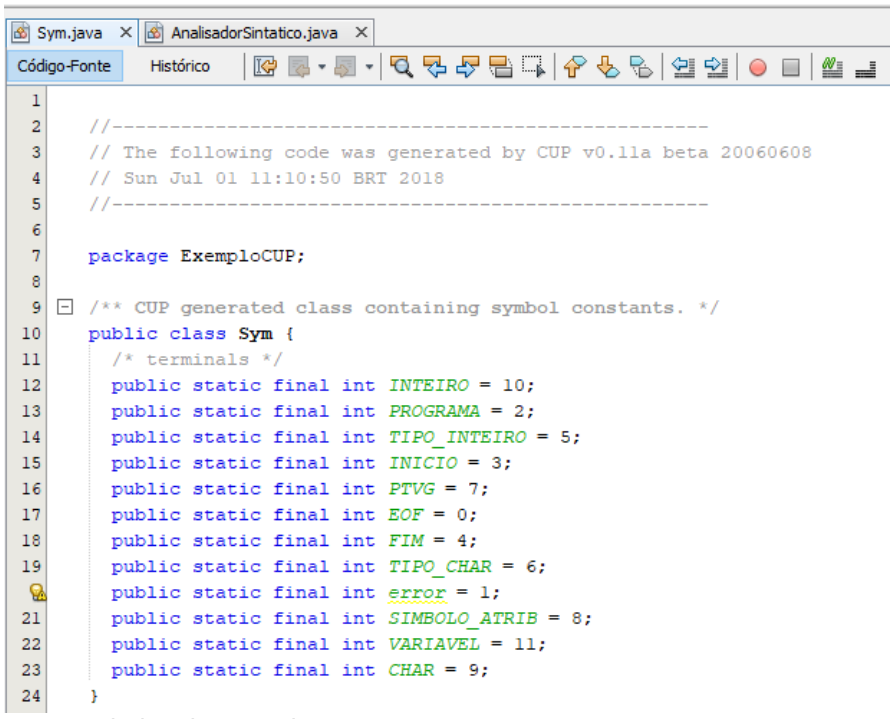
Figura *34* – Execução



Fonte: dados da pesquisa.

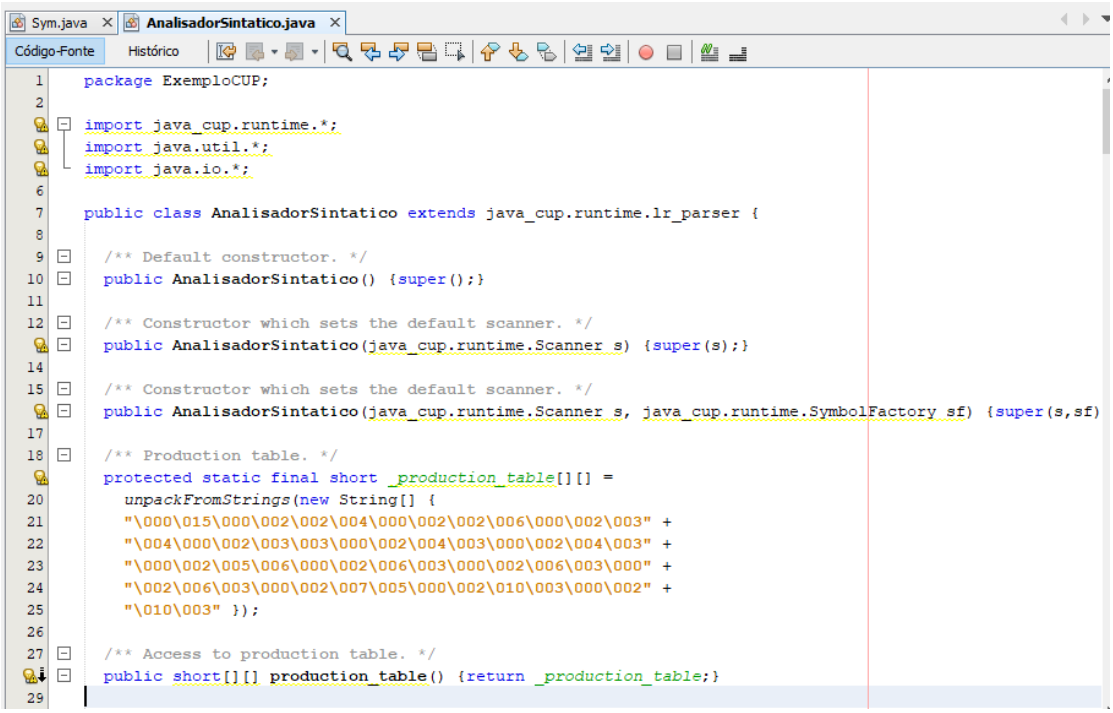
Voltando ao Projeto, verifique que foi criado as classes: Sym.java (Figura 35) e AnalisadorSintatico.java (Figura 36).

Figura 35 – Sym.java



Fonte: dados da pesquisa.

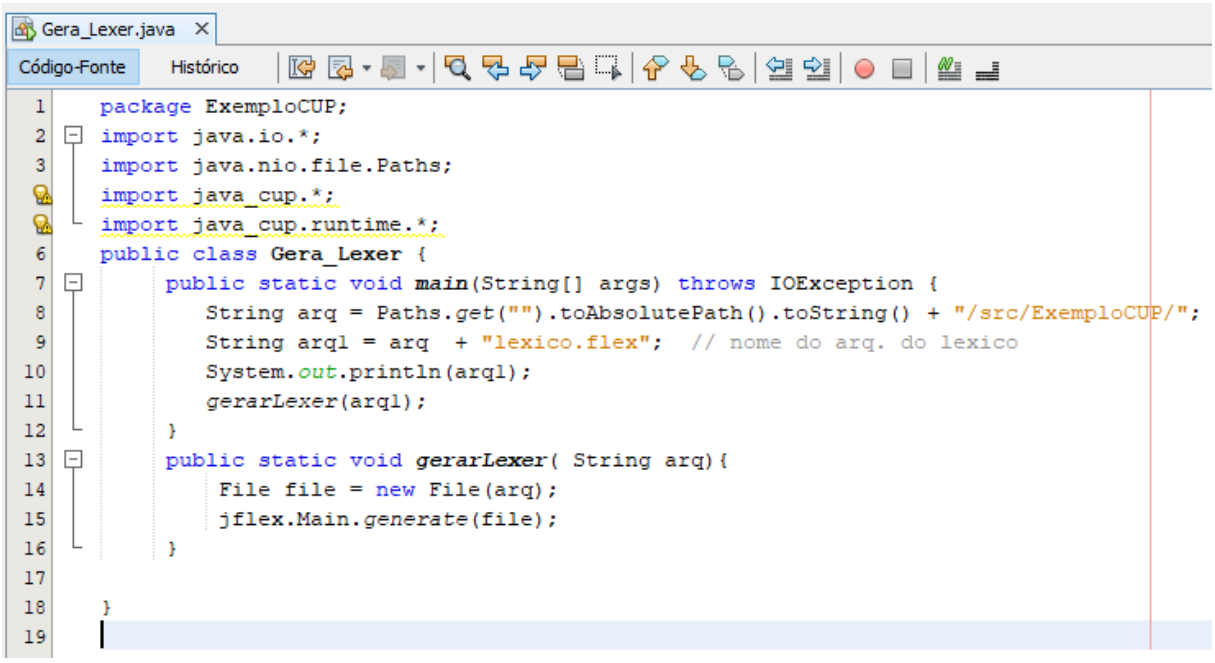
Figura 36 – AnalisadorSintatico.java



Fonte: dados da pesquisa.

7. Criar uma classe: GeraLexer.java (Figura 37).

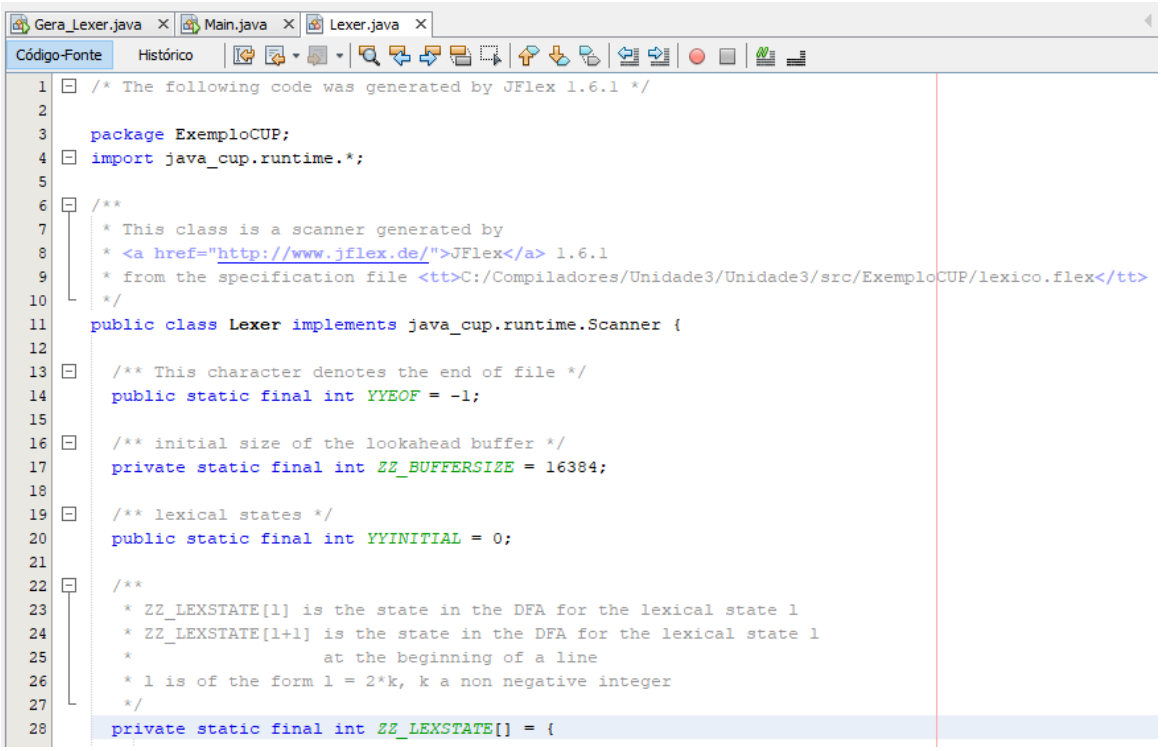
Figura 37 – GeraLexer.java



Fonte: dados da pesquisa.

8. Agora você pode executar o projeto, e o resultado será uma nova classe no seu projeto, denominada Lexer (Figura 38).

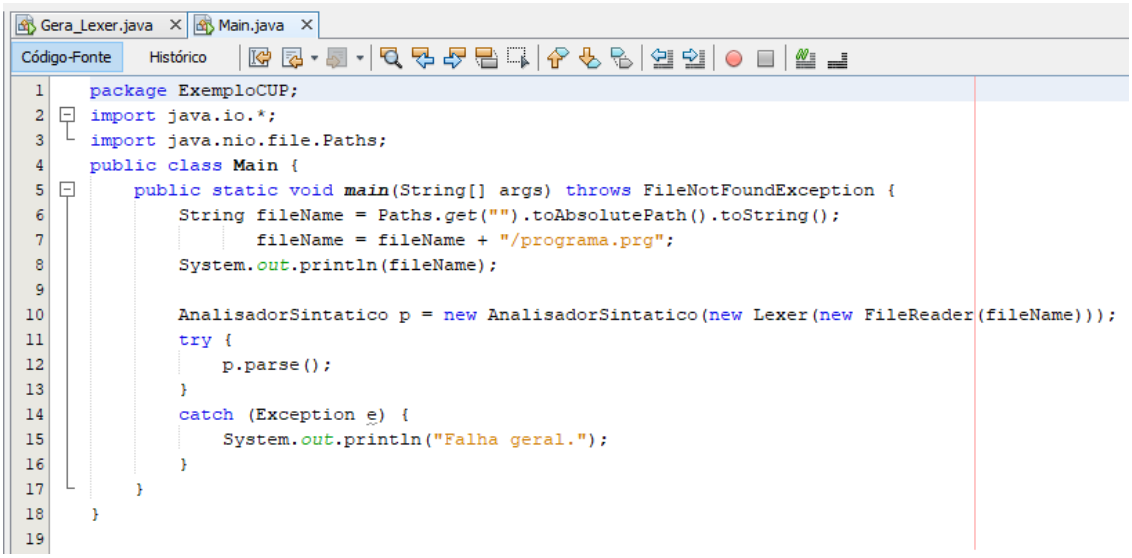
Figura 38 – Lexer.java



Fonte: dados da pesquisa.

9. Criar uma classe: Main.java (Figura 39).

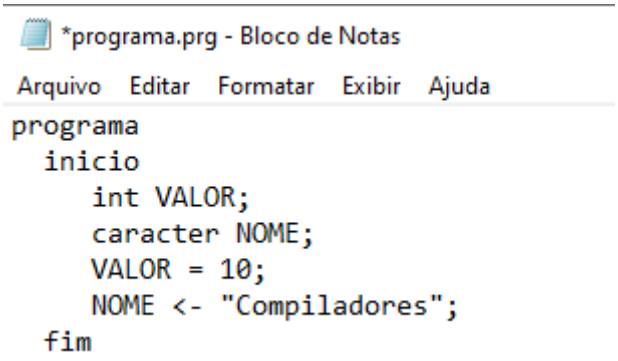
Figura 39 – Main.java



Fonte: dados da pesquisa.

10. Escrever um programa (bloco de notas): programa.prg (Figura 40).

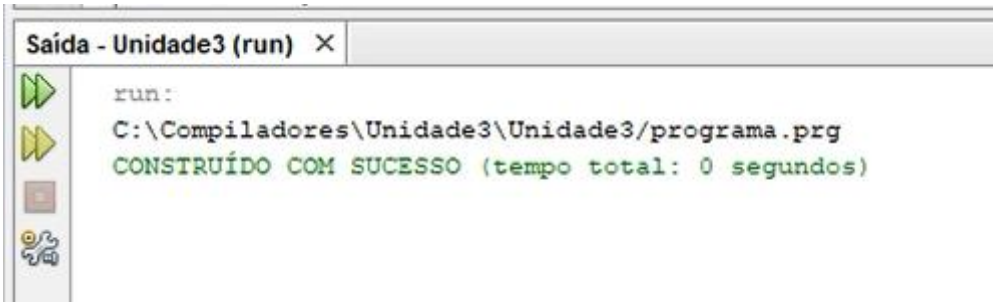
Figura 40 – programa.prg



Fonte: dados da pesquisa.

Feito o programa de teste, podemos finalmente executar o projeto, que pode ser visto na Figura 41.

Figura 41 – Saída



Fonte: dados da pesquisa.

UNIDADE 4 – GERAÇÃO DE CÓDIGO INTERMEDIÁRIO, DO CÓDIGO ALVO E OTIMIZAÇÃO

SEÇÃO 1 – GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Após a verificação semântica, é gerado o **código intermediário**, que é uma representação de mais baixo nível que o código fonte, mas ainda de alto nível em relação ao código de máquina. O código intermediário é geralmente usado para otimizações e para facilitar a geração do código de máquina. Existem diversos tipos de código intermediário, como árvores de expressão, notação polonesa reversa, código de três endereços, entre outros.

Segundo Fedozzi (2018), a criação de um **código intermediário** traz grandes vantagens para o projetista do compilador. Com a utilização do código intermediário, não é necessário escrever um compilador que traduza diretamente a representação da árvore sintática anotada para o código da máquina, ou que é um processo altamente complexo. Caso feito dessa forma, seria necessário escrever um compilador para cada tipo de arquitetura existente. No entanto, se uma fase de síntese for subdividida, criando um código intermediário que não esteja vinculado a uma determinada arquitetura, a tradução do código intermediário para várias plataformas será muito mais simples do que traduzir diretamente para o código da máquina.

Outra vantagem da geração do **código intermediário** está em permitir uma análise mais eficiente e detalhada da etapa mais complexa do desenvolvimento de um compilador: a otimização do código. Sem a utilização do código intermediário, a otimização do código seria muito difícil de ser realizada, conforme Fedozzi (2018).

Segundo Mogensen (2010), é crucial escolher o código intermediário transcendente, a fim de evitar a perda do objetivo pretendido. Para isso, o código intermediário deve garantir que:

a) a tradução da linguagem fonte, de alto nível, para a representação protegida seja fácil;

b) a tradução da representação representativa para o código de máquina também seja fácil;

c) a representação seja genuína para a otimização.

No entanto, conciliar os itens (a) e (b) pode ser difícil, e em alguns casos, para simplificar a construção do compilador, pode ser necessário ter vários códigos

intermediários. Isso é comum em compiladores com várias passagens, quando o processo de bootstrapping é aplicado repetidas vezes.

Segundo Aho (2007), há três formas básicas de **representação intermediária** (RI):

1. **Árvores sintáticas abstratas:** a árvore sintática abstrata (AST) é uma estrutura de dados hierárquica que representa a estrutura sintática de uma expressão ou declaração de programa. Cada nó na árvore representa um operador ou um elemento da expressão ou declaração, e as folhas representam os operandos ou variáveis. A AST é uma RI comum porque pode ser facilmente percorrida por um compilador para gerar código intermediário ou final.

2. **Notação pós-fixa:** a notação pós-fixa (ou notação polonesa reversa) é uma forma de escrever expressões matemáticas sem o uso de parênteses. Nessa notação, os operadores são escritos depois dos operandos, em vez de entre eles. Por exemplo, a expressão "3 + 4 * 5" seria escrita como "3 4 5 * +". A notação pós-fixa é uma RI porque pode ser facilmente convertida em código intermediário ou final, especialmente se armazenada em uma pilha.

3. **Código de três endereços:** o código de três endereços é uma forma simplificada de código intermediário que usa apenas três endereços para cada instrução. Cada instrução contém uma operação (como acompanhamento, aritmética, lógica, etc.) e dois operandos, que podem ser constantes, variáveis ou referências a outras instruções. Por exemplo, a instrução "x = a + b" seria escrita como "t1 = a + b; x = t1;". O código de três endereços é uma RI eficiente porque é fácil de gerar e pode ser facilmente otimizado antes de gerar o código final.

Vamos escrever o código em Java para converter uma expressão infixa em pósfixa, utilizando conceitos de orientação a objetos, fila, pilha e hash. O nosso objetivo é gerar uma RI na forma pós-fixa. Neste ponto, pressupomos que os analisadores sintáticos e semânticos já foram executados e a sentença a ser julgada está correta. A estrutura da nossa classe será:

```
import java.util.HashMap;
```

```
import java.util.Map;
import java.util.Stack;
import java.util.Queue;
import java.util.LinkedList;
public class Converte {
    private Map<String, Integer> precedencia;
    private Stack<String> pilhaOperadores;
    private Queue<String> filaSaida;
    public Converte() {
        precedencia = new HashMap<>();
        precedencia.put("+", 1);
        precedencia.put("-", 1);
        precedencia.put("*", 2);
        precedencia.put("/", 2);
        precedencia.put("^", 3);
        pilhaOperadores = new Stack<>();
        filaSaida = new LinkedList<>();
    }
    public String converteInfixaParaPosfixa(String expressao) {
        String[] tokens = expressao.split(" ");
        for (String token : tokens) {
            if (isOperando(token)) {
                filaSaida.offer(token);
            } else if (isOperador(token)) {
                while (!pilhaOperadores.empty() &&
temPrecedencia(pilhaOperadores.peek(), token)) {
                    filaSaida.offer(pilhaOperadores.pop());
                }
                pilhaOperadores.push(token);
            } else if (token.equals("(")) {
                pilhaOperadores.push(token);
            } else if (token.equals(")")) {
                while (!pilhaOperadores.empty() &&
```

```
!pilhaOperadores.peek().equals("(")) {
                    filaSaida.offer(pilhaOperadores.pop());
                }
                pilhaOperadores.pop();
            }
        }
        while (!pilhaOperadores.empty()) {
            filaSaida.offer(pilhaOperadores.pop());
        }
        return String.join(" ", filaSaida);
    }
    private boolean isOperando(String token) {
        return !isOperador(token) && !token.equals("(") && !token.equals(")");
    }
    private boolean isOperador(String token) {
        return token.equals("+") || token.equals("-") || token.equals("*") ||
token.equals("/") || token.equals("^");
    }
    private boolean temPrecedencia(String op1, String op2) {
        return precedencia.get(op1) >= precedencia.get(op2);
    }
}
```

O código apresenta a implementação de um algoritmo em Java para converter expressões na notação infixa para a notação pós-fixa, utilizando estruturas de dados como **HashMap**, pilha e fila.

A **classe Converte** tem três atributos: **precedência** que é um mapa que armazena a precedência de cada operador, **pilhaOperadores** que é uma pilha para armazenar os operadores e **filaSaida** que é uma fila para armazenar os operandos e operadores na notação pós-fixa.

O construtor da **classe Converte** inicializa o mapa **precedência** com as precedências de cada operador, inicializa a **pilha pilhaOperadores** e a **fila filaSaida**.

O método **converteInfixaParaPosfixa** é responsável por receber uma expressão na notação infixa como parâmetro e retornar a mesma expressão na

notação pós-fixa. Ele começa dividindo uma expressão em tokens usando o método split. Em seguida, itera pelos tokens verificando se cada um é um operando, um operador, um parêntese aberto ou um parêntese fechado.

Se o token for um operando, ele é adicionado à **fila filaSaida**. Se para um operador, é verificado se a **pilha pilhaOperadores** está vazia ou se o operador tem precedência maior ou igual ao topo da pilha. Caso a condição seja satisfeita, o operador no topo da pilha é removido e adicionado à **fila filaSaida**. Em seguida, o operador atual é adicionado à **pilha pilhaOperadores**.

Se o token for um parêntese aberto, ele é adicionado à **pilha pilhaOperadores**. Se para um parêntese fechado, é verificado se a **pilha pilhaOperadores** está vazia ou se o topo da pilha é um parêntese aberto. Se a condição for satisfatória, o topo da pilha é removido e adicionado à **fila filaSaida**.

Por fim, o método itera sobre a **pilha pilhaOperadores** e adiciona todos os operadores restantes na **fila filaSaida**. A expressão final na notação pós-fixa é formada pela emoção dos elementos da **fila filaSaida**, separados por espaço, usando o método join.

Os métodos privados **isOperando** são usados para verificar se um token é um operador **isOperador**, tem **Precedencia** operador ou se um operador tem precedência sobre outro, respectivamente.

UNIDADE 4 – GERAÇÃO DE CÓDIGO INTERMEDIÁRIO, DO CÓDIGO ALVO E OTIMIZAÇÃO

SEÇÃO 2 – GERAÇÃO DE CÓDIGO E OTIMIZAÇÃO DE CÓDIGO

Após a **geração do código intermediário**, é realizada a **otimização de código**, que visa melhorar a eficiência do código gerado, eliminando redundâncias e reorganizando as operações para que sejam executadas de forma mais eficiente.

Por fim, a última etapa é **a geração do código de máquina** a partir do código intermediário otimizado. Essa etapa é realizada pelo gerador de código, que traduz o código intermediário para o código de máquina específico da plataforma de destino.

De acordo com Fedozzi (2018), a primeira etapa da **geração de código** é a conversão da AST em uma linguagem protegida, que contém comandos mais simples do que a linguagem fonte, facilitando a conversão para a linguagem de máquina. Além disso, a otimização do código gerado também é importante para garantir que ele seja eficiente e utilize plenamente os recursos do processador. Embora não exista um método que garanta o código alvo mais eficiente, é possível garantir um código bom através da aplicação de técnicas de otimização.

Aho (2007) afirma que a fase de otimização é mais complexa do que a fase de geração de código, pois envolve a análise de várias propriedades de acordo com o tipo de estrutura gerada. Algumas dessas propriedades incluem a preservação do significado do programa, a interpretação do programa gerado e a explicação do esforço gasto na otimização em relação à eficiência do programa gerado. A eficiência média do programa é o que importa nesse ponto da otimização, e é mensurável pela taxa de crescimento (Big-Theta).

Em resumo, a **geração de código e otimização** são etapas importantes no processo de compilação, e embora a otimização seja mais complexa, é possível garantir um código bom e eficiente através da aplicação de técnicas de otimização e da análise dos cuidados das propriedades do programa gerado.

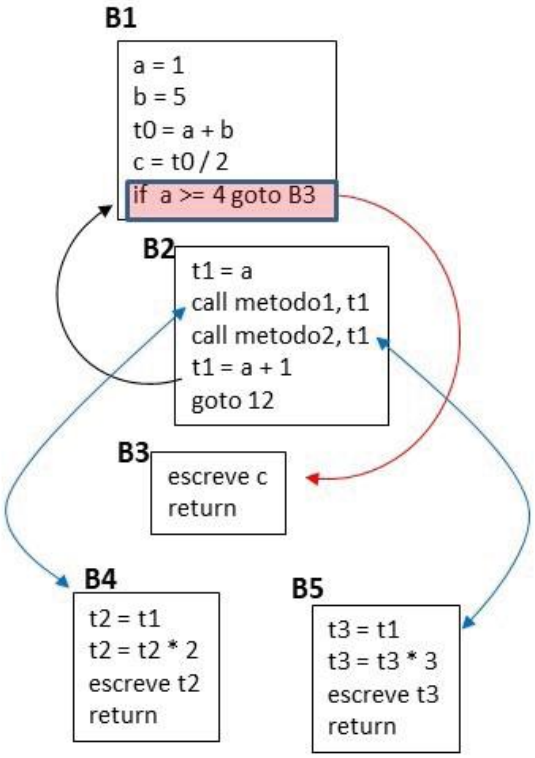
Conforme Fedozzi (2018), vamos entender o processo de otimização e a geração do código por meio de um exemplo. Seja a classe Exemplo.java a seguir:

```
public class Exemplo {
public static void main(String[] args){
```

```
        double a, b, c;
        a = 1;
        b = 5;
        c = ( a + b ) / 2;
        while (a < 4 ){
              Exemplo.metodo1(a);
                a++;
           }
       }
       public static void metodo1(double x){
            System.out.println(x*2);
            Exemplo.metodo2(x);
        }
        public static void metodo2(double x){
              System.out.println(x*3);
        }
     }
```

De acordo com Fedozzi (2018), vamos converter o código-fonte para a RI com 3 endereços e separar o código em blocos básicos. A Figura 42 mostra esta conversão e os blocos B1 a B5.

Figura 42 – Grafos de Fluxo de Controle (CFG) e Blocos Básicos (BB)



Fonte: Fedozzi (2018).

De acordo com Aho (2007), compreender as representações em blocos básicos e gráficos de fluxo é crucial para entender os algoritmos de três endereços, e os gráficos ajudam a coletar informações sobre o código intermediário durante a conversão para o código de máquina. Dessa forma, é fundamental entender o conceito de bloco básico (BB) para sistematizar o processo de otimização. Segundo Aho (2007), um bloco básico é uma sequência de instruções seguidas na qual o controle entra no início e sai no final, sem parar ou bifurcar, exceto no final.

Conforme Fedozzi (2018) é importante observar que a instrução de alto nível while foi convertida em um desvio condicional na conversão para três endereços. Nesse exemplo, é possível realizar a otimização na conversão para três endereços no bloco B4, conforme mostrado na Figura 43.

Figura 43 – Bloco B4 otimizado

Bloco b4

```
t2 = t1
t2 = t2 * 2
escreve t2
return
```

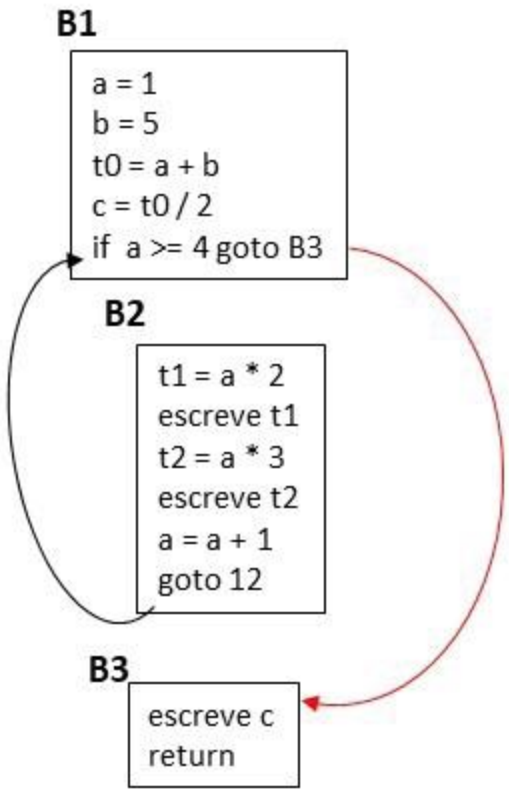
Bloco B4 - otimizado

```
t2 = t1 * 2
escreve t2
return
```

Fonte: Fedozzi (2018).

Conforme Fedozzi (2018), na Figura 44 o grafo de fluxo de controle está otimizado e representa a otimização final da RI, e, para isso, foram analisados todos os BBs e transformações no código intermediário para melhorar a eficiência do código.

Figura 44 – Grafos de Fluxo de Controle Otimizado



Fonte: Fedozzi (2018).

Observe o código original, à direita, e as otimizações, à esquerda, a seguir na Figura 45.

Figura 45 – Código x otimização

TRECHO DO CÓDIGO-FONTE	COMENTÁRIOS COM RELAÇÃO

`while (a < 4 ){`
`  Exemplo.metodo1(a);`
`  a++;`
`  }`
`}`
`public static void metodo1(double x){`
`  System.out.println(x*2);`
`  Exemplo.metodo2(x);`
`}`
 	
`public static void metodo2(double x){`
`  System.out.println(x*3);`
`}`
`}`
	A Chamada ao **metodo1** está dentro de um loop
	
	O **metodo1** chama o **metodo2**
	
	Assim no processo de otimização os blocos repetitivos foram otimizados dentro do loop
	E ficou:
	`t1 = a * 2`
	`escreve t1` ações do metodo1
	`t2 = t1 * 2`
	`escreve t2` ações do metodo2
	
	Portanto, as quatro linhas consistem em um bloco básico e com menos operações de memória e desvios.

Fonte: Fedozzi (2018).

A fase de geração de código é responsável por converter um RI em código executável na máquina de destino. Cooper (2014) define as seguintes ações nesta fase:

- **Seleção de instruções** - escolha de uma sequência de instruções da máquina-alvo que implemente as operações da RI;
- **Escalonamento das instruções** - definição da ordem em que as operações serão executadas;
- **Alocação de registros** - definição dos valores que serão armazenados nos registradores em cada ponto do programa.

Para cada uma dessas ações, o compilador deve reescrever o código intermediário para refletir as decisões tomadas. No caso da geração de código para a classe Exemplo.java, como o código fonte foi escrito em Java, a máquina alvo será a JVM e o conjunto de instruções da máquina será o bytecode.

De acordo com Fedozzi (2018), apesar do grande número de instruções que uma máquina pode ter, a classe Exemplo.java é simples e seu código em RI consiste em instruções básicas, as quais são desenvolvidas e demonstradas na

Figura 46.

Figura 46 – RI otimizado x Bytecode

RI – otimizado	*Bytecode*	Comentários
a =1	0: dconst_1 1: dstore_1 2: dload_1 3: ldc2_w #10	Carrega constante **1** na pilha Armazena na variável **a** Carrega variável **a** na pilha Carrega constante **4** na pilha
if a > = 4 goto 45	6: dcmpg 7: ifge 45	Compara variável **a** com **4** Se a >= 4 desvia para instrução 45
t1 = a	10: dload_1	Carrega variável **a** na pilha
2 t1 * 2	**11**: dstore_3 15: dload_3 16: ldc2_w #6 19: dmul	Armazena na variável **t1** (da pilha) Carrega variável **t1** na pilha Carrega constante **2** na pilha Multiplica **t1** * **2** (valores da pilha)
	12: getstatic #12 20: invokevirtual #13	System.out Método println – imprime valor da pilha
t2 = t1	23: dload_3 **24**: dstore 5 29: dload 5 31: ldc2_w #8 34: dmul	Carrega variável **t1** na pilha Armazena na variável **t2** (pilha) Carrega variável **t2** na pilha Carrega constante **3** na pilha Multiplica **t2** * **3** (valores da pilha)
	26: getstatic #12 35: invokevirtual #13	System.out Método println – imprime valor da pilha
a++	38: dload_1 39: dconst_1 40: dadd 41: dstore_1	Carrega variável a na pilha Carrega a constante **1** na pilha Adiciona **a** + **1** (valores da pilha) Armazena na variável a (pilha)
	42: goto 2	Fim do loop – retorna a instrução 2
	45: return	Fim do programa

Fonte: Fedozzi (2018).

UNIDADE 4 – GERAÇÃO DE CÓDIGO INTERMEDIÁRIO, DO CÓDIGO ALVO E OTIMIZAÇÃO

SEÇÃO 3 – ESPECIFICAÇÃO DE UMA PROPOSTA DE LINGUAGEM

Conforme Fedozzi (2018), a estrutura conceitual dos compiladores continua a mesma nos dias atuais, mas, como muitos outros sistemas complexos, eles não são mais monolíticos. Além disso, a portabilidade se tornou essencial diante da possibilidade de o código gerado precisar ser executado em várias plataformas. A época em que a elaboração levou minutos já passou. Hoje, os desenvolvedores esperam uma IDE (Integrated Development Environment) que apresenta rapidamente os erros no código-fonte e facilita ao máximo a edição de código.

De acordo com Fedozzi (2018), atualmente, o uso de conceitos aplicados em compiladores tem se tornado mais comum para resolver problemas, em comparação com o passado, principalmente devido à necessidade de sistemas autônomos e flexíveis e ao aumento da aplicação de tecnologias de inteligência artificiais (IA). Esse é o caso das Linguagens Específicas de Domínio (DSLs). Por exemplo, a criação de DSLs requer o uso de conceitos de construção de compiladores e sua utilização está se tornando cada vez mais frequente.

Segundo Kung (2008), as DSLs aumentam a expressividade do código, ou seja, tornam o código mais legível. Fowler (2012) define como pequenas linguagens focadas em um problema específico. Desenvolvedor DSLs requer conhecimento em análise léxica, sintática e semântica, especialmente se elas forem externas.

De acordo com o índice TIOBE de abril de 2023, as linguagens de programação mais populares são: Python, Java, C, C++, JavaScript, C #, R, PHP, Objective-C.

Vale ressaltar que o índice TIOBE é uma medida de popularidade baseada na quantidade de resultados de busca por uma determinada linguagem em motores de busca populares, como Google, Bing e Yahoo!. Portanto, a posição de uma linguagem no índice pode variar ao longo do tempo e não necessariamente reflete a qualidade ou a capacidade da linguagem em si.

Entre as linguagens de programação mais populares de acordo com o índice TIOBE, a maioria é interpretada, o que significa que o código-fonte é executado diretamente pelo interpretador sem passar por uma etapa de compilação. Essas

linguagens incluem Python, JavaScript, PHP, Ruby e Swift.

No entanto, também há linguagens compiladas entre as mais populares, como C, C# e Objective-C. Essas linguagens passam por uma etapa de compilação, onde o código-fonte é traduzido para uma linguagem de baixo nível (como o código de máquina) que pode ser executada diretamente pelo processador do computador.

Além disso, algumas linguagens como C++ e Java utilizam um modelo híbrido, onde o código-fonte é compilado para uma linguagem testada que pode ser interpretada em diferentes plataformas, permitindo que o código seja executado em diferentes sistemas operacionais e arquiteturas de processador.

Em geral, a escolha entre uma linguagem interpretada ou compilada depende das necessidades e objetivos do desenvolvedor e do projeto em questão. Linguagens interpretadas são geralmente mais fáceis de escrever e mais flexíveis, enquanto linguagens compiladas tendem a ter um desempenho melhor e ser mais seguras.

As novas linguagens de compiladores têm sido impulsionadas pelo aumento da necessidade de desenvolvimento de software mais eficiente, seguro e escalável. Algumas das principais tendências incluem:

1. **Compiladores de linguagem natural:** Estes compiladores permitem que os desenvolvedores escrevam código em linguagem natural, em vez de uma linguagem de programação específica. O compilador então converte o código natural em código executável. Isso pode tornar a programação mais acessível para pessoas que não têm formação em ciência da computação, além de tornar a comunicação entre equipes de desenvolvimento mais fácil.
2. **Compiladores de baixo código:** Estes compiladores permitem que os desenvolvedores criem aplicativos usando interfaces gráficas de usuário (GUIs) em vez de escrever código manualmente. O compilador então gera o código executável a partir da GUI. Isso pode ajudar a acelerar o desenvolvimento de aplicativos e reduzir a dependência de programadores altamente especializados.
3. **Compiladores baseados em inteligência artificial:** Esses compiladores usam aprendizado de máquina e outras técnicas de inteligência artificial para analisar o código-fonte e encontrar maneiras de otimizar o desempenho ou reduzir a quantidade de código necessário. Isso pode ajudar a melhorar a eficiência e a segurança do software.

4. **Compiladores para arquiteturas de hardware específicas:** Com o aumento da popularidade de tecnologias como a Internet das coisas (IoT) e a computação em nuvem, os compiladores estão se tornando mais solicitados para a especificação específica e outras arquiteturas de hardware. Isso pode ajudar a melhorar o desempenho e a eficiência do software em dispositivos específicos.
5. **Compiladores para programação quântica:** Com o crescente interesse em computação quântica, os compiladores estão sendo apresentados para traduzir código em linguagem de programação de alto nível em instruções que podem ser executadas em computadores quânticos. Esses compiladores têm o desafio de lidar com a natureza complexa e probabilística da computação quântica.

Referências

AHO, A.V.; SETHI R.; ULLMAN J.D. Compiladores: princípios, técnicas e ferramentas. São Paulo: Pearson, 2007.

APPEL, A.W. Modern Compiler Implementation in Java. Cambridge University Press, 2002.

BROWN, D.; LEVINE, J.; MASON, T. Lex & Yacc. Sebastopol – CA: O'Reilly Media, 2012.

CHOMSKY, N. Estruturas Sintacticas. Lisboa: Edições 70, 1980.

COOPER, K. D.; TORCZON, L. Construindo Compiladores. Rio de Janeiro: Elsevier, 2014.

COX, R. A correspondência de expressões regulares pode ser simples e rápida. Jan. 2007. Disponível em: <https://swtch.com/~rsc/regexp/regexp1.html>. Acesso em: 4 abr. 2023.

DELAMARO, M.E. Como construir um compilador utilizando ferramentas Java. São Paulo: Novatec, 2004.

GRUNE, D. et al. Modern Compiler Design. New York: Springer, 2012.

FORBELLONE, A.L.V.; EBERSPÄCHER, H.F. Lógica de Programação: a construção de algoritmos e estruturas de dados. São Paulo: Pearson Prentice Hall, 2005.

FEDOZZI, R. Compiladores. Londrina: Editora e Distribuidora Educacional. 2018.

KLEIN, G.; ROWE, S.; DÉCAMPS R. JFlex User's Manual - The Fast Lexical Analyser Generator. 2015. v. 1.6.1. Disponível em: <http://jflex.de/manual.html>. Acesso em: 2 abr. 2023.

MENEZES, P.B. Linguagens formais e autômatos. Porto Alegre: Sagra Luzzatto, 2005.

PRICE, A.M. de A.; TOSCANI, S. S. Implementação de linguagens de programação: Compiladores. Porto Alegre: Sagra-Luzzatto, 2001. (Série Livros Didáticos – nº 9).
RICARTE, I., Introdução a Compilação. 1. ed. Editora Elsevier, 2008.

SÁ, C. C.; SILVA, M.F. Haskell: uma abordagem prática. São Paulo: Novatec, 2006.

SCOWEN, R. S. Information technology: Syntactic metalanguage — Extended BNF. Middlesex: ISO/IEC 14997, 1996.

SEBESTA, R.W. Conceitos de Linguagens de Programação. São Paulo: Bookman, 2011.

TUCKER, A.; NOONAN, R. Linguagens de programação: princípios e paradigmas. 2 Porto Alegre: McGraw-Hill, 2009.