UEL – UNIVERSIDADE ESTADUAL DE LONDRINA
CCE – CENTRO DE CIÊNCIA EXATAS
CIÊNCIA DA COMPUTAÇÃO

GABRIEL VIANA POLETTI
GABRIEL ANGELO PEREZ GASPARINI SABAUDO
GUILHERME HENRIQUE GONÇALVES SILVA
JOSÉ VICTOR ANDREI DALTO MORENO

ANÁLISE LÉXICA SINTÁTICA E SEMÂNTICA

LONDRINA – PR
2020

GABRIEL VIANA POLETTI
GABRIEL ANGELO PEREZ GASPARINI SABAUDO
GUILHERME HENRIQUE GONÇALVES SILVA
JOSÉ VICTOR ANDREI DALTO MORENO

ANÁLISE LÉXICA SINTÁTICA E SEMÂNTICA

Trabalho solicitado pela professora Cinthyan da disciplina Paradigmas de Computação, do curso de Ciência de Computação, turno integral da instituição Universidade Estadual de Londrina.

LONDRINA – PR

2020

RESUMO

Linguagens de programação são notações para se descrever computações para pessoas e para máquinas. Cada vez mais o mundo que conhecemos depende de linguagens de programação, pois o software executado em todos os computadores foi escrito em alguma linguagem de programação, mas antes que possa rodar, um programa primeiro precisa ser traduzido para formato que lhe permita ser executado por um computador.

Os sistemas de software que fazem essa tradução são denominados compiladores. Este artigo tem como objetivo analisar algumas poucas ideias básicas que podem ser utilizadas na construção de tradutores para uma grande variedade de linguagens de máquinas. E analisar qual a contribuição da utilização dos softwares educacionais na disciplina de Compiladores a fim de analisar como estes instrumentos podem auxiliar no processo de aprendizagem desta disciplina.

ABSTRACT

Programming languages are notations for describing computations for people and machines. Increasingly the world we know depends on programming languages, since the software running on all computers was written in some programming language, but before it can run, a program must first be translated into a format that allows it to be executed by a computer.

The software systems that do this translation are called compilers. This paper aims at analyzing a few basic ideas that can be used in the construction of translators for a wide variety of machine languages, and to analyze what is the contribution of the use of educational software in the discipline of Compilers in order to analyze how these instruments can help in the learning process of this discipline.

Sumário

1 INTRODUÇÃO

As linguagens de programação permitem instruções humanas (palavras símbolos e números) para escrever o programa de computador, para instruir o processador ou o hardware em geral para executar uma determinada tarefa, fazendo com que o programador não tenha que se preocupar com o hardware somente com o software.

Todo software executado em todos os computadores foi escrito em alguma linguagem de programação. Mas, antes que possa rodar, um programa primeiro precisa ser traduzido para um formato que lhe permita ser executado por um computador.

O computador é incapaz de executar o código escrito pelo programador em uma linguagem de programação, só compreende e executa instruções em binário. Os sistemas de software que fazem essa tradução são denominados de compiladores e interpretadores.

2 PROCESSADORES DE LINGUAGEM

"Compilador e interpretadores é um programa que recebe como entrada um programa em uma linguagem de programação (linguagem fonte) e o traduz para um programa equivalente em outra linguagem (linguagem objeto)" (Aho, 2008, p.1). Com isso, um dos principais objetivos do compilador e interpretador é relatar quaisquer erros no programa fonte que podem acontecer durante esse processo de tradução. No entanto, a forma que isso que isso acontece é diferente para cada um. Segue abaixo um comparativo entre compiladores e interpretadores:

2.1 Características de Compiladores

- O compilador lê e analisa linha a linha em busca de erros.
- Se não tiver nenhum erro no código do programador, traduz tudo para código de máquina.
- Salvar o código traduzido em um arquivo. (programa executável ou arquivo binário, no Windows é o arquivo .exe).
- O arquivo traduzido, agora pode ser executado pelo usuário.
- Linguagem C é compilada.
- Os programas são autossuficientes.
- Os programas são mais velozes.
- Compilação pode ser um processo demorado se o código for muito grande.
- O programa compilado roda em plataforma específica. Um programa compilado para o Windows só vai rodar no Windows.

2.2 Características de Interpretadores

- Linguagem PHP ou Python são linguagem interpretadas.
- O processo de interpretação traduz e executa o código simultaneamente linha a linha.
- Não é gerado um arquivo final executável.
- O arquivo executável já é o próprio código.
- Independentemente do tamanho do programa, já pode ser executado imediatamente.
- Normalmente Multiplataforma.
- Programas mais lentos. Toda vez que for executar o programa vai ter que traduzir cada linha e só depois executar.
- Necessita do interpretador.

3 EVOLUÇÃO DAS LINGUAGENS DE PROGRAMAÇÃO

Os processadores executam instruções, entretanto, existem várias arquiteturas e processadores diferentes (MIPS, SPARV, AMD64, ARM, x86, x86_64). As arquiteturas mudam completamente a forma como os processadores executam as instruções.

De acordo com (Vasconcelos, A. em “Introdução a Arquitetura de Computadores”), um processador só entende a linguagem de máquina que seria o código binário. Cada processador tem sua linguagem de máquina.

Os primeiros computadores eletrônicos apareceram na década de 1940 e eram programados em linguagem de máquina, por sequência de 0s e 1s que diziam ao computador quais operações deveriam ser executadas e em que ordem. As operações em si eram de muito baixo nível: mover dados de um local para outro, somar o conteúdo de dois registradores e assim por diante. Esse tipo de programação era lento, cansativo e passível de erros.

Uma linguagem de baixo nível segue as características do processador, com isso, a forma de como o programador vai programar nessa linguagem muda completamente dependendo do processador, pois as arquiteturas de processadores têm sua própria linguagem de máquina. As linguagens de baixo nível trabalham com conjuntos de instruções, e esses conjuntos de instruções mudam de uma arquitetura para outra. Com isso, a forma de como o programador vai trabalhar com essa linguagem muda de uma arquitetura para a outra.

Definição de linguagem de baixo nível: São linguagem extremamente dependentes das características das arquiteturas de processadores. Com isso, elas seguem as características do processador que manipulam diretamente das instruções definidas naquela arquitetura de processador.

Linguagem de máquina e de montagem (Assembly) são consideradas de baixo nível. Já as linguagens de alto nível (C, C++, Java, Java script, Python, PHP).

A forma da escrita do código em linguagem de alto nível não depende com instruções diretas do processador, com isso, não importa com seja a arquitetura do processador. É uma linguagem simplificada para poder criar programas sem a preocupação do hardware.

3.1 Projeto de novas arquiteturas de computador

Nos primeiros projetos de arquiteturas de computadores, os compiladores só eram desenvolvidos após a construção das máquinas. Mas isso mudou. Como o usual é programar em linguagem de alto nível, o desempenho de um sistema de computação é determinado não somente por sua velocidade, mas também pela forma como os compiladores podem explorar seus recursos. Assim, no desenvolvimento de arquiteturas de computadores modernos, os compiladores são desenvolvidos no estágio e projeto do processador, e o código compilado, executando em simuladores. É usado para avaliar os recursos arquitetônicos propostos. (ESCOLA, Equipe Brasil. "Revolução do Computador").

4 ESTRUTURA DE UM COMPILADOR

Podemos resumir que existe duas partes de um compilador: análise e síntese.

Aho (2008, p.3) aponta que a parte de análise impõe uma estrutura gramatical sobre o programa fonte, para detectar se o programa fonte esta sintaticamente mal formatado ou semanticamente incorreto, então ele precisa oferecer mensagens esclarecedoras, para que o usuário possa tomar ações corretivas. Além de coletar informações sobre o programa fonte e as armazenar tabela de símbolos. Essa parte tem o nome de front-end. É tipicamente dividida em análise léxica, análise sintática e análise semântica.

A parte de síntese constrói o programa objeto desejado a partir das informações na tabela de símbolos e requer as técnicas mais especializadas. Entre essas técnicas estão geradores do código intermediário, otimização do código e por fim o gerador de código. Essa parte tem o nome de back-end.

Se examinarmos o processo de compilação detalhadamente, veremos que ele é desenvolvido como uma sequência de fase. As principais fases que o compilador executa ao formar um código fonte de um programa em um programa executável seria: análise léxica, análise sintática e análise semântica.

5 ANÁLISE LÉXICA

A princípio, é importante esclarecer o que significa léxico. Léxico é um conjunto de palavras pertencentes a um determinado idioma, e por isso sua utilização num compilador é primordial.

5.1 Objetivo

A fase de análise léxica (ou analisador léxico, ou *lexer*) em compiladores tem o propósito de transformar o programa-fonte de uma sequência de carácteres em uma sequência de tokens.

Os *tokens* são sequências logicamente coesas de caracteres inclusas num programa, e representam um único símbolo (Tucker, 2008).

Os comandos de condição como *if* ou *else* podem ser usados como exemplos de *tokens.* Neste caso, a análise léxica ignora qualquer linha ou espaço em branco que não envolva os *tokens*.

De acordo com (Sebesta, 2011), o lexer é considerado uma fase de compilação por algumas questões que abordam sua eficiência. Tais como: Ele é baseado num modelo de máquina mais simples e rápido do que o modelo de contexto livre usado na análise sintática; A maioria do tempo consumido na leitura de um código fonte é por conta do *lexer*, então qualquer melhoria que possa encurtar e simplificar o código será de grande ajuda na compilação.

Historicamente falando, o lexer pôde efetuar grandes mudanças no processo de compilação. Antes da criação da tabela ASCII, havia um conjunto de caracteres único para cada computador, dificultando a leitura deles. O *lexer* fez com que portar um compilador para novas máquinas fosse muito mais fácil.

5.2 Utilização

Basicamente, o analisador léxico faz um escaneamento do programa fonte caractere por caractere, traduzindo-os em uma sequência de símbolos. Assim, o lexer coleta e reúne caracteres de maneira lógica e atribui códigos internos a estes agrupamentos de caracteres, de acordo com sua estrutura (Sebesta, 2011). Estes agrupamentos podem ser chamados de *lexemas*. Além disso, o analisador léxico

insere *lexemas* para nomes definidos pelo usuário na tabela de símbolos, que podem ser usados mais tarde em fases posteriores da compilação.

A seguir, é possível observar um exemplo de como *tokens* e *lexemas* se comportam numa sentença de atribuição:

media = a + b / 2;

Token	***Lexema***
IDENT	media
ASSIGN_OP	=
IDENT	a
ADD_OP	+
IDENT	b
DIV_OP	/
INT_LIT	2
SEMICOLON	;

Os analisadores léxicos utilizam os *lexemas* de uma cadeia de entradas definidas, e produzem os tokens correspondentes.

Neste processo, o *lexer* evita deixar comentários e espaços em branco nos *lexemas*, visto que estes não possuem relevância nesta etapa.

Antigamente, os analisadores léxicos processavam por inteiro um programa e então era produzido um arquivo de *tokens e l*exemas. Atualmente, grande parte dos analisadores são subprogramas que localizam o lexema seguinte no input, geram seu *token* associado e o retornam para seu chamador, o analisador sintático (Sebesta, 2011). De maneira geral, a saída da fase léxica é o programa de entrada da análise sintática.

5.3 Construção

Com base em (Sebesta, 2011), existem 3 abordagens para construir um analisador léxico, que serão explicadas abaixo:

Primeiramente, escrever uma descrição precisa e formal dos padrões conhecidos de *tokens.* Estas descrições coletadas serão usadas como entradas para um software de escolha da pessoa, que gera automaticamente um analisador léxico. Há um software muito conhecido e muito usado, chamado Lex, e faz parte dos sistemas UNIX.

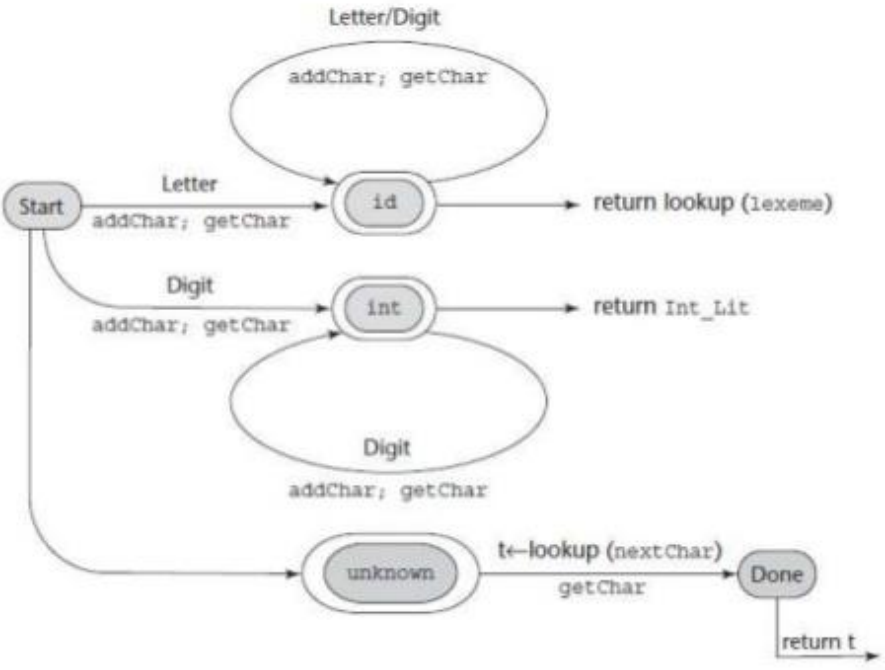
A seguir, é necessário produzir um diagrama de transição de estados que possa descrever o padrão de *tokens* conhecidos da linguagem, em seguida implementar o código deste diagrama.

Um diagrama de transição de estados é uma representação do estado ou situação em que um objeto pode se encontrar no decorrer da execução de processos de um sistema.

Por fim, deve-se descrever o diagrama de transição de estados que descreva os padrões de *tokens* da linguagem e construir manualmente uma implementação baseada em tabela de diagrama de estados.

A forma com que os diagramas de transição de estados são usados em analisadores léxicos os classificam como uma classe de máquinas matemáticas denominadas de autômatos finitos (Aho, 2008, p.1).

Autômatos finitos podem ser projetados para reconhecer uma classe de linguagens chamadas de linguagens regulares (Sebesta, 2011). Resumindo, os *tokens* de uma linguagem de programação formam uma linguagem regular, e o analisador léxico é o autômato finito. Para maior entendimento, segue abaixo uma ilustração de como funciona a construção de um analisador léxico com um diagrama de estados:



O diagrama ilustrado inclui as ações necessárias em cada transição do diagrama de estados.

As funções addChar e getChar serão responsáveis por coletar o digito de informado pelo usuário. A addChar terá também a responsabilidade de obter a próxima entrada de caractere e colocá-la na variável global nextChar no vetor *lexeme*. O subprograma getChar também deverá determinar a classe do caractere de entrada e colocá-la na variável global charClass.

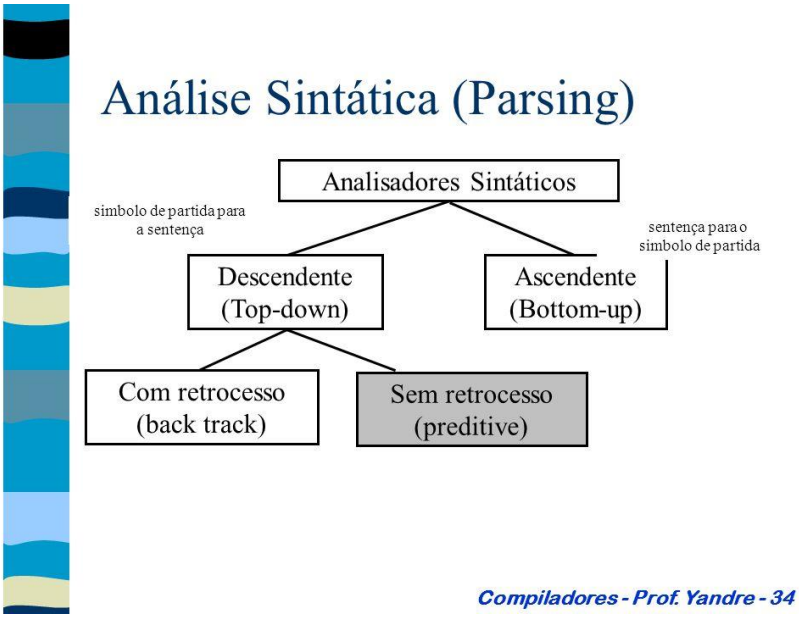
O *lexema* a ser construído pelo *lexer* (análisador léxico), que pode ser implementado como um vetor ou uma cadeia de caracteres, é a função chamada *lexeme*.

O caractere inserido em nextChar será adicionado para o vetor *lexeme* através da função addChar.

Por fim, a função *lookup* terá a função de validar um código de *token* para os *tokens* de caracteres únicos, são estes os símbolos como parênteses, ou operadores de aritmética. Os códigos de *token* são números inteiros atribuídos a estes símbolos pelo desenvolvedor do programa.

6 ANÁLISE SINTÁTICA

A segunda fase da compilação é chamada de análise sintática, também conhecida como parsing. Podemos utilizar os dois termos como sinônimos. Este processo trabalha em conjunto com a análise lexical. Analisadores sintáticos para linguagens de programação constroem árvores de análise sintática para programas informados. Em alguns casos, a árvore de análise sintática é apenas construída implicitamente, significando que talvez apenas um percurso da árvore seja gerado (Sebesta, 2011). Mas, em todos os casos, as informações necessárias para criar a árvore de análise sintática são criadas durante a análise. Tanto árvores de análise sintática quanto derivações incluem todas as informações sintáticas necessárias para um processador de linguagem, ou seja, o "analisador sintático" é um programa de computador que realiza a função de carregar os dados de entrada e construir uma estrutura de dados com eles.



A entrada de dados que é analisada pelo analisador é normalmente um código de uma linguagem de programação, mas podem ser também textos em linguagem natural, e nesse caso não é construída uma árvore de análise, mas só são extraídas algumas partes do texto. As funções dos analisadores variam desde analisar comandos simples de um código a programas muito complexos. Uma forma importante de realizar a análise é usando expressões regulares, onde uma expressão regular define uma linguagem regular e um mecanismo de expressão regular gerando automaticamente um analisador para a linguagem.

Em alguns casos as expressões regulares são usadas antes da própria análise sintática, como etapa da análise lexical cuja saída será utilizada pelo analisador sintático.

Existem dois objetivos distintos da análise sintática: primeiro, verificar o programa de entrada para determinar se ele está sintaticamente correto. Quando um erro for encontrado, o analisador deve produzir uma mensagem de diagnóstico e se recuperar (Sebesta, 2011). A recuperação significa que ele deve voltar a um estado normal e continuar a análise do programa de entrada. Esse passo é necessário para que o compilador encontre o máximo de erros possível durante uma análise do programa de entrada. Se não for feita corretamente, a recuperação de erros pode criar mais erros, ou pelo menos mais mensagens de erro. Os erros de programação mais comuns podem ocorrer em diferentes níveis, no caso dos erros sintáticos, incluem ponto-e-vírgulas mal colocados ou chaves extras ou faltando. O segundo objetivo da análise sintática é produzir uma árvore de análise sintática completa, ou ao menos percorrer a estrutura da árvore, para uma entrada sintaticamente correta (Sebesta, 2011). A árvore de análise sintática (ou seu percurso) é usada como base para a tradução.

Os analisadores sintáticos são categorizados de acordo com a direção na qual eles constroem as árvores de análise sintática. As duas amplas classes de analisadores sintáticos são os analisadores descendentes, nos quais a árvore é construída a partir da raiz em direção às folhas, e ascendentes, a partir das folhas em direção à raiz (Sebesta, 2011).

6.1 Análise sintática descendente

O método de análise sintática descendente constrói a árvore de derivação para a cadeia de entrada de cima para baixo, ou seja, da raiz para as folhas, criando os nós da árvore em pré-ordem. Neste processo, análise sintática descendente pode ser vista como o método que produz uma derivação mais à esquerda para uma cadeia de entrada (Aho, 2008, p.1).

A cada passo de uma análise sintática descendente, o problema principal é determinar a produção a ser aplicada para um não terminal, digamos *A.* Quando uma produção-A é escolhida, o restante do processo de análise consiste em “casar” os símbolos terminais do corpo da produção com a cadeia de entrada.

Ainda no tópico sobre sintática descendente, devemos mencionar a análise sintática descendente recursiva. Um método de análise de descida recursiva consiste em um conjunto de procedimentos, um para cada não-terminal da gramática. A execução começa com a ativação do procedimento referente ao símbolo inicial da gramática, que para e anuncia sucesso se o seu corpo conseguir escandir toda a cadeia de entrada.

O método de análise sintática descendente pode exigir retrocesso, ou seja, pode demandar voltar atrás no reconhecimento, fazendo repetidas leituras sobre a entrada. As construções presentes nas linguagens de programação raramente necessitam do retrocesso durante suas análises, de modo que os analisadores com retrocesso não são usados com frequência. Até mesmo em situações que envolvem o reconhecimento de linguagem natural, o retrocesso não é muito eficiente, e os métodos dos baseados em tabelas, como o algoritmo de programação dinâmica, ou o método de Early, são os preferidos (Aho, 2008, p.1).

6.2 Análise sintática ascendente

A análise sintática ascendente corresponde a construção de uma árvore de derivação para uma cadeia de entrada a partir das folhas (a parte de baixo) em direção a raiz (o topo) da árvore. Embora a árvore de derivação seja utilizada para descrever os métodos de análise, um *front-end* pode executar uma tradução diretamente, portanto, na prática, ela nunca é efetivamente construída.

Dentro do tópico de sintática ascendente, ainda devemos mencionar as reduções. Podemos pensar na análise ascendente como o processo de “reduzir” uma cadeia *w* para o símbolo inicial da gramática (Aho, 2008, p.1). Em cada passo da *redução*, uma subcadeia específica, casando-se com o lado direito de uma produção, é substituída pelo não terminal na cabeça dessa produção. As principais decisões relacionadas com a análise sintática ascendente em cada passo de reconhecimento são: determinar quando reduzir e determinar a produção a ser usada para que a análise prossiga.

Além das reduções ainda em sintática ascendente, temos também o *Poda do Handle.* A análise sintática ascendente ao ler a entrada, da esquerda para a direita, constrói uma derivação mais à direita ao inverso (Aho, 2008, p.1). Informalmente um “handle” de uma cadeia de símbolos é uma subcadeia que se

casa com o corpo de uma produção, e cuja redução para o não terminal do lado esquerdo representa um passo da derivação a direita ao inverso.

6.3 Complexidade da análise sintática

Os algoritmos de análise sintática que trabalham para qualquer gramática não ambígua são complicados e ineficientes. Na prática, a complexidade de tais algoritmos é O (n 3), ou seja, a quantidade de tempo que eles levam é na ordem do cubo do tamanho da cadeia a ser analisada. Essa quantidade de tempo relativamente grande é necessária porque esses algoritmos geralmente precisam voltar e reanalisar parte da sentença que está sendo analisada. A reanálise sintática é necessária quando o analisador sintático tiver cometido um erro no processo de análise. Voltar o analisador sintático também requer que a parte com problemas da árvore de análise sintática que está sendo construída (ou seu percurso) deve ser desmantelada e reconstruída. Os algoritmos O (n 3) normalmente não são úteis para processos práticos, como a análise sintática para um compilador, porque são muito lentos (Sebesta, 2011). Nesse tipo de situação, os cientistas da computação buscam algoritmos mais rápidos, apesar de não tão gerais.

A generalidade é trocada pela eficiência. Em termos de análise sintática, foram encontrados algoritmos mais rápidos funcionando para apenas um subconjunto do conjunto de todas as gramáticas possíveis. Esses algoritmos são aceitáveis desde que o subconjunto inclua gramáticas que descrevem linguagens de programação (na verdade, a classe inteira das gramáticas livres de contexto não é adequada para descrever toda a sintaxe da maioria das linguagens de programação.) Todos os algoritmos usados pelos analisadores sintáticos dos compiladores comerciais têm complexidade O(n), ou seja, o tempo que eles levam é linearmente relacionado ao tamanho da cadeia que está sendo analisada sintaticamente (Sebesta, 2011).Tais algoritmos são amplamente mais eficientes do que os algoritmos O (n 3).

7 ANÁLISE SEMÂNTICA

Sabemos que a análise sintática consegue verificar se uma expressão segue às regras de formação de uma dada gramática, porém, algumas regras como, declaração de uma variável ou verificação do tipo de uma expressão ou variável, podem passar despercebidas.

De acordo com Ivan Ricarte (2003, p. 81)

> O objetivo da análise semântica é trabalhar nesse nível de inter-relacionamento entre partes distintas do programa. As tarefas básicas desempenhada durante a análise semântica incluem a verificação de tipos, a verificação do fluxo de controle e a verificação da unicidade da declaração de variáveis. Dependendo da linguagem de programação, outros tipos de verificações podem ser necessários.

As principais motivações para se definir com precisão uma linguagem de programação são:

1. Fornecer aos programadores uma definição oficial do significado de todas as construções da linguagem.

2. Fornecer aos escritores de compiladores uma definição oficial do significado de todas as construções, evitando, assim, diferenças nas implementações.

3. Fornece uma base para a padronização da linguagem.

7.1 Princípios básicos

O estudo de linguagens de programação, como o estudo de linguagens naturais, pode ser dividido em exames acerca da sintaxe e da semântica.

A sintaxe de uma linguagem de programação é a forma de suas expressões, sentenças e unidades de programas. Sua semântica é o

significado dessas expressões, sentenças e unidades de programas. Por exemplo, a sintaxe de uma sentença while em Java é:

while (expressão_booleana) sentença.

A semântica desse formato de sentença é que quando o valor atual da expressão booleana for verdadeiro, a sentença dentro da estrutura é executada. Caso contrário, o controle continua após a construção while. O controle retorna implicitamente para a expressão booleana para repetir o processo.

Apesar de normalmente serem separadas para propósitos de discussão, a sintaxe e a semântica são bastante relacionadas. Em uma linguagem de programação bem projetada, a semântica deve seguir diretamente a partir da sintaxe; ou seja, a aparência de uma sentença deve sugerir o que a sentença realiza. Descrever a sintaxe é mais fácil do que descrever a semântica, especialmente porque uma notação aceita universalmente está disponível para a descrição de sintaxe, mas nenhuma ainda foi desenvolvida para descrever semântica (Sebesta, 2011).

7.2 Associatividade

Quando uma expressão inclui dois operadores que têm a mesma precedência (como * e / normalmente têm) – por exemplo, A / B * C – uma regra semântica é necessária para especificar qual dos operadores deve ter precedência. Essa regra é chamada de *associatividade* (Sebesta, 2011).

7.3 Semântica estática

A semântica estática de uma linguagem é apenas indiretamente relacionada ao significado dos programas durante a execução; em vez disso, ela tem a ver com as formas permitidas dos programas (sintaxe em vez da semântica). Muitas regras de semântica estática de uma linguagem definem suas restrições de tipos. A semântica estática é assim chamada porque a análise necessária para verificar essas especificações pode ser feita em tempo de compilação.

Por causa dos problemas da descrição de semântica estática com BNF, uma variedade de mecanismos mais poderosos foi criada para essa tarefa. Um

desses mecanismos, as gramáticas de atributos, foi projetado por Knuth (1968a) para descrever tanto a sintaxe quanto a semântica estática de programas.

As gramáticas de atributos são uma abordagem formal, usada tanto para descrever quanto para verificar a corretude das regras de semântica estática de um programa. Apesar de elas não serem sempre usadas de maneira formal no projeto de compiladores, os conceitos básicos das gramáticas de atributos são ao menos informalmente usados em todos os compiladores (Sebesta, 2011).

7.4 Semântica dinâmica

Semântica dinâmica, ou o significado, das expressões, sentenças e unidades de programa de uma linguagem de programação. Por causa do poder e da naturalidade da notação disponível, descrever a sintaxe é algo relativamente simples. Por outro lado, nenhuma notação ou abordagem universalmente aceita foi inventada para semântica dinâmica.

Se existisse uma especificação precisa de semântica de uma linguagem de programação, os programas escritos na linguagem poderiam, potencialmente, ser provados corretos sem a necessidade de testes.

Os desenvolvedores de software e os projetistas de compiladores normalmente determinam a semântica das linguagens de programação pela leitura de explicações em linguagem natural disponíveis nos manuais da linguagem. Como são explicações normalmente imprecisas e incompletas, essa abordagem é claramente insatisfatória. Em decorrência da falta de especificações semânticas completas de linguagens de programação, os programas são raramente provados corretos sem testes, e os compiladores comerciais nunca são gerados automaticamente a partir de descrições de linguagem (Sebesta, 2011).

7.5 Semântica operacional

A ideia da semântica operacional é descrever o significado de uma sentença ou programa pela especificação dos efeitos de rodá-lo em uma máquina. Os efeitos na máquina são vistos como sequências de mudanças em seu estado, onde o estado da máquina é a coleção de valores em seu armazenamento. Uma descrição de semântica operacional óbvia é dada pela execução de uma versão compilada do programa em um computador. A maioria dos programadores, em alguma ocasião, deve ter escrito um pequeno programa de teste para determinar o significado de alguma construção de linguagem de programação, especialmente enquanto estava aprendendo essa linguagem. Essencialmente, o que a pessoa está fazendo é usando semântica operacional para determinar o significado da construção.

Existem diversos problemas com o uso dessa abordagem para descrições formais completas de semântica. Primeiro, os passos individuais na execução da linguagem de máquina e as mudanças resultantes no estado da máquina são muito pequenos e numerosos. Segundo o armazenamento de um computador real é muito grande e complexo. Existem normalmente diversos níveis de dispositivos de memória, assim como conexões para incontáveis outros computadores e dispositivos de memória por meio de redes. Dessa forma, linguagens de máquina e computadores reais não são usados para semântica operacional formal. Em vez disso, linguagens de nível intermediário e interpretadores para computadores idealizados são projetados especificamente para o processo (Sebesta, 2011).

O processo básico da semântica operacional é usual. O conceito é bastante usado em livros-texto de programação e manuais de referência de linguagens de programação. Por exemplo, a semântica da construção for em C pode ser descrita em termos de sentenças mais simples, como em

Sentença C	Significado

for (i=0; i<100; i++) {
...
}
	i=0;
	loop: if i == 100 goto out
	...
	i++;
	goto loop
	out:...

7.6 Semântica denotacional

A semântica denotacional é o método mais rigoroso e mais conhecido para a descrição do significado de programas. Ela é solidamente baseada na teoria de funções recursivas.

O processo de construção de uma especificação de semântica denotacional para uma linguagem de programação requer que alguém defina, para cada entidade da linguagem, tanto um objeto matemático quanto uma função que mapeie as instâncias dessa entidade de linguagem para instâncias do objeto matemático. Como os objetos são definidos rigorosamente, eles modelam o significado exato de suas entidades correspondentes. A ideia é baseada no fato de que existem maneiras rigorosas de manipular objetos matemáticos, mas não construções de linguagens de programação.

A dificuldade ao usar esse método está na criação dos objetos e das funções de mapeamento. O método é chamado denotacional porque os objetos matemáticos denotam o significado de suas entidades sintáticas correspondentes.

Na semântica denotacional, o domínio é chamado de domínio sintático, porque são mapeadas estruturas sintáticas. A imagem é chamada de domínio semântico. Na semântica denotacional, as construções de linguagem de programação são mapeadas para objetos matemáticos – conjuntos ou funções. Entretanto, diferentemente da semântica operacional, a semântica denotacional

não modela o processamento computacional passo a passo dos programas (Sebesta, 2011).

7.7 Semântica axiomática

A semântica axiomática, chamada assim porque é baseada em lógica matemática. Em vez de especificar diretamente o significado de um programa, a semântica axiomática especifica o que pode ser provado sobre o programa.

Na semântica axiomática, não existe um modelo do estado de uma máquina ou programa, nem um modelo de mudanças de estado que ocorrem quando o programa é executado. O significado de um programa é baseado nos relacionamentos entre variáveis e constantes de um programa, os quais são o mesmo para cada execução do programa.

Quando a semântica axiomática é usada para especificar formalmente o significado de uma sentença, ele é definido pelo efeito da sentença em asserções sobre os dados afetados pela sentença. (Sebesta, 2011).

7.7.1 Asserções

As expressões lógicas usadas na semântica axiomática são chamadas de predicados, ou asserções. Uma asserção que precede imediatamente uma sentença de programa descreve as restrições nas variáveis do programa naquele ponto. Desenvolver uma descrição axiomática ou prova de um programa requer que toda sentença do programa tenha tanto uma pré-condição quanto uma pós-condição.

Como um exemplo simples, considere a seguinte sentença de atribuição e a pós-condição:

sum = 2 * x + 1 {sum > 1}

As asserções de pré e pós-condições são apresentadas em chaves para distingui-las das demais partes das sentenças dos programas. Uma possível pré-condição para essa sentença é {x > 10}.

Em semântica axiomática, o significado de uma sentença específica é definido por sua pré-condição e sua pós-condição. Na prática, as duas asserções especificam precisamente o efeito de executar uma sentença.

O conceito mais geral de semântica axiomática é expressar precisamente o significado de sentenças e programas em termos de

expressões lógicas. A verificação de programas é uma aplicação das descrições axiomáticas de linguagens (Sebesta, 2011).

7.7.2 Sentenças de atribuição

A pré-condição e a pós-condição de uma sentença de atribuição definem exatamente o seu significado. Para definir o significado de uma sentença de atribuição, deve existir uma maneira de calcular sua pré-condição a partir de sua pós-condição.

Por exemplo, se tivéssemos a sentença de atribuição e pós-condição

a = b / 2 - 1 {a < 10}

a pré-condição mais fraca seria computada pela substituição de b / 2 – 1 por a na pós-condição {a < 10}, como:

b / 2 - 1 < 10

b < 22

Logo, a pré-condição mais fraca para a sentença de atribuição dada e sua pós-condição é {b < 22}. Lembre-se que o axioma de atribuição é correto apenas na ausência de efeitos colaterais. Uma sentença de atribuição tem um efeito colateral se ela modifica alguma variável que não seja seu alvo (Sebesta, 2011).

7.7.3 Seleção

A seguir, consideramos a regra de inferência para sentenças de seleção, cuja forma geral é

if B then S1 else S2

Essa regra indica que as sentenças de seleção devem ser provadas tanto quando a expressão booleana de controle for verdadeira quanto quando for falsa.

A sentença de seleção de exemplo é

if x > 0 then

y = y - 1

else

y = y + 1

Suponha que a pós-condição, Q, para essa sentença de seleção seja

{y > 0}.

Podemos usar o axioma para atribuição na cláusula then

y = y - 1 {y > 0}.

Isso produz {y – 1 > 0} ou {y > 1}, que pode ser usado como a parte P da pré-condição para a cláusula then. Agora, aplicamos o mesmo axioma para a cláusula else

y = y + 1 {y > 0}

O que produz a pré-condição {y + 1 > 0} ou {y >-1}. Como {y > 1} => {y > -1}, a regra de consequência permite usarmos {y > 1} para a pré-condição de toda a sentença de seleção (Sebesta, 2011).

7.7.4 Laços lógicos com pré-teste

Outra construção essencial das linguagens de programação imperativa são os pré-testes lógicos, ou laços while. Computar a pré-condição mais fraca para um laço while é inerentemente mais difícil do que para uma sequência, porque o número de iterações não pode sempre ser pré-determinado. Em um caso onde o número de iterações é conhecido, o laço de repetição pode ser expandido e tratado como uma sequência.

O problema de computar a pré-condição mais fraca para laços de repetição é similar ao problema de provar um teorema acerca de todos os números inteiros positivos. No último caso, a indução normalmente é usada, e o mesmo método indutivo pode ser usado para alguns laços. O passo principal na indução é encontrar uma hipótese indutiva. O passo correspondente na semântica axiomática de um laço while é encontrar uma asserção chamada de invariante de laço, crucial para encontrar a pré-condição mais fraca.

Se um laço de repetição computa uma sequência de valores numéricos, é possível encontrar uma invariante de laço com uma abordagem usada para determinar a hipótese indutiva quando a indução matemática é utilizada para provar uma sentença a respeito de uma sequência matemática. O relacionamento entre o número de iterações e a pré-condição para o corpo do laço de repetição é computado para alguns casos, com a esperança de que um padrão apareça e possa ser aplicável para o caso geral. Ajuda tratar o processo de produzir uma pré-condição mais fraca como uma função, wp . Em geral

wp (instrução, pós-condição) = pré-condição.

Considere o laço de exemplo

while y <> x do y = y + 1 end {y = x}

Lembre-se de que o sinal de igualdade está sendo usado para dois propósitos aqui. Em asserções, ele representa igualdade matemática; fora das asserções, significa o operador de atribuição.

Para zero iterações, a pré-condição mais fraca é, obviamente, {y = x}

Para uma iteração, é

wp (y = y + 1, {y = x}) = {y + 1 = x}, ou {y = x - 1}

Para duas iterações, é

wp (y = y + 1, {y = x - 1}) = {y + 1 = x - 1}, ou {y = x - 2}

Para três iterações, é

wp (y = y + 1, {y = x - 2}) = {y + 1 = x - 2}, ou {y = x - 3}

Está agora óbvio que {y < x} será suficiente para os casos de uma ou mais iterações. Combinando isso com {y = x} para o caso de zero iterações, temos {y <= x}, que pode ser usada como a invariante do laço de repetição. Uma pré-condição para a sentença while pode ser determinada a partir da invariante do laço de repetição (Sebesta, 2011).

8 SÍNTESE

Com a parte de analise já realizada passamos para a parte de síntese de um compilador. Essa parte síntese ocorre três processos: geração de código intermediários, otimização de código e geração de código.

8.1 Geração do código intermediários

Segundo Aho et al. (2008) depois da análise do programa fonte, muitos compiladores geram uma representação intermediaria explicita de baixo nível ou do tipo linguagem de máquina, que podemos imaginar como um programa para uma máquina abstrata. Essa representação intermediaria deve ter duas propriedades importantes: ser facilmente produzida e ser facilmente traduzida para a máquina alvo.

No modelo de análise e síntese de um compilador, o front-end analisa um programa fonte e cria uma representação intermediaria, a partir da qual o back-end gera o código objeto. Com uma representação definida adequadamente, um compilador para a linguagem *i* e a máquina *j* pode ser então ser construído, combinando-se o front-end para a linguagem *i* com o back-end para a máquina *j.*

8.2 Otimização de código

Alguns compiladores possuem uma fase de otimização independente de máquina entre o front-end e o back-end. A finalidade dessa fase de otimização é realizar transformações na representação intermediária, de modo que o back-end possa produzir um programa objeto melhor do que teria produzido a partir de uma representação intermediaria não otimizada. De acordo com Aho et al. (2008) Normalmente, melhor significa mais rápido, mas outros objetivos podem ser desejados, como um código menor ou um código que consuma menos energia.

O número de otimizações de código realizado por diferentes compiladores varia muito. Aqueles que explora ao máximo as oportunidades são chamadas “compiladores otimizadores”. Quanto mais otimizado, mais tempo é gasto nessa fase. Mas existem otimizações simples que melhoram

significativamente o tempo de execução do programa objeto sem atrasar muito a compilação.

8.3 Geração de código

O gerador de código é a última fase em nosso modelo de compilador. Segundo Aho et al. (2008) nessa última fase ele recebe como entrada a representação intermediaria produzida pelo front-end do compilador e as informações da tabela de símbolos, e produz como saída um código objeto semanticamente equivalente à entrada, com isso, o código objeto precisa preservar o significado semântico do programa fonte ser de alta qualidade como usar efetivamente os recursos disponíveis da máquina destino e ser executado eficientemente.

Existe um grande desafio de gerar um código objeto ótimo para determinado programa fonte, alguns desses problemas seria a alocação de registradores. Felizmente temos técnicas heurísticas maduras para que um gerador de código projetado cuidadosamente possa produzir códigos bons, mas não perfeitos.

9 FERRAMENTAS EDUCACIONAIS

Alkimim e Mello (2010) enfatiza que linguagens de programação e compiladores são conteúdo de relativa dificuldade de compreensão, portanto, torna-se necessário o desenvolvimento de ferramentas educacionais voltadas à construção de compiladores. Contudo, poucas dessas ferramentas possuem capacidade suficiente para abarcar uma porção considerável das técnicas ensinadas em sala de aula, segundo Backes e Dahmer (2006).

Com a utilização de ferramentas educacionais tem auxiliado o processo de ensino e aprendizagens com uma melhor fixação dos conceitos e práticas apresentados durante o estudo de compiladores. Assim como, “é necessário, que o professor perceba a importância de oportunizar e reconhecer que a prática pedagógica mediada com as Tecnologias Digitais de Informação e Comunicação (TDICs) contribui muito em sala de aula trazendo novos processos de ensino e de aprendizagem” (Cinthyan Renata Sachs C. de Barbosa , Robson P. Bonidia , Joao Coelho Neto,2013,p.2).

Assim, o estudo da Construção de Compiladores é componente importante de um curso de Ciência da Computação. Sua prática é imprescindível, a familiarização com ferramentas de geração automática de analisadores (Lex/Flex, Jflex/Jcup), que permitem abordar as características reais de um compilador. Alkmim e Mello (2010) salientam que a ferramenta Flex (Fast lexical analyzer generator) e JFlex são programas que recebem como entrada um arquivo com um conjunto de expressões regulares, as quais podem associar um token específico a cada uma delas.

Desse modo, o presente trabalho tem o objetivo apresentar e analisar as ferramentas Flex, JFlex. Um enfoque a fase Análise Léxica será dado pois ela é a primeira etapa do processo da construção de um compilador. Ainda existem outras ferramentas como Lex, Yacc [Levine et al. 1992] e SCC [Foleiss et al. 2009] porém não serão abordadas no trabalho por serem pouco conhecidas e essas nem sempre estavam disponíveis as academias universitárias. Além disso, a fase de análise sintática e semântica também não será enfatizada, pois envolve uma gama maior de fatores como a gramática descrita, técnicas de análises sintáticas e teriam que ser abordadas outras ferramentas de apoio.

9.1 Flex

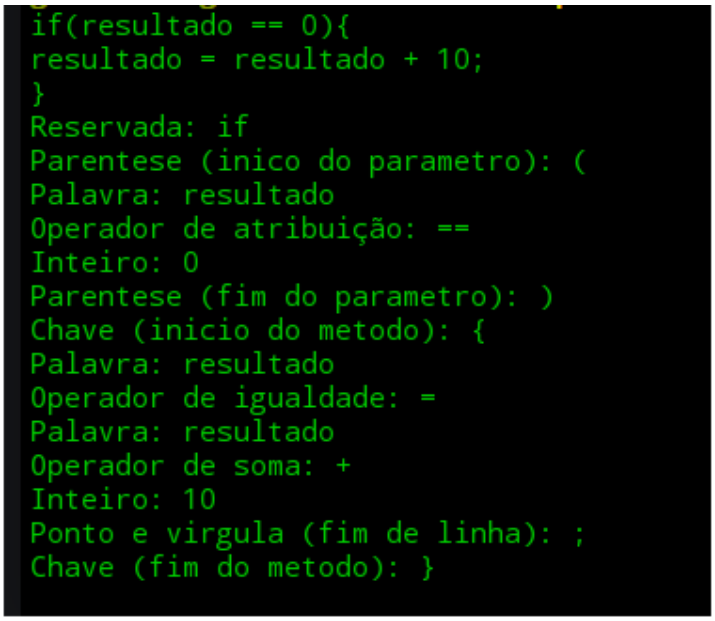
A ferramenta Flex é uma versão livre da antiga ferramenta Lex. Flex significa “Analisador Léxico Veloz” (do inglês Fast Lexical Analyzer). Esse analisador é utilizado para reconhecer padrões léxico em um texto. O Flex gera um código fonte usualmente na linguagem C. que é o analisador léxico [Levine 2009]. O analisador é gerado quando uma expressão é reconhecida, assim os comandos associados com ela são executados. Uma característica desta ferramenta é que não apresenta interface, ou seja, o usuário é deverá escrever o código em algum editor de texto. O código abaixo segue como exemplo:

```
DIGITO[0-9]
BINARIO[ZO-1]
COMENTARIO[^*/]*
LETRA[a-zA-Z]
WHITESPACE[ ]

%%
end                             {printf("Reservada: %s\n",yytext);}
if                              {printf("Reservada: %s\n",yytext);}
else                            {printf("Reservada: %s\n",yytext);}
while                           {printf("Reservada: %s\n",yytext);}
do                              {printf("Reservada: %s\n",yytext);}
for                             {printf("Reservada: %s\n",yytext);}
int                             {printf("Reservada: %s\n",yytext);}
float                           {printf("Reservada: %s\n",yytext);}
float                           {printf("Reservada: %s\n",yytext);}
char                            {printf("Reservada: %s\n",yytext);}
string                          {printf("Reservada: %s\n",yytext);}
include                         {printf("Reservada: %s\n",yytext);}
short                           {printf("Reservada: %s\n",yytext);}
break                           {printf("Reservada: %s\n",yytext);}
case                            {printf("Reservada: %s\n",yytext);}
switch                          {printf("Reservada: %s\n",yytext);}
const                           {printf("Reservada: %s\n",yytext);}
default                         {printf("Reservada: %s\n",yytext);}
double                          {printf("Reservada: %s\n",yytext);}
return                          {printf("Reservada: %s\n",yytext);}
long                            {printf("Reservada: %s\n",yytext);}
static                          {printf("Reservada: %s\n",yytext);}
"+"                             {printf("Operador de soma: %s\n",yytext);}
"-"                             {printf("Operador de subtração: %s\n",yytext);}
"*"                             {printf("Operador de multiplicação: %s\n",yytext);}
"/"                             {printf("Operador de divisão: %s\n",yytext);}
"=="                            {printf("Operador de atribuição: %s\n",yytext);}
"="                             {printf("Operador de igualdade: %s\n",yytext);}
"!"                             {printf("Negação: %s\n",yytext);}
"("                             {printf("Inico do parametro: %s\n",yytext);}
")"                             {printf("Fim do parametro: %s\n",yytext);}
"{"                             {printf("Inicio do metodo: %s\n",yytext);}
"}"                             {printf("Fim do metodo: %s\n",yytext);}
";"                             {printf("Ponto e virgula: %s\n",yytext);}
{LETRA}+                        {printf("Palavra: %s\n",yytext);}
{DIGITO}+                       {printf("Inteiro: %s\n",yytext);}
{DIGITO}+"."{DIGITO}*           {printf("Real: %s\n",yytext);}
{BINARIO}*                      {printf("Binario: %s\n",yytext);}
```

Para compilar esse programa é necessário ter o Flex instalado e um compilador C. Na primeira linha de comando foi executado: flex ”nome do arquivo.l”, isso irá criar um outro arquivo com o nome “lex.yy.c” que deverá ser compilado na linha de comando: gcc lex.yy.c -o ”nome da saída”. Por fim só precisa ser executado com o comando: ./”nome da saída”.

A imagens a seguir corresponde ao código da imagem logo acima:



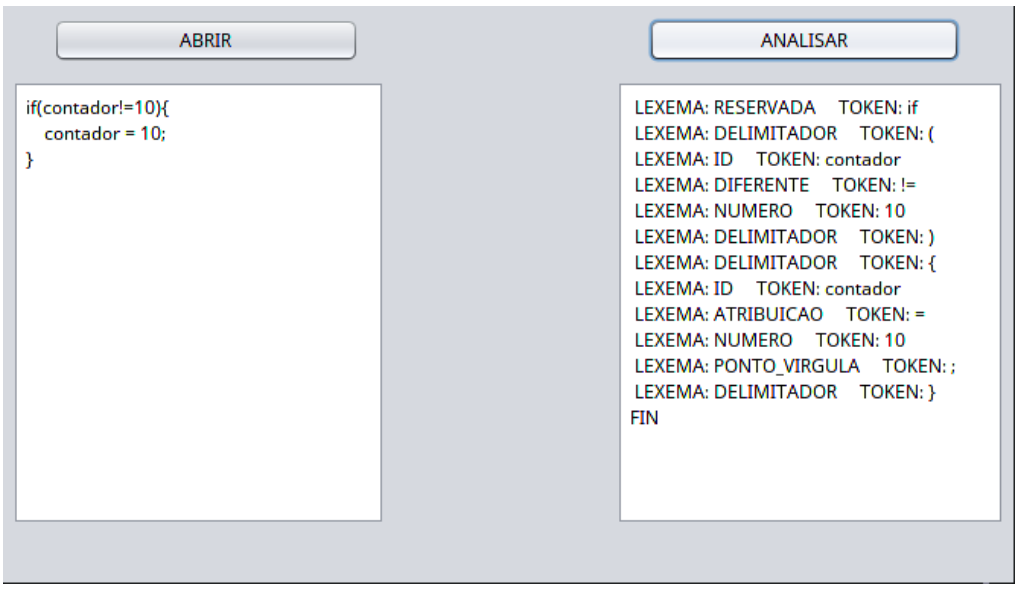
Essa ferramenta é bastante conhecida e possui muitos artigos explicativos de como utilizá-la e demostrando seus benéficos para a matéria de Compiladores. A única dificuldade encontrada que o usuário possa ter seria que ele necessita ter o conhecimento da linguagem C e de linhas de comando.

9.2 JFlex

JFlex é um gerador de análise lexical, pode ser conhecido como gerador de scanner para Java. Implementado também nessa linguagem, seria uma releitura da ferramenta Flex. O programa JFlex serve para criar uma classe Java que faz a análise léxica de qualquer arquivo texto. Esse faz o reconhecimento das linguagens por meio de tabelas de transição de autômatos determinísticos e de expressões regulares [Rojas e Mata 2005] [JFlex 2018]. O texto é dividido por seções %%. Tudo o que está escrito nessa área vai ser gerado pelo JFlex, como mostra a imagens abaixo. A classe gerada lê as entradas especificas por meio do conjunto de expressões regulares. Assim, é gerado o programa que irá quebrar os Tokens, que são as palavras chaves, comentários, operadores etc.

```
1  package tutorial;
2  import java.io.*;
3  import static tutorial.token.*;
4  %%
5  %class lexer
6  %type token
7  %line
8  %column
9  D=[0-9]
10 L=[a-zA-Z_]
11 CA="\""[^*]~"\"" | "\"" + "\""
12 WHITE=[ \t\r\n]
13 %{
14 public String lexeme;
15 InformacionCodigo c=new InformacionCodigo();
16 %}
17 %%
18 {WHITE} {/*Ignore*/}
19 <YYINITIAL> "+" {c.linea=yyline;lexeme=yytext(); return ADICAO;}
20 <YYINITIAL> "==" {c.linea=yyline;lexeme=yytext(); return IGUALDADE;}
21 <YYINITIAL> "-" {c.linea=yyline;lexeme=yytext(); return SUBTRACAO;}
22 <YYINITIAL> "!=" {c.linea=yyline;lexeme=yytext(); return DIFERENTE;}
23 <YYINITIAL> "||" {c.linea=yyline;lexeme=yytext(); return OR;}
24 <YYINITIAL> "&&" {c.linea=yyline;lexeme=yytext(); return AND;}
25 <YYINITIAL> "{" {c.linea=yyline;lexeme=yytext(); return DELIMITADOR;}
26 <YYINITIAL> "}" {c.linea=yyline;lexeme=yytext(); return DELIMITADOR;}
27 <YYINITIAL> "(" {c.linea=yyline;lexeme=yytext(); return DELIMITADOR;}
28 <YYINITIAL> ")" {c.linea=yyline;lexeme=yytext(); return DELIMITADOR;}
29 <YYINITIAL> "=" {c.linea=yyline;lexeme=yytext(); return ATRIBUICAO;}
30 <YYINITIAL> "/" {c.linea=yyline;lexeme=yytext(); return ENTRE;}
```

A imagem a seguir mostra o funcionamento do Analisador Léxico confeccionado pelo software JFlex:



No entanto, a aprendizagem pode se tornar um pouco mais complicada para quem não tem nenhum conhecimento da linguagem Java. Alguns dos objetivos do JFlex são: Geração de Scanner rápido e independência de plataforma.

10 CONCLUSÃO

Este trabalho visa analisar as principais etapas para a construção de um compilador. O objetivo geral foi apresentar aos alunos do curso de Ciência da Computação como seria a construção de um compilador, começando com o significado básicos e a diferença entre um compilador e interpretador, entendimento entre linhagens de baixo e altos nível. Partindo para as etapas de construção, identificando peculiaridades que possam ser empregadas no desenvolvimento dos compiladores, identificando dificuldades que possam aparecer no projeto e na implementação destes para uma determinada arquitetura e ideias básicas que podem ser utilizadas na construção. Com uma análise mais detalhada na parte de Análise Léxica, em virtude de ser a primeira etapa e que o aluno possa ter um entendimento do assunto mais fácil, já que a matéria de compiladores é uma que apresenta os índices mais elevados de evasão na área de Comutação.

E demostrando que a utilização de ferramentas educacionais ajuda os alunos a verem na pratica o que está sendo realizado e os professores, pois essas tentam auxiliar na melhoria da qualidade do ensino em ambientes de aprendizagem. Uma visão mais ampla de cada etapa da compilação se torna mais atrativa e didática, aliando a teoria à prática.

Além dos compiladores, os princípios e técnicas para o projeto de um compilador se aplica a vários outros domínios que provavelmente serão reutilizados muitas vezes na carreira de um cientista da computação.

11 BIBLIOGRAFIA

Aho, Alfred Vaino. et al. Compiladores: princípios, técnicas e ferramentas. Person Education do Brasil, 2008.

Vasconcelos, A. de Brit. Introdução a Arquitetura de Computadores. Disponível:<http://producao.virtual.ufpb.br/books/edusantana/introducao-a-arquitetura-de-computadores-livro/livro/livro.chunked/index.html>. Acesso em 20 de maio de 2020.

ESCOLA, Equipe Brasil. "Revolução do Computador"; *Brasil Escola*. Disponível:<https://brasilescola.uol.com.br/informatica/revolucao-do-computador.htm>. Acesso em 07 de julho de 2020.

Tucker. Allen B.; Noonan.Robert E. Linguagens de Programação: Princípios e Paradigmas. Mcgraw Hill, 2008.

Alkmim, G. P. e Mello, B. A. (2010). Ferramenta de apoio as fases iniciais do ensino ` de linguagens formais e compiladores. In XXI Simposio Brasileiro de Inform ´ atica na ´ Educac¸ao˜, Joao Pessoa, 1–4.

Backes, J. e Dahmer, A. (2006). C-gen–ferramenta de apoio ao estudo de compiladores. In XIV Workshop sobre Educac¸ao em Computac¸ ˜ ao˜, Porto Alegre, 1–8.

Cinthyan Renata Sachs C. de Barbosa, Robson P. Bonidia, João Coelho Neto. Flex, JFlex e GALS: Ferramentas de Apoio ao Ensino de Compiladores.

Levine, J., Mason, T. e Brown, D. (1992). Lex e Yacc. O'Reilly Media, Inc., 2 edition.

Foleiss, J. H., Assunc¸ao, G. P., Cruz, E., Gonc¸alves, R. e Feltrin, V. (2009). Scc: Um com- ˜ pilador c como ferramenta de ensino de compiladores. In Workshop sobre Educac¸ao˜ em Arquitetura de Computadores, Sao Paulo, 15–22.

Manning. Christopher D.; Schutze.Hinrich. Foundations of Statistical Natural Language Processing. Second printing, 1999.

Ricarte, I. L. M. (2003). Programação de Sistemas: uma introdução. *São Paulo*, *188*.

Levine, J. (2009). Flex & Bison: Text Processing Tools. O'Reilly Media, Inc.

Rojas, S. G. e Mata, M. A. M. (2005). Java a Tope: Traductores Y Compiladores Con Lex/yacc, Jflex/cup Y Javacc. Universidad de Malaga. 305p.

JFlex (2018). Jflex home page. Disponível em: http://jflex.de/. Acessado em 28/04/2020.

Costa, K. A. P., Silva, L. A. e Brito, T. P. (2008). Auxílio no ensino de compiladores: Software simulador como ferramenta de apoio na área de compiladores.

Sebesta, R. Conceitos de Linguagem de Programação. 9.ed, Bookman Company Ed, 2011.