Analisador Léxico para a Linguagem de Programação Swift Utilizando a Biblioteca JFlex

João Batista Romão Damasceno[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)
Maracanaú – CE -Brasil

joao.batista.romao06@aluno.ifce.edu.br

Abstract. *The compiler is a ferment present in the daily life of program developers, for most of them it is just a black box that receives a code in one language and returns it in another language that we want. However, if you look at the background a compiler is composed with a sequence of phases very well constructed so that this conversion is done correctly, both for the generation of code and to alert the developer about any problem that it may cause. In addition, the idea of a compiler is not something modern, but was conceived around 1950, along with the advent of computers.*

Resumo. *O compilador é uma ferramenta com diversas camadas que é muito usada pelos desenvolvedores para fazer a transformação de uma linguagem de alto nível para uma de baixo nível, a primeira camada chamada analise léxica tem como função classificar todo o texto escrito pelo desenvolvedor para que possa ser usado nas outras camadas. Para esse projeto foi desenvolvido um programa chamado Swift Lexical na linguagem Java com a biblioteca JFlex para fazer a análise léxica de para a linguagem de programação Swift, com intuito de aprendizado de como funciona essa camada assim com a importância dela para todo o fluxo de camadas de um compilador.*

1. Introdução

Existem diversas linguagens de programação no mundo, cada uma feita com um proposito, como por exemplo, para gerar programas de computadores ou para gerar uma aplicação em um dispositivo moveis. Essas linguagens de programação podem ser agrupadas em linguagens de alto nível como Java, Swift ou C++, que são linguagens cuja a escrita é mais próxima de um idioma falado, geralmente sendo esse o inglês, cujo proposito e facilitar a escrita de programas para esses desenvolvedores e a outra categoria são linguagens de baixo nível como o Assembler, que a sua escrita e feita para ser mais próximo do que o computador entende, utilizando os registradores e endereçamento de memória, pois elas não conseguem entender as linguagens de alto nível.

O Swift é um exemplo de linguagem de alto nível desenvolvida pela Apple para que possam desenvolver aplicações para as suas plataformas como iOS e macOS, sabendo disso fica a dúvida, como um programa escrito em uma linguagem de alto nível pode ser executado em um dispositivo que só entende linguagens de baixo nível?

Para que seja possível isso temos o compilador que é uma das ferramentas mais importantes que se tem no dia a dia de um desenvolvedor de software, a sua função é simples, a de converter um programa de uma linguagem de alto nível para uma linguagem de baixo nível, possibilitando assim que um programador que não sabe assembler, por exemplo, possa desenvolver uma aplicação para um dispositivo.

Entretanto para quem um compilador possa fazer essa conversão são necessários vários passos, e cada passo pode ser definido como uma camada. O intuito desse projeto é desenvolver a primeira camada definida como a de analise léxica, que lê um fluxo de entrada composto por caracteres tipicamente a partir de um arquivo no computador e o agrupa em sequencias significativas chamadas lexemas para a linguagem de programação Swift, utilizando um programa desenvolvido em Java com auxílio da biblioteca JFlex, que é um gerador de analisador léxico, e por fim gerar os lexemas com base em algumas palavras e estruturas reservadas por essa linguagem escolhida que possa ser usada na próxima camada do compilador que é o de analise sintática

2. Compilador

Foi definido que um compilador é um tipo de caixa preta, que pega um programa de escrito em uma linguagem de alto nível e o transforma em um programa de baixo nível com a mesma equivalência, entretanto para Cooper (2014) um compilador precisa respeitar sempre respeitar dois princípios. O primeiro princípio é que ele deve preservar o significado do programa a ser compilado, ou seja, o programa de saída tem que realizar as mesmas funções do programa de entrada. E o segundo é que o compilador deve melhorar o programa de entrada de alguma forma, e que seja perceptível, se ele não seguir esses princípios, ele não é um compilador.

Entretanto, no interior dessa caixa preta existem diversos processos ocorrendo, como ilustra a imagem 2, que podem ser divididos em dois grupos o de análise e o de síntese, comumente chamado de front-end e back-end respectivamente.

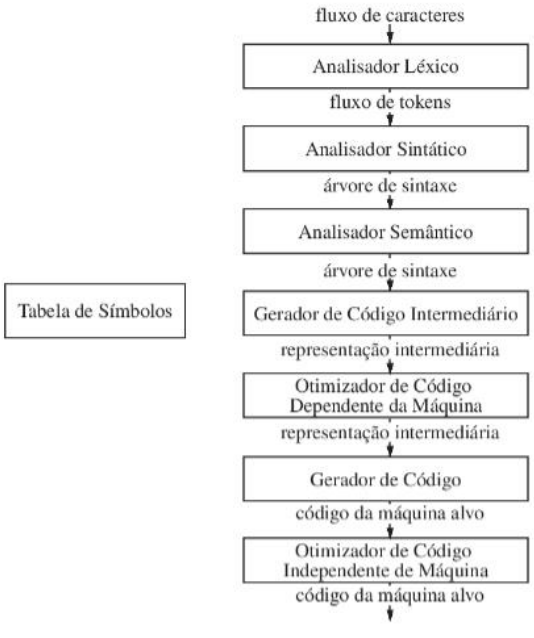


Figura 1. Fases de um compilador [Aho 2008]

3. Analise Léxica

Esse trabalho terá o foco na primeira camada que é de nosso compilador que é denominada de analisador léxico ou scanner, o compilador ê um fluxo de entrada composto por caracteres tipicamente a partir de um arquivo no computador e o agrupa em

sequencias significativas chamadas lexemas. Essas sequencias de caracteres forma o elemento básico do nosso compilador que são os tokens, que representa uma sequência de caracteres logicamente coesiva, ou seja, ele pertence a linguagem de entrada.

Alguns termos devem ser explicados para que possa ter um maior entendimento sobre a camada de analise léxica, dentre é a distinção entre token, padrão e o lexema. Os tokens são grupos de valores no formato nome do token e valor do atributo que segundo Aho(1995) o nome do token é um símbolo abstrato que será usado posteriormente na análise sintática, e já o valor-atributo é um ponteiro que aponta par uma entrada na tabela de símbolos referente ao token. Essa tabela é utilizada durante todo o processo do compilador.

Já o padrão é uma descrição da forma como os lexemas de um token podem assumir. Por exemplo no caso do token, o padrão dele é apenas uma sequencia de caracteres que formam uma palavra-chave, já para os identificadores o padrão podem ser muitas sequências de caracteres. E por fim o lexema, que é uma sequência de caracteres no programa fonte que casa com o padrão para um token, e esses lexemas são identificados pelo analisador léxico como uma instancia desse token.

O analisador léxico interage bastante com a camada de analise sintática, pois é ela que recebe os tokens que são gerados. Essa interação se dá da geralmente quando o analisador sintático chama o analisador léxico, que lê os caracteres de sua entrada até que ele possa identificá-lo e produzir um novo token que é retornado o analisador sintático como mostrado na figura 2.

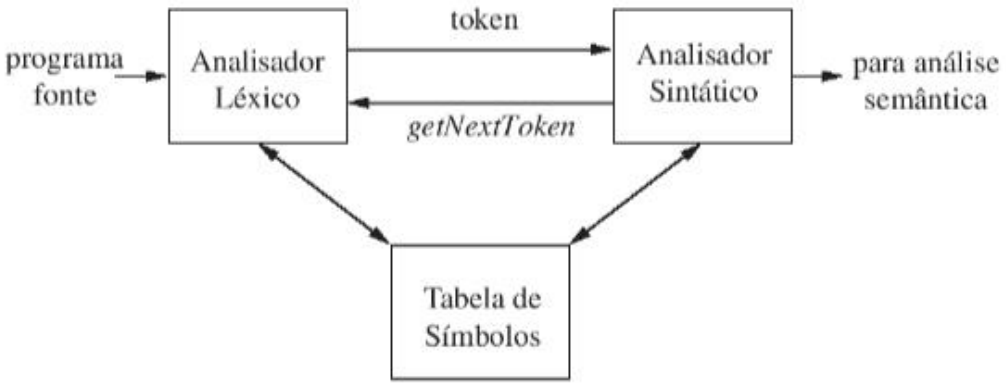


Figura 2. Comunicação entre a camada léxica e sintática [Aho 2008]

Além da criação de tokens, essa camada também pode realizar tarefa de remover comentários, espaços em branco, quebras de linha dentre outros tipos, que são dados que não importam para a lógica do programa em si, e por fim correlacionar as mensagens de erro geradas pelo compilador com o programa fonte.

Segundo Aho (2008), o analisador léxico pode ser dividido em dois processos sendo o primeiro o escandimento que consiste no processo simples de varredura de entrada a procura de comentários, caracteres em branco e outros caracteres que devem ser ignorados e os remove-los do arquivo. E a segunda parte é a analise léxica em si, que é quando o analisador produz os tokens de saída.

A maioria das linguagens de programação ocidentais, possuem microssintaxe, como exemplo as letras são agrupadas da esquerda para a direita e o espaçamento ou uma pontuação define o fim de uma palavra. Quando o analisador léxico pega um grupo de palavras em potencial ele verifica no dicionário a sua validade e cria o seu token. Entretanto existem palavras especiais chamadas palavra-chave ou palavras reservadas, que correspondem a uma determinada regra, como por exemplo a palavra while, que é usada em diversas linguagens, se ao programar nessa linguagem o desenvolvedor escrever uma variável com esse nome, o compilador irá dá um alerta de erro, pois não poderia criar um token separado para ela.

3.1. Classificação de Tokens

Obter do arquivo de entrada as strings que compõem o programa que será compilado é apenas uma das partes do processo dessa camada como mencionado acima, outra de suas responsabilidades é a de classificação de tokens. Classificar um token, segundo Ricarte (2008), não é uma tarefa tão simples quando ler uma string do arquivo. Uma possibilidade que ajudaria com essa tarefa seria criar uma tabela no analisador léxico que conseguisse mapear as strings lidas aos seus respectivos tipos, entretanto essa estratégia só é viável para as palavras reservadas de uma linguagem como if, else, func, pois os identificadores podem ter incontáveis combinações.

Para que se possa reconhecer esses tipos de símbolos, no caso as palavras reservadas, é necessário que tenha alguma estrutura que permite realizar esse reconhecimento, e neste caso é um autômato finito.

3.2. Autônomo Finito

Um autômato finito é uma máquina de estados finitos que permite reconhecer, por meio de um conjunto de estados e transições dirigidas pela ocorrência de símbolos de um alfabeto, se uma determinada string pertence ou não a uma linguagem regular [Ricarte 2008]. Nele existe um estado inicial, que determina o início da análise da sentença a ser reconhecida. A medica que os caracteres são lidos pelo autônomo, o controle da máquina vai passando pelos estados até que chegue em um estado de aceitação, quando ele é reconhecido pelo autônomo, ou no estado aonde ele recusa a string. Por exemplo, a string if é reconhecida como um token da linguagem para palavras reservadas que deve estar na tabela.

Desta maneira, se nossos autônomos finitos forem definidos para diferentes classes de tokens, que são usados pela linguagem, esse pode ser usado como classificadores para a camada de analise léxica

4. Materiais e métodos

4.1. Linguagem de alto nível Swift

Como dito anteriormente, o compilador converte uma linguagem de alto nível em uma linguagem de baixo nível, e um analisador léxico pega os dados do programa em alto nível e cria os token dentre outras coisas, o analisador léxico desenvolvido por esse projeto tem como linguagem de alto nível a linguagem de programação Swift.

O Swift, segundo Apple (2021) é uma linguagem de programação consistente e intuitiva, desenvolvida pela Apple para a criação de apps para iOS, Mac, Apple TV e

Apple Watch. Ela foi criada para dar ainda mais liberdade para os desenvolvedores. Swift é fácil de usar e em código aberto, para que qualquer pessoa com uma boa ideia consiga fazer coisas surpreendentes.

O Swift é derivado da linguagem de programação Objective-C, então algumas de suas estruturas são semelhantes entre si, caso futuramente ocorra uma adaptação para essa linguagem algumas palavras chaves poderiam ser reutilizadas.

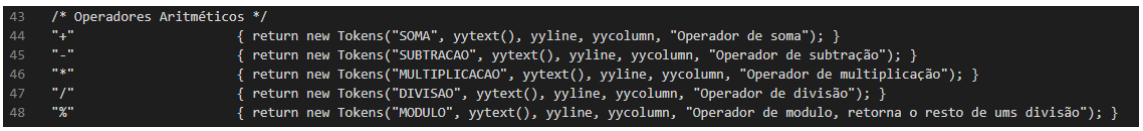
4.2. Linguagem Java

O analisador léxico foi feito na linguagem Java, que é uma linguagem de programação com suporte da empresa Oracle (2021) que a define como uma linguagem de programação e plataforma de desenvolvimento nº 1. Com milhões de desenvolvedores executando mais de 51 bilhões de Java Virtual Machines(JVM) em todo o mundo, sendo a plataforma de desenvolvimento preferida por empresas e desenvolvedores em todo o mundo.

Portanto, ao gerar o programa ele pode ser executado em qualquer plataforma que tenha uma JVM, no caso deste projeto foi testado em uma máquina com o sistema operacional Windows 10 com o JVM no build 1.8.

4.3. Biblioteca JFlex

A biblioteca JFlex é um gerador de analisador léxico para a linguagem de programação. Os lexers são baseados em autônomos finitos determinísticos pois são rápidos e sem retrocesso de caros [Jflex.2021]. Ele só precisa de um arquivo na extensão flex com as regras a qual ele deve trabalhar, como por exemplo ignorar os comentários, e os tokens reconhecidos, como por exemplo a estrutura condicional if. Na figura abaixo podemos ver um exemplo de tokens que devem ser reconhecidos pelo nosso analisador léxico para a linguagem swift para os operadores aritméticos.



```
43  /* Operadores Aritméticos */
44  "+"                    { return new Tokens("SOMA", yytext(), yyline, yycolumn, "Operador de soma"); }
45  "-"                    { return new Tokens("SUBTRACAO", yytext(), yyline, yycolumn, "Operador de subtração"); }
46  "*"                    { return new Tokens("MULTIPLICACAO", yytext(), yyline, yycolumn, "Operador de multiplicação"); }
47  "/"                    { return new Tokens("DIVISAO", yytext(), yyline, yycolumn, "Operador de divisão"); }
48  "%"                    { return new Tokens("MODULO", yytext(), yyline, yycolumn, "Operador de modulo, retorna o resto de ums divisão"); }
```

Figura 3. Exemplos de criação de tokens

4.4. IDE's

Uma IDE (Ambiente de desenvolvimento integrado) é um software que combina ferramentas comuns de desenvolvimento, como por exemplo o autocomplete, em uma única interface gráfica do usuário (GUI), facilitando o desenvolvimento de aplicações no geral. Para desenvolver o analisador em Java junto com a sua interface foi utilizado a IDE netbeans na sua versão 8.2. E para desenvolver os tokens no arquivo flex, foi utilizado o Visual Studio Code na versão 1.53 pois podia-se baixar um plugin que marcava as palavras chaves do código deixando-o assim mais fácil desenvolver as regras, ambos os softwares são gratuitos e tem disponibilidade para a plataforma Windows, Linux e MacOS.

5. Swift Lexical

O swift lexical é um programa criado na linguagem java que analisa um código na linguagem swift e mostra o resultado da análise léxica em uma tabela com os seguintes nomes: Identificador, que informa ao usuário qual o nome do token utilizado, podendo

ser String, variável, dentre outras, o valor que é a string lida pelo analisador, a linha a qual essa string pertence e por sim uma descrição, que informa ao usuário aonde é usado aquela string e com exemplo como no caso de somar que explica que operador “+” realizar a soma entre dois valores da seguinte forma a + b.

Para gerar o analisador primeiro foi inserido no projeto a biblioteca JFlex, que pega os dados de um arquivo .flex e gera uma classe com autômato determinístico que classifica as strings de entrada com base nos do arquivo de origem.

5.1. Lexer.flex

O arquivo lexer.flex juntamente com a biblioteca JFlex são o coração da aplicação, nesse arquivos foram criadas todas as regras e identificadores que podem ser classificados na aplicação.

Primeiramente definimos o nome da classe gerada pelo JFlex, que neste caso é o Analisador, para defini-la é somente colocar o nome class Nome da classe após o caractere %, que será identificado pela biblioteca assim como uma classe que vai receber o resultado da análise chamada Tokens. Ao ser identificado uma palavra reservada, ele inicia um objeto dessa classe e passa o nome do token, o texto analisado, a linha que ele se encontra, a coluna e por sim uma breve descrição, esses são os dados usados posteriormente para mostrar ao usuário o resultado da análise.

Para que seja identificado palavras utilizadas no código, que não são estáticas, ou seja, não são as palavras reservadas, foram criadas algumas regras de expressões regulares comumente chamadas de regex. Que nada mais são que padrões utilizados para identificar determinadas ocorrências em uma cadeia de caracteres.

Por exemplo, se for necessário identificar uma cadeira de caracteres alfa numéricos para representar uma variável, que obrigatoriamente tem que iniciar com um caractere não numérico, pode-se utilizar a seguinte regra [a-zA-Z0-9]. Essa expressão significa que aceita qualquer texto, maiúsculo ou minúsculo, que pode ou não ter um numeral no final. Essa regra no arquivo foi atribuída a um identificador cujo nome e alfanumérico. E ao combinar esses identificadores, obtém-se as logicas para pegar alguns valores mais utilizados na linguagem swift. Pode-se observar abaixo na figura 4 alguns identificadores genéricos utilizados, como para detectar números inteiros, decimais, arrays ou strings em um formato especifico. Pode-se notar alguns identificadores que devem ser ignorados pelo nosso analisador léxico, como tabulações e quebras de linha.

```
17  LineTerminator = \r|\n|\r\n
18  InputCharacter = [^\r\n]
19  WhiteSpace     = {LineTerminator} | [ \t\f]
20  Alfa = [a-zA-Z]
21  AlfaNumerico = [a-zA-Z0-9]
22  Caractere = '[a-zA-Z]'
23  Identificador = ({Alfa}|_)({AlfaNumerico}|_)*
24  IdentificadorMetodo = ({Alfa}|_)({AlfaNumerico}|_)*("!"|"?")?
25  Digito = [0-9]
26  Inteiro = {Digito}+("_"{Digito}+)*
27  Decimal = {Digito}+"."{Digito}+|"."{Digito}+|{Digito}+"."
28  DecOrInt = {Decimal}|{Inteiro}
29  Array = \[(([a-zA-Z0-9]+\,[a-zA-Z0-9]+))*\]|\[{Identificador}*\]
30  Matrix = \[?({Array}+\,{Array}+)\]|\[{Array}*\]
```

Figura 4. Identificadores genéricos da linguagem

Após serem criadas as regras mais gerais do código, foram criadas as regras dos operadores, já que esses operadores são palavras reservadas da linguagem, pode-se passar o seu valor real como o símbolo “+”, como visto na figura 3. Os operadores identificados pela programação os de aritmética, comparação lógicos, bit a bit, atribuição, intervalo e os operadores usuais, como parênteses e colchetes.

Por fim, também foram criados os identificadores para algumas palavras reservadas mais usadas na linguagem, já que essa aplicação é mais voltada para o ensino que a criação de um analisador complexo, foram utilizadas somente 45 palavras reservadas dentre elas controle de fluxo como if, e de loops como o for, e no final foi adicionado as regras para ignoras os comentários e os espaços em branco, parta isso, após ser informado o identificado deve-se apenas inserir a seguinte linha de código { /* ignore */ }, e os identificadores genéricos que foi criado no início do arquivo.

5.2. Classes Java

A classe com nome principal tem uma única responsabilidade, a de passar para a biblioteca JFlex o arquivo .flex e gerar a classe que irá analisar o código. Para isso e chamada a função generateLexer que passa o caminho para o arquivo, e no projeto e gerada a classe automaticamente denominada Analisador. Todas vezes que uma nova regra é adicionada no Lexer, essa classe tem que ser executada, para que o analisador possa criar um novo autônomo para identificar as novas regras.

Já na classe FramePrincipal, é onde o usuário vai interagir com a aplicação, pode-se observar a interface da aplicação na figura 5. O usuário pode realizar apenas quatro ações, mas são mais que suficiente para se obter a análise. Primeiro ele pode analisar um código que ele colocou na interface, segundo e ele abrir um arquivo do com extensão swift, a terceira é apagar a análise que foi feita, e por fim exportar o resultado na análise no formato txt com o nome definido por ele.

Dentro dessa classe, é que todas as classes mencionadas se encontrarem, primeiramente e criada uma instancia da nossa classe analisador, e é passado um arquivo txt, criado pelo próprio programa, para armazenar o código passado pelo usuário, o analisador lê esse código, e a nova string lida ele retorna um objeto do tipo Token, com os dados sobre aquela string, e ficam em loop até que toda a classe seja analisada, e esses resultados são armazenados em duas variáveis, uma que salva o objeto do tipo token que será apresentado ao usuário em forma de tabela com nome dataSource e outra que será utilizado para salvar a analise quando o usuário clicar na opção de exportação.

5.3. Tratamento de erros

O analisador léxico feito com a biblioteca mencionada anteriormente, pode retornar alguns erros, como receber um caractere que não é reconhecido. Para evitar isso houve três tratamentos de erros.

O primeiro para tratar esse erro de receber caractere, como o do idioma japonês, ele retorna o identificador como erro, informa o valor, no caso o caractere não identificado, e pôr fim a mensagem de erro em forma de descrição. Esse erro é o do tipo de tempo de execução, que só ocorre quando a aplicação faz uma analises.

O segundo tratamento é para erros genéricos, no caso do tipo IOException, que são todos os erros possíveis, que retorna ao usuário o identificar erro, o valor como

desconhecido, por que são erros a qual não foi identificado no projeto, mas que possa vir a ocorrer de alguma forma desconhecida, com a linha como zero e pôr fim a mensagem de erro detectado pelo programa.

E por fim, o tratamento de erro para quando o usuário digitar o nome de arquivo que deve ser analisado, que podem ser dois, o primeiro e quando o arquivo não existe, isso faz com que na área de texto apareça uma mensagem dizendo que o arquivo não foi encontrado já o segundo e quando ocorre algum erro interno de java, que retorna uma mensagem genérica.

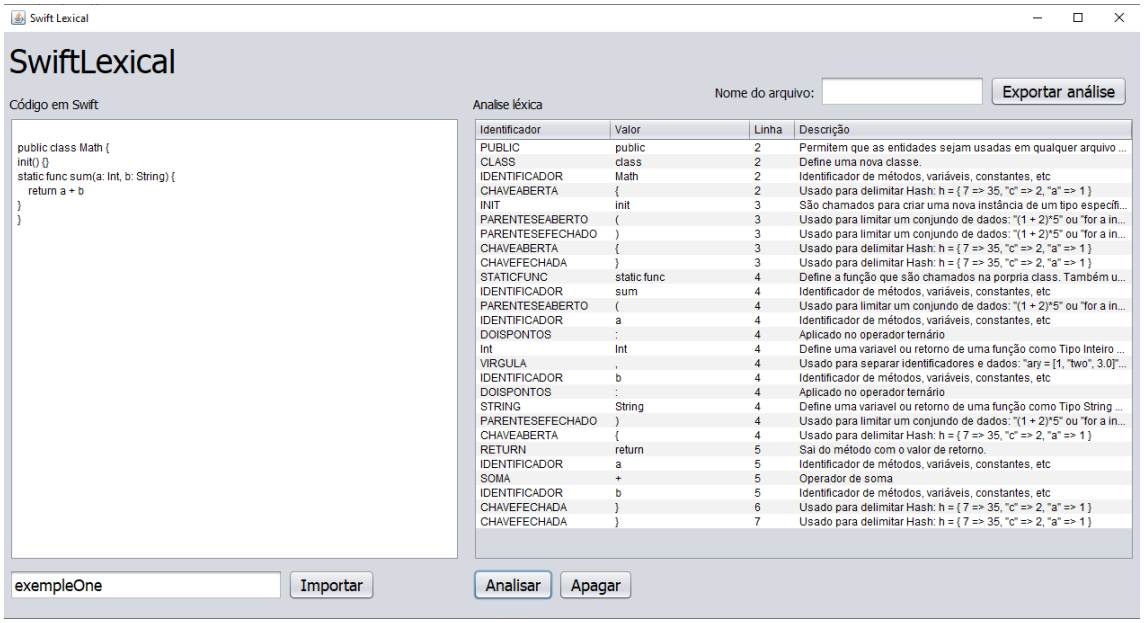


Figura 5. Interface gráfica do SwiftLexical

6. Considerações finais

Como pode ser visto no decorrer desse projeto, a criação de um analisador léxico, utilizando a biblioteca JFlex é algo bem simples, a parte mais complicada feita para o programa em si é a criação das regras que devem ser identificadas pelo analisador. Entretanto é muito útil pois é possível observar como ocorrer uma das peças que existem no compilador, que geralmente para os desenvolvedores é apenas uma caixa preta, e ter um entendimento melhor de como funciona essa camada na forma prática

Também pode-se notar que é essa primeira camada é uma das mais importantes, pelo fato dela ser a entrada do código em si e fazer a classificação desses caracteres, pois se o analisador léxico ignorar algum regra importante, ou não ter alguma regra essencial para aa linguagem o erro pode ir se propagando pois seus valores serem utilizados no decorrer da camada front-end na tabela de símbolos, até que encontre uma camada que o ira identificar e relatar ao usuário.

Então uma boa estruturação dessa camada, para que possa aceitar todos os tokens da linguagem é essencial para que o compilador, por exemplo, da linguagem swift possa realizar o seu trabalho de fato.

Além disso, o link para o manual de instalação, manual de uso e vídeo demonstração desse projeto e o projeto no Github podem ser encontrados em Damasceno (2021).

7. Referências

Aho, Alfred V., et. al (1995) “Compiladores: Princípios, técnicas e ferramentas. 2° ed.

Aho, Alfred V., et. al (2008) “Compiladores: Princípios, técnicas e ferramentas. 1° ed.

Apple (2021). “Oracle Java”. Disponível em: https://developer.apple.com/swift/#:~:text=Swift%20is%20a%20powerful%20and,software%20that%20runs%20lightning%2Dfast.. Acesso: 21 fev. 2021.

Cooper, Keith D. e Torczon, Linda (2014) “Construindo Compiladores”. 2° ed.

Damasceno, João Batista Romão (2021). “Manuais e video”. Disponível em: https://drive.google.com/drive/folders/12flqMfR9xniq1BjVLVRRWOAGz9BMnNf9?usp=sharing. Acesso: 23 fev. 2021.

Damasceno, João Batista Romão (2021). “Projeto SwiftLexical”. Disponível em: https://github.com/JBatista1/Swift-Lexical. Acesso: 23 fev. 2021.

JFlex(2021). “JFlex”. Disponível em: https://jflex.de/. Acesso: 21 fev. 2021.

Oracle (2021). “Swift”. Disponível em: https://www.oracle.com/br/java/. Acesso: 21 fev. 2021.

Ricarte, Ivan (2008) “Introdução à compilação”.