UNIOESTE – Universidade Estadual do Oeste do Paraná

Curso de Ciência da Computação

**Projeto 1.2**

**Compiladores**

**LARISSA LEWARTOSKI WONG**

Foz do Iguaçu – PR

2022

**Nome do Software:**

* Analisador léxico e sintático para subconjunto de C: Lex

**Nome dos integrantes da equipe:**

* O projeto foi realizado de forma individual.

Larissa Lewartoski Wong.

**Descrição da linguagem:**

A linguagem X é formada por 26 classes de tokens, sendo elas:

* Um token de encerramento do arquivo;
* Um tipo de token de Operador para os operadores:
  + \*
  + /
  + +
  + –
  + <
  + <=
  + >
  + >=
  + %
  + ==
  + !
  + !=
  + &&
  + ||
* Tokens para os símbolos:
* (
* )
* {
* }
* ;
* ,
* :
* #
* .
* Lambda
* Tokens para palavras-chave, reservadas ou tipos de dados:
* if
* Else
* While
* Default
* Switch
* Case
* Break
* Tipodado (engloba os tipos int, float, char e double)
* Token de identificador
* Token de número
* Token para string

Os nomes das classes de tokens são:

Finaldaentrada, Palavrachave\_if, Palavrachave\_include, Operador, Palavrachave\_switch, Palavrachave\_case, Palavrachave\_else, Palavrachave\_while, Abparentese, Fcparentese, Abchave, Fcchave, Pontoevirgula, Virgula, Identificador, Inteiro, String, Tipodado, Palavrachave\_for, Palavrachave\_return, Lambda, Cerquilha, Ponto, Doispontos, Palavrachave\_break, Palavrachave\_default

As classes que começavam com Op todas foram inseridas em uma classe só chamada Operador.

As classes que começam com Palavrachave ou Tipodado são de palavras-chave, palavras reservadas ou tipos de dado da linguagem, foram incluídas as palavras while, if, else, for, return, include, break, switch, case e os tipos de dado int, float, double e char.

A classe do token de finalização do arquivo é utilizada quando ocorre a entrada de um caractere inválido ou o arquivo lido acaba.

São classes de símbolos as classes com nomes dos próprios símbolos como a classe Virgula. No caso de classes com símbolos em que há abertura e fechamento, como por exemplo o caso de parênteses, foi adotada a forma Ab para sinalizar token de abertura e Fc para sinalizar token de fechamento.

O token de identificador aceita identificadores que se iniciem com letras (a até z), números (0 a 9) ou o caractere \_. Seguidos de mais caracteres seguindo a mesma regra.

O token de número aceita números de 0 a 9.

O token de string aceita linhas com o formato de string da linguagem.

**Descrição das expressões regulares ou autômatos utilizados:**

Para a maior parte dos tokens não foi necessário gerar expressões regulares complexas. No caso de tokens como os operadores, os símbolos e as palavras especiais as expressões são idênticas às palavras.

Para tokens como identificador, número e string foi necessário gerar expressão regular.

Assim temos:

Finaldaentrada = ( \u0000 )

Operador = ( = | \* | / | % | + | - | ! | < | <= | > | >= | != | && | || )

Palavrachave\_if = ( if )

Palavrachave\_else = ( else )

Palavrachave\_while = ( while )

Abparentese = ( ( )

Fcparentese = ( ) )

Abchave = ( { )

Fcchave = ( } )

Pontoevirgula = ( ; )

Virgula = ( , )

Ponto = ( . )

Doispontos = ( : )

Cerquilha = (#)

Palavrachave\_switch = (switch)

Palavrachave\_case = (case)

Palavrachave\_break = (break)

Identificador = [a – z | A – Z | \_ ] ( a – z | A – Z | \_ )\*

Inteiro = [0 – 9]+

String = “[ ^” \n]\*”

Tipodado = ( int | float | double | char )

Palavrachave\_for = ( for )

Palavrachave\_return = ( return )

Palavrachave\_include. = ( include )

**Partes do código em que são definidos os tokens:**

Tela de computador

Descrição gerada automaticamente

Tela de computador

Descrição gerada automaticamente

Tela de computador

Descrição gerada automaticamente

Tela de computador

Descrição gerada automaticamente

Tela de computador

Descrição gerada automaticamente

**Descrição da gramática:**

A gramática desenvolvida foi:

<BLOCO> ::= IF | ELSE | INCLUDE | RETURN | DECLARACAO | FOR | WHILE | IDENTIFICACAO | SWITCH

<IF> ::= Palavrachave\_if ( IDENTIFICACAO ) { BLOCO }

<ELSE> ::= Palavrachave\_else { BLOCO }

<INCLUDE> ::= # Palavrachave\_include < Identificador . h >

<RETURN> ::= Palavrachave\_return IDENTIFICACAO ;

<DECLARACAO> ::= Tipodado Identificador ;

<FOR> ::= Palavrachave\_for ( IDENTIFICACAO ; IDENTIFICACAO ; IDENTIFICACAO ){ BLOCO }

<WHILE> ::= Palavrachave\_while ( IDENTIFICACAO ) { BLOCO }

<IDENTIFICACAO> ::= TERMOID B | TERMOINT A;

<SWITCH> ::= Palavrachave\_switch ( Identificador ){ CASE }

<CASE> ::= Palavrachave\_case IDENTIFICACAO : BLOCO Palavrachave\_break ; CASE

<CASE> ::= Palavrachave\_default IDENTIFICACAO DEFAULT

<DEFAULT> ::= : BLOCO

<A> ::= Operador TERMOID

<A> ::= Operador TERMOINT

<A> ::= "Lambda/vazio"

<TERMOID> ::= Identificador

<TERMOINT> ::= Inteiro

<B> ::= EXPRESSAO

<B> ::= ATRIBUICAO

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

A definição da gramática no código foi dividida na realização das regras como:

Tela de computador

Descrição gerada automaticamente

Que se repetiu para todas as regras:

Tela de computador

Descrição gerada automaticamente

Tela de computador

Descrição gerada automaticamente

E após a definição foram desenvolvidas as funções que seriam chamadas para resolução das regras como:

Desenvolvimento da regra CASE

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Desenvolvimento da regra DEFAULT

Tela de computador

Descrição gerada automaticamente

Desenvolvimento das regras SWITCH, ATRIBUICAO e B

Tela de computador

Descrição gerada automaticamente

Desenvolvimento das regras IDENTIFICACAO e FOR

Tela de computador

Descrição gerada automaticamente

Desenvolvimento das regras WHILE e ELSE

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Desenvolvimento da regra IF

Tela de computador

Descrição gerada automaticamente

Desenvolvimento da regra BLOCO

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Desenvolvimento das regras RETORNO, TERMOINT e TERMOID

Tela de computador

Descrição gerada automaticamente

Desenvolvimento das regras A, EXPRESSAO e INCLUDE

Tela de computador

Descrição gerada automaticamente

Desenvolvimento da regra DECLARACAO

Tela de computador

Descrição gerada automaticamente

**Descrição do funcionamento do software:**

O software realiza as análises léxica e sintática a partir de um arquivo texto. Após a leitura do arquivo as informações são processadas e o software imprime no terminal eventuais mensagens de erro. Se tudo correr sem erros o arquivo é lido completamente, se for encontrado algum erro esse erro é impresso no terminal juntamente com o local do código em que ele se encontra.

Em relação à análise sintática os erros são impressos com a linha na qual o erro ocorreu, na análise léxica é mostrada a linha e a coluna do erro.

**Instruções para a execução:**

Para rodar o código basta ter o compilador para a linguagem Java, pois o trabalho foi desenvolvido nessa linguagem. É necessário compilar e todos os arquivos fonte devem estar no mesmo diretório. Se for de interesse compilar utilizando comando, para realizar esse processo é necessário acessar o local do arquivo lex.java e executar o seguinte comando:

**javac Main.java**

Esse comando realiza a compilação e gera arquivos .class, para que seja utilizado o caminho correto do arquivo no comando sugere-se que digite javac lex e utilize a tecla tab para completar o caminho do arquivo. Após a compilação é possível executar o programa com o comando:

**java Main**

Com a execução o programa vai imprimir uma mensagem pedindo que seja digitado o nome ou caminho do arquivo em questão. Não é necessário digitar o final (.txt), apenas o nome ou caminho do arquivo desejado. Exemplos de inserção de arquivo:

**C:\Users\nomeUsuario\Desktop\teste1**

**teste1**

Com a execução do programa será gerado um arquivo texto com o processamento da cadeia de entrada no seguinte formato:

Tela de computador

Descrição gerada automaticamente

Em que a cada chamada de não terminal aparecerá qual foi a regra utilizada e a tabela de entrada desta regra com os símbolos adquiridos na análise léxica.

**Tratamento de erros:**

A estratégia de tratamento de erros é: identificar o erro, mostrar na tela a posição do erro (linha e coluna). Os erros identificados pelo programa são:

* Identificação de caractere vazio
* Sequência de escape não reconhecida (\r)
* Caractere não reconhecido
* Constante multi caracteres
* Final de arquivo enquanto lia string (fechamento de string faltando ou fora do final da linha)
* Número inválido (começando com número porém continuando com outros caracteres como 785abaco)

Para tratamento de erros sintáticos é realizada a checagem na gramática inserida no programa e cada regra possui suas próprias mensagens de erro como por exemplo:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Se a análise sintática ocorrer sem erros é impressa uma mensagem na tela constatando o fato.

Para tratamento dos erros que ocorrem no programa foi utilizada a abordagem do modo pânico, em que alguns tokens foram escolhidos para servirem de sincronização em caso de erro no arquivo. Os tokens escolhidos estão representados a seguir:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

**Processo de construção:**

O processo de construção do software foi iniciado utilizando a ferramenta auxiliar JavaCC, porém ao final do código gerado foi localizada uma falha na leitura de números inválidos, na qual o programa acabava por separar a palavra. Exemplo: 123bacon aparecia como número e identificador, ao invés de ocorrer o erro. Por conta da dificuldade em encontrar o problema e após maiores pesquisas foi encontrado um exemplo com implementação e desenvolvido a partir deste ponto na linguagem Java.

Após a realização de testes com o código encontrado como exemplo online foi decidido realizar adequações para que funcionasse com a linguagem pensada e do modo definido na especificação do trabalho.

As bibliotecas java.io.File e java.io.FileNotFoundException são utilizadas para manipulação do arquivo txt e geração de excessão caso o arquivo não seja encontrado.

As bibliotecas java.util.HashMap e java.util.Map são utilizadas na criação de um HashMap com as strings e tipos de tokens, para que os tokens fiquem guardados na lista. E Map faz parte da ligação das chaves aos valores.

Por fim a biblioteca java.util.Scanner é utilizada para a entrada do nome/caminho do arquivo e da leitura das informações de dentro do arquivo texto.

Tal como enfatizado anteriormente para utilizar o software é necessário ter o compilador para linguagem Java no computador. Também é necessário ter o Java instalado para que a linguagem seja interpretada posteriormente.

No caso da utilização do JavaCC é necessária a instalação e alteração em variáveis de ambiente do computador, porém essa solução foi descartada. Não foi utilizada IDE para o desenvolvimento, apenas o editor de texto Visual Studio Code e todos os testes e compilação foram realizados por meio de terminal (Powershell).

O processo de construção do analisador sintático incluiu uma reestruturação geral do código, tendo sido dividido em mais arquivos seguindo a estratégia dividir para conquistar. Também foram realizadas mudanças nos tokens lidos para que a gramática tivesse regras condizentes com os tokens.

Iniciou-se o desenvolvimento pela geração das regras mais simples e assim estas regras foram englobadas pelas mais complexas no final.

Para virtude de testes com realização mais fácil foi gerado um script de formato .bat com o seguinte conteúdo:

Tela de computador

Descrição gerada automaticamente

Esse script permitia que fosse enviado apenas um comando ao terminal para realização de todo o processo necessário para execução do programa, realizando a limpeza dos arquivos .class gerados.

**Exemplo de árvore:**

Para o código:

Tela de computador

Descrição gerada automaticamente

Os passos demonstrados no arvore\_fonte1.txt são:

Início do processamento com toda a cadeia lida

Tela de computador

Descrição gerada automaticamente

Entrada na regra BLOCO

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Entrada na regra IF

Tela de computador

Descrição gerada automaticamente

Entrada na regra IDENTIFICACAO e consumo do if

Tela de computador

Descrição gerada automaticamente

Entrada na regra TERMOID

Tela de computador

Descrição gerada automaticamente

Entrada na regra B, consumo do identificador e entrada na regra EXPRESSAO

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Entrada na regra A, entrada na regra TERMOINT e consumo do inteiro

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Entrada na regra BLOCO, entrada na regra RETORNO, consumo do comando de retorno, entrada na regra de IDENTIFICACAO, entrada na regra TERMOINT e consumo do inteiro

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Entrada na regra EXPRESSAO e entrada na regra A, encerrando a leitura do arquivo.

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

**Referências:**

**HashMap Java: Trabalhando com Listas key-value.** Disponível em: <https://www.devmedia.com.br/hashmap-java-trabalhando-com-listas-key-value/29811 >

**HashMap (Java Platform SE 8** **)**. Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

**Java.io.File Class in Java - GeeksforGeeks**. Disponível em: <https://www.geeksforgeeks.org/file-class-in-java/>

**java - Making a lexical Analyzer - Stack Overflow**. Disponível em: <https://stackoverflow.com/questions/17848207/making-a-lexical-analyzer>

**Implement simple lexical analysis in Java (linuxtut.com**). Disponível em: <https://linuxtut.com/en/d7ea5d0e0dbc0409a01f/>

**C program to detect tokens in a C program - GeeksforGeeks.** Disponível em: https://www.geeksforgeeks.org/c-program-detect-tokens-c-program/

**JavaCC.** Disponível em: <https://javacc.github.io/javacc/>

**p01.analise-lexica.pdf (ufop.br).** Disponível em: <http://www.decom.ufop.br/romildo/2014-1/bcc328/slides/p01.analise-lexica.pdf>

**Experimento 1: Diseño e implementación del programa de análisis léxico (implementación de Java) - programador clic.** Disponível em: <https://programmerclick.com/article/60381479536/>

**Compiler/lexical analyzer - Rosetta Code**. Disponível em: <https://rosettacode.org/wiki/Compiler/lexical\_analyzer>