UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE FLORESTAL
CIÊNCIA DA COMPUTAÇÃO

ATHENA SARANTÔPOULOS (2652)
JOSE GRIGORIO NETO (3046)
THIAGO OLIVEIRA (3037)
VITOR HUGO (3049)

COMPILADORES

TRABALHO PRÁTICO 2

FLORESTAL, MG
5 de novembro de 2020

ATHENA SARANTÔPOULOS (2652)
JOSE GRIGORIO NETO (3046)
THIAGO OLIVEIRA (3037)
VITOR HUGO (3049)

COMPILADORES
TRABALHO PRÁTICO 2

Documentação do 2º trabalho prático de Compiladores que tem como objetivo de criar uma linguagem de programação

Orientador: Prof. Dr. Daniel Mendez Barbosa

FLORESTAL, MG
5 de novembro de 2020

Resumo

Este trabalho tem por finalidade o desenvolvimento de uma linguagem de programação que possa ser reconhecida por uma gramática livre de contexto e computada por um autômato de pilha. Para esta linguagem foi implementado também um compilador, que é capaz de gerar código de máquina à partir de um código fonte escrito na mesma. A objetivo principal é fomentar a aplicação de maneira prática, os conceitos e o conteúdo teórico visto em sala de aula. Sendo assim, a criação da linguagem, que envolve a escolha dos tipos primitivos, escolhas de palavras chave, palavras reservadas, paradigma de programação, gramática a ser utilizada, dentre outras decisões de projeto, serão melhor explicadas ao longo deste documento. Ao fim deste trabalho espera-se que o leitor obtenha algum entendimento acerca das principais decisões tomadas, bem como o que levou a elas e como se dá o funcionamento da linguagem, tanto sintática quanto semanticamente, sendo assim capaz de desenvolver códigos nela.

Palavras-chaves: lex, yacc, gramática, compiladores, linguagem.

Abstract

This work aims to develop a programming language that can be recognized by a context-free grammar and computed by a stack automaton. A compiler has also been implemented for this language, which is capable of generating machine code from a source code written in it. The main objective is to promote practical application, concepts and theoretical content seen in the classroom. Therefore, the creation of the language, which involves the choice of primitive types, choice of keywords, reserved words, programming paradigm, grammar to be used, among other design decisions, will be better explained throughout this document. At the end of this work, the reader is expected to gain some understanding about the main decisions made, as well as what led to them and how the language works, both syntactically and semantically, thus being able to develop codes in it.

Key-words: lex, yacc, grammar, compilers.

Sumário

1 Introdução

Este trabalho consiste na criação de uma linguagem. Nessa primeira parte, iremos definir o nome da linguagem e origem do nome, tipos de dados primitivos, comandos disponíveis, paradigma de programação, palavras-chave, palavras reservadas, gramática da linguagem, com suas variáveis, terminais (tokens) e padrões de lexema dos tokens e a implementação do analisador léxico.

Figura 1 – Logo da linguagem

2 A Linguagem

A linguagem desenvolvida recebeu o nome de KUALA. A origem desse nome se deu pelo fato da linguagem ser inteiramente inspirada em animais, sendo todas as suas palavras-chave e palavras reservadas definidas com nomes de animais e além do fato de que o animal Coala dorme por dia 14 horas, sendo parecido com os membros do grupo. Apesar da inspiração incomum, esta linguagem incentiva os programadores (especialmente iniciantes) no aprendizado do idioma inglês, levando em consideração que todos os nomes de palavras reservadas e palavras chaves da linguagem são nomes de animais em inglês.

A linguagem criada, a KUALA, foi inspirada na linguagem de programação C. E por inspirada, quer dizer que ela possui o mesmo paradigma de programação, que é o paradigma estruturado e case sensitive, e os mesmos formatos de estruturas e comandos. Assim como na linguagem C os delimitadores de início e fim de escopo são marcados pelo uso de chaves. Os comandos de repetição (que serão melhor abordados na sessão Comandos) possuem a sintaxe idêntica a da linguagem C. Além disso a linguagem permite a criação de funções, podendo receber alguns valores como parâmetro e retornando um valor que corresponde ao tipo definido para a função. Da mesma forma, é possível a definição de variáveis, também de algum tipo primitivo. Possui o comando condicional *if* e outros dois tipos de comandos de repetição. Apesar da linguagem ser inspirada na linguagem de programação C, esta não possui todo o poder computacional e nem mesmo oferece todos os recursos que a linguagem C oferece a seus programadores. Porém para a finalidade deste trabalho ela pode ser considerada suficiente, já que apresenta as principais características de uma linguagem de alto nível, além de proporcionar o poder computacional de tais linguagens.

2.1 Dados primitivos

Como a linguagem foi construído sobre o paradigma estruturado, e é baseada na linguagem C, ela possui tipos de dados muito bem definidos, sendo assim uma linguagem fortemente tipado. Por não ser do paradigma orientado a objeto, a linguagem possui apenas os tipos e estruturas definidos na própria linguagem, não possibilitando ao programador a criação de uma nova estrutura de dados. Diferentemente de C que possui um tipo de dados chamado struct, que possibilita a criação de novas estruturas de dados, para esta linguagem este recurso não foi implementado. A justificativa para isso é o fato de que levaria a um maior esforço de implementação, aumentando assim o escopo do projeto da linguagem.

Para esta linguagem foi definido os tipos de dados: inteiro, decimal, texto e boole-

ano. Cada um desses tipos de dados foi representado com um nome de animal que possui uma inicial com a mesma letra do tipo de dados que ele representa. Cada um desses quatro tipos de dados citados estão representados abaixo.

Tipo de dados em C	Tipo de dados em KUALA
int	*ibis*
float	*frog*
string	*snake*
bool	*bug*

2.2 Comandos

Esta linguagem possui, como já mencionado, um comando condicional e dois comandos de repetição. O comando condicional é de fundamental importância em linguagens de programação, pois eles dão ao programador a possibilidade de mudar o fluxo de execução do programa. Isto pois, em muitos momentos é necessário avaliar uma condição para decidir qual o próximo trecho de código a ser executado. Levando isso em consideração, foi decidido no escopo do projeto que somente um comando condicional seria implementado, sem possibilidade de uso de comandos incondicionais, pois eles podem ser mal utilizados pelo programador.

O comando *if* recebe como parâmetro uma expressão lógica relacional, que deve retornar um valor booleano, sendo *verdadeiro* ou *falso* (este valor decide qual será o próximo trecho de código a ser executado). Após a expressão do comando é possível escrever um ou mais comandos e expressões, contanto que estejam entre os delimitadores de início e fim de escopo. Além disso pode haver também o comando *else* que determina um trecho de código a ser executado caso a expressão retorne um valor falso. Ao fim desta sessão será apresentada uma tabela com as relações entre os nomes dos comandos da linguagem C e KUALA.

Além deste, foram implementados dois comandos de repetição. Estes, assim como os comandos condicionais, também são fundamentais para uma linguagem de programação. Isto pois, as linguagens tem por finalidade possibilitar a automação de determinadas tarefas, que poderiam (ou deveriam) ser feitas por pessoas. Para este propósito, os comandos de repetição são necessários. Os comandos de repetição implementados foram: o comando *for* e o comando *while*.

Para o comando de repetição *for* foram implementadas duas formas de uso. Uma recebe como parâmetro três expressões, uma que será executada uma única vez ao início do comando, uma expressão lógica que verifica a condição de parada do algoritmo e uma que é executada a cada iteração do algoritmo. Após ela é executado a cada iteração do comando uma sequência de uma ou mais expressões e/ou comandos, contanto que estejam

entre os delimitadores de início e fim de escopo. No outro modo de uso deste comando, ao invés dele receber três comando, ele recebe uma variável e um vetor do mesmo tipo da variável. E a cada iteração do comando, esta variável recebe um elemento do vetor.

Por fim o comando *while*, diferentemente do comando *for* recebe somente uma expressão, que é avaliada a cada fim de iteração da mesma. Caso a condição seja *verdadeira* a execução continua por mais uma iteração. E assim como os outros comandos, ele também recebe um ou mais comandos e/ou expressões, dentro dos delimitadores de início e fim de escopo.

Abaixo é apresentada a relação de nomes dos comandos em linguagem C e KUALA.

Comandos em C	Comandos em KUALA
if	*iguana*
for	*fox*
while	*whale*

Por fim vale ressaltar que, para cada um dos comandos citados, não necessariamente deve haver um ou mais comandos e/ou expressões, já que eles devem funcionar mesmo que dentro dos delimitadores de escopo esteja um conjunto de zero expressões e comandos (conjunto vazio).

2.3 Paradigmas

Assim como já foi mencionado, esta linguagem possui como paradigma de programação, o paradigma estruturado. Este é um paradigma de programação simples que consiste em três elementos básicos, que são: sequência, decisão e iteração. Por ser um paradigma de programação relativamente simples, os comandos implementados para esta linguagem são suficientes para que esta linguagem funcione conforme o manda o paradigma.

2.4 Palavras-chave e palavras reservadas

As palavras-chave são identificadores predefinidos e que são reservados com significados especiais para o compilador. Nem sempre as palavras-chave são reservadas, mas na linguagem KUALA, as palavras-chave são palavras reservadas. Segue abaixo as palavras reservadas:

- ibis e todas as formas de sua escrita. Um exemplo: IbIs;
- frog e todas as formas de sua escrita. Um exemplo: Frog;

- snake e todas as formas de sua escrita.Um exemplo: SNAke;
- bug e todas as formas de sua escrita. Um exemplo: BUG;
- iguana e todas as formas de sua escrita. Um exemplo: IguANA;
- eel e todas as formas de sua escrita. Um exemplo: EeL;
- fox e todas as formas de sua escrita. Um exemplo: Fox;
- whale e todas as formas de sua escrita. Um exemplo: WHAle;
- kuala e todas as formas de sua escrita. Um exemplo: KUALa;
- rabbit e todas as formas de sua escrita. Um exemplo: RaBbiT;
- viper e todas as formas de sua escrita. Um exemplo: VIpEr;
- Desenho do coala*

2.5 Implementação

Na implementação iremos comentar sobre o trabalho realizado para se reconhecer as expressões regulares através do analisador Lex e reconhecer as gramáticas estabelecidas através do analisador YACC. Comentaremos ainda um pouco sobre detalhes práticos e planejamentos realizados para a linguagem criada.

2.5.1 Expressões

Para essa primeira parte do trabalho de uma criação de uma linguagem, foi implementado um analisador léxico feito através do Flex que só imprime os tokens quando há uma correspondência no fluxo de entrada. No Flex é preciso criar expressões regulares para o processamento léxico do fluxo de entrada. Segue abaixo essa implementação e sua explicação:

```
1  %{
2
3  #include "prints.h"
4  #include "defines.h"
5
6  %}
7
8  %option noyywrap
9
10 /* definicoes regulares */
11
```

```
12 delim [ \t\n]
13 ws {delim}+
14 upperCase [A-Z]
15 lowerCase [a-z]
16 bothCase [A-Za-z]
17 digit [0-9]
18 id {bothCase}({bothCase}|{digit})*
19 positive \+?{digit}+
20 negative \-[1-9]{digit}*
21 float \.{digit}+
22 decimal {positive}{float}
23 decimalNegative \-{digit}+{float}
24 especialChar [;|=|,|{|}|(|)|\[|\]|?|:|&|||^|!|~|%|<|>|\'|\/']
25 string \"(\\.|[^\])*\/"
26 simpleComments \/\/(.|[^\n])*
27 blockComments [/][*]([^*]|[*]*[^*/])*[*][/]
28
29 %%
30
31 {simpleComments}

      {bluePrint("Foi encontrado um comentario simples.",yytext);}
32
33 {blockComments}

      {bluePrint("Foi encontrado um comentario em bloco.",yytext);}
34
35 {ws}            ;
36
37 {string}

      {bluePrint("Foi encontrado uma STRING.",yytext);}
38
39 kuala

      {bluePrint("Foi encontrado um Kuala.",yytext);}
40
41 "desenho do koala*"

      {bluePrint("Foi encontrado um  desenho do koala*.",yytext);}
42
```

```
43 iguana
      {bluePrint("Foi encontrado um IF.",yytext);}
44
45 eel
      {bluePrint("Foi encontrado um ELSE.",yytext);}
46
47 whale
      {bluePrint("Foi encontrado um WHILE.",yytext);}
48
49 fox
      {bluePrint("Foi encontrado um FOR.",yytext);}
50
51 ibis
      {bluePrint("Foi encontrado um INT.",yytext);}
52
53 frog
      {bluePrint("Foi encontrado um FLOAT.",yytext);}
54
55 bug
      {bluePrint("Foi encontrado um BOOL.",yytext);}
56
57 snake
      {bluePrint("Foi encontrado um STRING.",yytext);}
58
59 viper
      {bluePrint("Foi encontrado um void.",yytext);}
60
61 {id}
      {bluePrint("Foi encontrado um ID.",yytext);}
62
63 {positive}
```

```
   {bluePrint("Foi encontrado um numero inteiro positivo.",yytext);}
64
65 {negative}

   {bluePrint("Foi encontrado um numero inteiro negativo.",yytext);}
66
67 {decimal}

   {bluePrint("Foi encontrado um numero com parte decimal.",yytext);}
68
69 "+"

   {bluePrint("Foi encontrado um +.",yytext);}//yylval = PLUS;
   return(OP);
70
71 "-"

   {bluePrint("Foi encontrado um -.",yytext);}//yylval = MINUS;
   return(OP);
72
73 "*"

   {bluePrint("Foi encontrado um *.",yytext);}//yylval = TIMES;
   return(OP);
74
75 "/"

   {bluePrint("Foi encontrado um /.",yytext);}//yylval = DIVIDE;
   return(OP);
76
77 "&&"

   {bluePrint("Foi encontrado um &&.",yytext);}//yylval = AND;
   return(RELOP);
78
79 "||"

   {bluePrint("Foi encontrado um ||.",yytext);}//yylval = OR;
   return(RELOP);
80
81 "<"
```

```
    {bluePrint("Foi encontrado um <.",yytext);}//yylval = LT;
    return(RELOP);
82
83  "<="

    {bluePrint("Foi encontrado um <=.",yytext);}//yylval = LE;
    return(RELOP);
84
85  "=="

    {bluePrint("Foi encontrado um ==.",yytext);}//yylval = EQ;
    return(RELOP);
86
87  "!="

    {bluePrint("Foi encontrado um !=.",yytext);}//yylval = NE;
    return(RELOP);
88
89  ">"

    {bluePrint("Foi encontrado um >.",yytext);}//yylval = GT;
    return(RELOP);
90
91  ">="

    {bluePrint("Foi encontrado um >=.",yytext);}//yylval = GE;
    return(RELOP);
92
93  {especialChar}

    {bluePrint("Foi encontrado um char especial.",yytext);}
94
95  %%
96
97  int main(void)
98  {
99      /* Call the lexer, then quit. */
100     yylex();
101     return 0;
102 }
```

2.5.1.1 Definições regulares

Na primeira parte do arquivo mostrado acima importamos dois arquivos um para realizar as mensagens de forma mais organizada no terminal e outro que usado para as definições dos tokens, apesar de ter sido definido nessa primeira parte do trabalho ele só será efetivamente usado na segunda parte, onde será gerado os tokens efetivamente. Após, isso definimos algumas expressões regulares que serão usadas tanto para apoio de outras como também para o reconhecimento. Segue abaixo essas definições:

- **delim** [\t\n] : Reconhece espaços vazios, tabulações e quebra de linha;
- **ws** delim+ : Reconhece um ou mais combinações de delim;
- **upperCase** [A-Z] : Reconhece letra maiúsculas;
- **lowerCase** [a-z] : Reconhece letra minusculas;
- **bothCase** [A-Za-z] : Reconhece tanto letra maiúscula e minúscula;
- **digit** [0-9] : Reconhece um digito de 0 a 9;
- **id** bothCase(bothCase|digit)* : Reconhece palavras que comecem com letras bothCase e esteja concatenado com 0 ou mais combinações de bothCase ou digit;
- **positive** ?digit+ : Reconhece palavras que comecem com o símbolo + e seguidos por uma ou mais combinações de digit;
- **negative** [1-9]digit* : Reconhece palavras que comecem com o símbolos - e seguidos por uma ou mais combinações de 1 a 9, seguidos por 0 ou mais combinações de digit;
- **float** .digit+ : Reconhece palavras que comecem com . seguidas de uma ou mais digit;
- **decimal** positivefloat : Reconhece uma palavra composta por um positive seguida por um float;
- **decimalNegative** digit+float : Reconhece uma palavra composta por um sinal - seguida por um ou mais digit concatenada com um float;
- **especialChar**

```
[;|=|,|{|}|(|)|\[|\]|?|:|&|||^|!|~|%|<|>|\'|\'] :
Reconhece os caracteres especiais compostos nela própria;
```

- **string**

```
\"(\\.|[\^\\])*\/" :
```

  Reconhece palavras compostas por um "seguido por qualquer palavra e finalizada com ";

- **simpleComments** (.|[^])* : Reconhece palavras compostas por dois // seguidas de qualquer palavra até uma quebra de linha;

- **blockComments** [/][*]([*̂]|[*]*[*̂/])*[*][/]: Reconhece um bloco de texto que comece com /* e termine com *\.

2.5.1.2 Regras de tradução

O segundo bloco do arquivo se refere as regras de traduções, nele fizemos o reconhecimento de todos os tokens e exibimos na tela os tokens reconhecidos, vale ressaltar que a ordem de declaração foi muito importante, pois se não fosse definido nessa ordem alguns tokens seriam reconhecidos pela metade e de forma incorreta. Segue abaixo essas expressões:

- **simpleComments** : Reconhece comentários simples, ou seja, frases seguidas por um //;
- **blockComments** : Reconhece comentários de bloco, ou seja, blocos que comecem com /* tenha alguns parágrafos e terminem com */;
- **ws** : Reconhece tabulações, espaços e quebra de linha;
- **string** : Reconhece strings, ou seja, frases iniciadas por "e terminadas por ";
- **kuala** : Reconhece a palavra kuala;
- **"desenho do koala*"** : Reconhece a o próprio símbolo;
- **iguana** : Reconhece a palavra iguana;
- **eel** : Reconhece a palavra eel;
- **whale** : Reconhece a palavra whale;
- **fox** : Reconhece a palavra whale;
- **ibis** : Reconhece a palavra ibis;
- **frog** : Reconhece a palavra frog;
- **bug** : Reconhece a palavra bug;
- **snake** : Reconhece a palavra snake;
- **viper** : Reconhece a palavra viper;
- **id** : Reconhece a palavras que serão usadas como identificadores;
- **positive** : Reconhece inteiros positivos;
- **negative** : Reconhece inteiros negativos;
- **decimal** : Reconhece um decimal positivo;
- **decimalNegative** : Reconhece um decimal negativo;

- "+" : Reconhece o operador operacional de **mais** ,que será alocado para uma tabela de símbolos;
- '-' : Reconhece o operador operacional de **menos** ,que será alocado para uma tabela de símbolos;
- "*" : Reconhece o operador operacional de **vezes**, que será alocado para uma tabela de símbolos;
- "/" : Reconhece o operador operacional de **divisão**, que será alocado para uma tabela de símbolos;
- "" : Reconhece o operador lógico **and**, que será alocado para uma tabela de símbolos;
- '||' : Reconhece o operador lógico **or**, que será alocado para uma tabela de símbolos;
- "<" : Reconhece o operador de comparação **menor** , que será alocado para uma tabela de símbolos;
- '<=' : Reconhece o operador de comparação **menor igual**, que será alocado para uma tabela de símbolos;
- "==" : Reconhece o operador de comparação **igual**, que será alocado para uma tabela de símbolos;
- '!=' : Reconhece o operador de comparação **diferente**, que será alocado para uma tabela de símbolos;
- '>' : Reconhece o operador de comparação **maior** , que será alocado para uma tabela de símbolos;
- '>= : Reconhece o operador de comparação **maior igual** , que será alocado para uma tabela de símbolos;
- **especialChar** : Reconhece uma lista de caracteres especiais já descritos nas definições regulares.

2.5.2 A gramática

Para a criação da gramática da linguagem KUALA, foi utilizado uma gramática como referência. Essa gramática está sendo mostrada logo abaixo.

program $\rightarrow$ *block*
block $\rightarrow'$ $\{'stmts'\}'$
stmts $\rightarrow$ *stmtsstmt*

| ϵ

$stmt \rightarrow$ **if**$'('expr')''\{'stmt'\}'$

| **if** '(' expr ')' '{' stmt '}' **else** '{' stmt '}'

| **while** '(' expr ')' '{' stmt '}'

| **for** '('expr ';' expr ';' expr ')' '{' stmt '}'

| **stmt**

| ϵ

$expr \rightarrow term$**relop**$term$

| expr **operational** term

| expr - term

| term

term $\rightarrow$ **id**

| term * factor

| term / factor

| factor

factor $\rightarrow$ **digit**$|'('expr')'$

$relop \rightarrow < | > | <= | >= | == |! =$

$id \rightarrow$ **id**$letter$

| **id**digit

| letter

letter $\rightarrow [A - Za - z]$

$digit \rightarrow 0|1|2|3|4|5|6|7|8|9$

Com essa gramática como referência foi possível criar a nossa própria gramática. Segue abaixo a gramática da linguagem KUALA.

program $\rightarrow block$

$block \rightarrow' \{'stmts'\}'$

$stmts \rightarrow stmtsstmt$

| ϵ

$stmt \rightarrow$ **iguana**$'('expr')''\{'stmt'\}'$

| **iguana** '('expr')' '{' stmt '}' **eel** '{' stmt '}'

| **whale** '(' expr ')' '{' stmt '}'

| **fox** '('expr ';' expr ';' expr ')' '{' stmt '}'

| **stmt**

| ϵ

expr → *term***relop***term*

| expr + term

| expr - term

| term

term → **id**

| term * factor

| term / factor

| factor

factor → **digit**|′(′*expr*′)′

relop →< | > | <= | >= | == |! =

id → **id***letter*

| **id**digit

| letter

letter → [*A* − *Za* − *z*]

digit → 0|1|2|3|4|5|6|7|8|9

2.6 Testes

Usamos o arquivo abaixo para testar se o analisador léxico estava reconhecendo corretamente a sintaxe proposta.

```
kuala(){
    //kuala hj amanha e sempre

    ibis int = 0;
    ibis array = [0,2,3,4]
    frog float = 1.0;
    bug bool = true;
    snake string = 'desenho do koala*';

    func(string)


    fox(ibis i=0;i<=10;i++){
        iguana(i<=2){
```

```
15              print(i+i);
16          }eel iguana(i<=5){
17              print(i-i);
18          }eel iguana(i<=8){
19              print(i*i);
20          }eel{
21              print(i/i);
22          }
23
24          iguana( 2 == i && i+2 == 4){
25              print("Logic yeahh!!!");
26          }
27
28          iguana(3==i || i+1==3){
29              print('Logic !!hhaey');
30          }
31
32          whale(i==2){
33              print("OooOoOOOOOOoooo0000");
34          }
35
36      }
37
38  }
39
40  viper func(snake str){
41      print("THIS IS A FUNCTION " + str);
42  }
43
44  /*
45
46     __           __
47   /"  "\       /"  "\
48  (  (\  )___(  /)  )
49   \                 /
50   /                 \
51  /     () ___ ()     \
52  |       (   )       |
53   \       \_/       /
54     \...__!__.../
55           "
56  MELHOR LINGUAGEM DO MUNDO.
```

```
57
58 88                                88
59 88                                88
60 88                                88
61 88   ,d8 aa       aa ,adPPYYba, 88 ,adPPYYba,
62 88 ,a8"  a8       8a ""     ‘Y8 88 ""     ‘Y8
63 8888[    8b       d8 ,adPPPPP88 88 ,adPPPPP88
64 88‘"Yba, "8a,   ,a8" 88,    ,88 88 88,    ,88
65 88   ‘Y8a ‘"YbbdP"’  ‘"8bbdP"Y8 88 ‘"8bbdP"Y8
66
67 */
```

Abaixo vemos os resultados da análise léxica sobre esse arquivo.

```
vitor@vitor-PC: ~/Desktop/Facu/PER/Compiladores/Tp_2
vitor in Tp_2 on  master [↑]
❯ ./a.out < kuala.kl
kuala LEXEMA: kuala
char especial LEXEMA: (
char especial LEXEMA: )
char especial LEXEMA: {
comentario simples LEXEMA: //kuala hj amanha e sempre
INT LEXEMA: ibis
ID LEXEMA: int
char especial LEXEMA: =
numero inteiro positivo LEXEMA: 0
char especial LEXEMA: ;
INT LEXEMA: ibis
ID LEXEMA: array
char especial LEXEMA: =
char especial LEXEMA: [
numero inteiro positivo LEXEMA: 0
char especial LEXEMA: ,
numero inteiro positivo LEXEMA: 2
char especial LEXEMA: ,
numero inteiro positivo LEXEMA: 3
char especial LEXEMA: ,
numero inteiro positivo LEXEMA: 4
char especial LEXEMA: ]
FLOAT LEXEMA: frog
ID LEXEMA: float
char especial LEXEMA: =
numero com parte decimal LEXEMA: 1.0
char especial LEXEMA: ;
BOOL LEXEMA: bug
ID LEXEMA: bool
char especial LEXEMA: =
ID LEXEMA: true
char especial LEXEMA: ;
STRING LEXEMA: snake
ID LEXEMA: string
char especial LEXEMA: =
char especial LEXEMA: '
·?·?·? LEXEMA: ?·?·?
char especial LEXEMA: '
char especial LEXEMA: ;
ID LEXEMA: func
char especial LEXEMA: (
ID LEXEMA: string
char especial LEXEMA: )
FOR LEXEMA: fox
char especial LEXEMA: (
INT LEXEMA: ibis
ID LEXEMA: i
char especial LEXEMA: =
numero inteiro positivo LEXEMA: 0
char especial LEXEMA: ;
ID LEXEMA: i
<= LEXEMA: <=
numero inteiro positivo LEXEMA: 10
char especial LEXEMA: ;
ID LEXEMA: i
+ LEXEMA: +
+ LEXEMA: +
char especial LEXEMA: )
char especial LEXEMA: {
IF LEXEMA: iguana
char especial LEXEMA: (
ID LEXEMA: i
<= LEXEMA: <=
numero inteiro positivo LEXEMA: 2
char especial LEXEMA: )
char especial LEXEMA: {
ID LEXEMA: print
char especial LEXEMA: (
ID LEXEMA: i
+ LEXEMA: +
ID LEXEMA: i
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
ELSE LEXEMA: eel
IF LEXEMA: iguana
char especial LEXEMA: (
ID LEXEMA: i
<= LEXEMA: <=
numero inteiro positivo LEXEMA: 5
char especial LEXEMA: )
char especial LEXEMA: {
ID LEXEMA: print
char especial LEXEMA: (
ID LEXEMA: i
- LEXEMA: -
ID LEXEMA: i
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
ELSE LEXEMA: eel
IF LEXEMA: iguana
char especial LEXEMA: (
ID LEXEMA: i
<= LEXEMA: <=
numero inteiro positivo LEXEMA: 8
char especial LEXEMA: )
char especial LEXEMA: {
ID LEXEMA: print
char especial LEXEMA: (
ID LEXEMA: i
* LEXEMA: *
ID LEXEMA: i
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
ELSE LEXEMA: eel
char especial LEXEMA: {
ID LEXEMA: print
char especial LEXEMA: (
ID LEXEMA: i
/ LEXEMA: /
ID LEXEMA: i
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
IF LEXEMA: iguana
char especial LEXEMA: (
numero inteiro positivo LEXEMA: 2
== LEXEMA: ==
ID LEXEMA: i
&& LEXEMA: &&
ID LEXEMA: i
numero inteiro positivo LEXEMA: +2
== LEXEMA: ==
numero inteiro positivo LEXEMA: 4
char especial LEXEMA: )
char especial LEXEMA: {
ID LEXEMA: print
```

```
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
IF LEXEMA: iguana
char especial LEXEMA: (
numero inteiro positivo LEXEMA: 2
== LEXEMA: ==
ID LEXEMA: i
&& LEXEMA: &&
ID LEXEMA: i
numero inteiro positivo LEXEMA: +2
== LEXEMA: ==
numero inteiro positivo LEXEMA: 4
char especial LEXEMA: )
char especial LEXEMA: {
ID LEXEMA: print
char especial LEXEMA: (
STRING LEXEMA: "Logic yeahh!!!"
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
IF LEXEMA: iguana
char especial LEXEMA: (
numero inteiro positivo LEXEMA: 3
== LEXEMA: ==
ID LEXEMA: i
|| LEXEMA: ||
ID LEXEMA: i
numero inteiro positivo LEXEMA: +1
== LEXEMA: ==
numero inteiro positivo LEXEMA: 3
char especial LEXEMA: )
char especial LEXEMA: {
ID LEXEMA: print
char especial LEXEMA: (
char especial LEXEMA: '
ID LEXEMA: Logic
char especial LEXEMA: !
char especial LEXEMA: !
ID LEXEMA: hhaey
char especial LEXEMA: '
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
WHILE LEXEMA: whale
char especial LEXEMA: (
ID LEXEMA: i
== LEXEMA: ==
numero inteiro positivo LEXEMA: 2
char especial LEXEMA: )
char especial LEXEMA: {
ID LEXEMA: print
char especial LEXEMA: (
STRING LEXEMA: "OooOoOOOOOOoooo0000"
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
char especial LEXEMA: }
char especial LEXEMA: }
void LEXEMA: viper
ID LEXEMA: func
char especial LEXEMA: (
STRING LEXEMA: snake
ID LEXEMA: str
char especial LEXEMA: )
char especial LEXEMA: {
ID LEXEMA: print
char especial LEXEMA: (
STRING LEXEMA: "THIS IS A FUNCTION "
+ LEXEMA: +
ID LEXEMA: str
char especial LEXEMA: )
char especial LEXEMA: ;
char especial LEXEMA: }
comentario em bloco LEXEMA: /*

   __           __
  /"  "\       /"  "\
 (  (\  )___(  /)  )
  \                 /
  /                 \
 /    () ___ ()     \
 |       (   )       |
  \       \_/       /
   \...__!__.../
          "
MELHOR LINGUAGEM DO MUNDO.

88                                88
88                                88
88                                88
88   ,d8 aa       aa ,adPPYYba, 88 ,adPPYYba,
88 ,a8"  a8       8a ""     `Y8 88 ""     `Y8
8888[    8b       d8 ,adPPPPP88 88 ,adPPPPP88
88`"Yba, "8a,   ,a8" 88,    ,88 88 88,    ,88
88   `Y8a `"YbbdP"'  `"8bbdP"Y8 88 `"8bbdP"Y8

*/
```

Figura 3 – Resultados 2

3 Consideração

* Não foi possível, colocar o desenho do koala no arquivo latex devido a incompatibilidade dos caracteres unicode, mas segue uma imagem do que deveria ser.

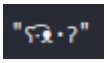

Figura 4 – Desenho do koala

4 Conclusão

Ao fim deste trabalho, foi possível obter algum entendimento mais concreto acerca do desenvolvimento, tanto de uma linguagem quanto de um compilador capaz de interpretar os comandos desenvolvidos na mesma. Além disso, também foi possível contemplar (não necessariamente com admiração) os desafios enfrentados durante o desenvolvimento de um projeto de linguagem e compilador, tanto pela seleção do paradigma, quanto dos tipos de estruturas utilizadas para representar os elementos da mesma. Por fim, foi proveitoso o contato com as ferramentas que auxiliam no desenvolvimento deste tipo de programa, referindo-se tanto ao yacc, quanto ao flex e outras ferramentas.

5 Referências

Lex - A Lexical Analyzer Generator . Disponível em: <http://dinosaur.compilertools.net/lex/> Acesso em: 23 de out.

Yacc: Yet Another Compiler-Compiler . Disponível em: <http://dinosaur.compilertools.net/yacc/> Acesso em: 23 de out.

Freepik. catalyststuff . Disponível em: <https://www.freepik.com/free-vector/cute-koala-sleep 10555272.htm#page=1&query=drawing%20animal&position=10>. Acesso em: 20 de out. de 2020.