UNIVERSIDADE FEDERAL DE VIÇOSA – CAMPUS FLORESTAL

DANIEL FREITAS MARTINS – 2304

JOÃO ARTHUR GONÇALVES DO VALE – 3025

MARIA DALILA VIEIRA – 3030

NAIARA CRISTIANE DOS REIS DINIZ – 3005

Relatório referente ao Trabalho Prático 2 – Criação de uma linguagem de programação, gramática e analisador léxico

Florestal

2020

DANIEL FREITAS MARTINS – 2304

JOÃO ARTHUR GONÇALVES DO VALE – 3025

MARIA DALILA VIEIRA – 3030

NAIARA CRISTIANE DOS REIS DINIZ – 3005

Relatório referente ao Trabalho Prático 2 – Criação de uma linguagem de programação, gramática e analisador léxico

Documentação apresentada à disciplina CCF 441 – Compiladores do curso de Bacharelado em Ciência da Computação da Universidade Federal de Viçosa – *Campus* Florestal.

Orientador: Daniel Mendes Barbosa

Florestal

2020

SUMÁRIO

1 - Introdução

Neste trabalho será apresentada a linguagem Chameleon, uma nova linguagem de programação. Essa linguagem foi criada pelo grupo com a finalidade de oferecer simplicidade e agilidade na programação. Ademais, a linguagem foi idealizada a fim de auxiliar a programação por parte de usuários portadores de deficiência visual, oferecendo suporte de customização através de seu pré-processador. Sendo assim, o pré-processador da linguagem oferece uma flexibilidade muito grande para a customização de códigos pelo programador e é totalmente opcional. A estrutura da linguagem e de seus recursos adicionais serão descritos com maiores detalhes nas próximas seções.

Este documento está assim organizado: a Seção 2 introduz a linguagem Chameleon, abordando suas características e seu pré-processador; a Seção 3 discute detalhes a respeito da implementação do arquivo lex.l para a construção do analisador léxico da linguagem, bem como as palavras-chave e palavras reservadas da linguagem e algumas de suas restrições; a Seção 4 apresenta programas válidos e inválidos lexicalmente para a linguagem, junto de seus testes com a utilização do analisador léxico; a Seção 5 apresenta e discute os testes realizados no JFLAP; a Seção 6 apresenta as considerações finais deste trabalho; na sequência as referências são apresentadas e os Apêndices A e B descrevem o corpo dos códigos para o FLEX e o corpo do código em Python para o pré-processador da linguagem, respectivamente.

2 - A Linguagem Chameleon

Como dito na seção anterior, a linguagem de programação Chameleon pretende oferecer um conjunto de comandos que seja interessante para programadores que desejam agilidade, simplicidade e espaço para customização. Sua construção foi baseada nas linguagens C, C++ e Python. Assim, temos uma linguagem imperativa procedural que segue o paradigma estrutural. A origem deste nome veio justamente desta característica de customização da linguagem com o uso de seu pré-processador. Em resumo, ela permite definir novas formas de escrita que são convertidas para a

sintaxe padrão da linguagem. Essa ideia é similar ao uso de macros na linguagem C, mas com um propósito diferente.

A logo da linguagem Chameleon pode ser visualizada na Fig. 2.1 a seguir. Maiores detalhes a respeito dessa linguagem serão abordados nas próximas seções.

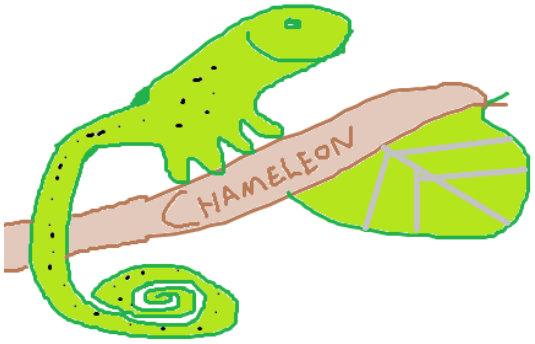

Figura 2.1: Logo da linguagem Chameleon. Fonte: Autores, 2020.

2.1 - Características

Nesta seção serão abordadas as características mais específicas da linguagem. A linguagem possui três tipos primitivos, equivalentes aos tipos **inteiro**, **string** e **real**. O tipo **caractere** não foi considerado pois este pode ser entendido como uma **string** de um caractere, apenas. Suas palavras-chave são consideradas palavras reservadas pela linguagem, isto é, identificadores não podem ser usados com o mesmo nome de palavras-chave da linguagem. As Tabelas 2.1.1, 2.1.2 e 2.1.3 abaixo mostram as equivalências dos comandos, símbolos e palavras-chave das linguagens C, C++ e Python para a nossa linguagem. Os operadores aritméticos e relacionais são os mesmos das linguagens C, com exceção dos operadores **E** e **OU** lógicos, que são os mesmos de Python, **and** e **or**, e da possibilidade de se efetuar uma operação de potência, com o uso do operador ^. Da mesma maneira que a linguagem C, a linguagem Chameleon considera que um número igual a 0 corresponde ao valor lógico *false* e um valor diferente de 0 corresponde ao valor lógico *true*.

Equivalências dos comandos em C para Chameleon	
C	**Chameleon**
if(*expr*) { *stmt* }	if *expr*: *stmt* endif
if(*expr*) { *stmt* } else { *stmt* }	if *expr*: *stmt* elif: *stmt* endif
for(*expr;expr;expr*) { *stmt* }	for *expr*, *expr*, *expr*: *stmt* endfor
while(*expr*) { *stmt* }	while *expr*: *stmt* endwhile
return *expr*	farewell *expr*
break	stop
goto	jumpto
printf(...)	say *expr*
scanf(...)	listen *variable*
int	integer
float	real
/*comentário*/	// comentário \\
{	begin
}	end
struct ...	squad ...

Tabela 2.1.1: Correspondência de sintaxe de comandos, símbolos e palavras-chave da linguagem C para a linguagem Chameleon.

Equivalências dos comandos em C++ para Chameleon	
C++	**Chameleon**
string	word
vector	vector

Tabela 2.1.2: Correspondência de sintaxe de comandos, símbolos e palavras-chave da linguagem C++ para a linguagem Chameleon.

Equivalências dos comandos em Python para Chameleon	
Python	**Chameleon**
function *id*(*params*): *stmt*	task *id params* : *stmt* endtask

Tabela 2.1.3: Correspondência de sintaxe de comandos, símbolos e palavras-chave da linguagem Python para a linguagem Chameleon.

2.2 - A Gramática

As principais referências para a construção da gramática foram [1] e [2]. No entanto, o grupo tentou desenvolver a gramática inicialmente sem influências de gramáticas já existentes, sendo um desafio interessante pensar em toda a estrutura que a mesma deveria ter. Neste sentido, ao consultar tais referências, a gramática definida para esta linguagem precisou sofrer diversas alterações e correções até chegar na versão definitiva mostrada abaixo. Ela está representada seguindo a convenção adotada nas aulas: em negrito, terminais (*tokens*); em itálico, variáveis. Os padrões de lexemas dos *tokens* podem ser encontrados no arquivo **lex.l** no Apêndice A deste documento.

A variável de partida da gramática é a variável *block*. Isso significa que todo código produzido na linguagem Chameleon deve começar com a palavra-chave **begin** e encerrar com a palavra-chave **end**.

word_expression → **word_value**
| *word_expression* **word_concat_operator** *expression*
| *expression* **word_concat_operator** *word_expression*

type → **integer**
| **word**
| **real**

label → *identifier* **;** *command*

variable_declaration → *type* **identifier**
| *squad_declaration*
| *vector_declaration*

variable_declarations → *variable_declaration*
| *variable_declaration variable_declarations*

vector_access → *identifier* **[** *number* **]**

squad_access → **identifier squad_access_derreference identifier**
| *squad_access* **squad_access_derreference identifier**

variable → **identifier**
| *vector_access*
| *squad_access*

squad_declaration → **squad** *identifier* **:** *variable_declarations* **endsquad**

vector_declaration → **vector identifier number**
| **vector identifier identifier**

command → *variable* **attribution** *expression*
| *if_command*
| *for_command*
| *while_command*
| *farewell_command*
| **stop**
| *jumpto_command*
| *say_command*
| *listen_command*
| *task_command*
| *label*

statement → **ε**
| *command statement*
| *variable_declaration statement*

if_command → **if** *expression* **:** *statement* **endif**
| **if** *expression* **:** *statement* **elif :** *statement* **endelif**

for_command → **for** *expression* **,** *expression* **,** *expression* **:** *statement* **endfor**

while_command → **while** *expression* **:** *statement* **endwhile**

farewell_command → **farewell** *expression*

jumpto_command → **jumpto** *identifier*

say_command → **say** *expression*

listen_command → **listen** *variable*

block → **begin** *statement* **end**

task_command → **task** *identifier task_parameters* **:** *expression* **endtask**

task_parameter → *expression*
| *expression* **,** *task_parameter*

task_parameters → **ε**
| *task_parameter*

task_call → **task** *identifier* **(** *task_parameters* **)**

add_expression → **add_operator** *expression*
| *expression* **add_operator** *expression*

div_expression → *expression* **div_operator** *expression*

pow_expression → *expression* **pow_operator** *expression*

logic_expression → *expression* **logic_operator** *expression*
| **neg_operator** *expression*

rel_expression → *expression* **rel_operator** *expression*

expression → *add_expression*
| *div_expression*
| *pow_expression*
| *logic_expression*
| *rel_expression*
| **(** *expression* **)**
| **number**
| **real_number**
| *word_expression*
| *variable*
| *variable* **attribution** *expression*

2.3 - O Pré-processador

O nosso sistema de macros, diferentemente do compilador, foi desenvolvido em Python e é um módulo completamente desacoplado do resto do sistema. O objetivo é ter um módulo fácil de implementar e com atualizações e correções independentes do compilador. Assim, ele é apenas uma interface para programadores que desejam uma experiência Chameleon completa. Seu código pode ser encontrado no Apêndice B deste documento.

Como mencionado anteriormente, o principal diferencial de Chameleon é ser uma linguagem que consegue mudar sua interface externa para se adaptar ao meio em que se insere. A motivação para a criação dela foi a ideia de metalinguagem, em que a própria linguagem poderia se definir por meio de seus próprios termos. Dessa maneira, decidimos por elaborar um sistema de macros para fazer modificações pontuais no código fonte de modo a permitir personalização do usuário na maneira em que ele irá programar. Essa pluralização da sintaxe da linguagem pretende atender a diferentes expectativas individuais de diferentes tipos de programadores de linguagens.

Sobretudo, empenhamos esforços em criar um sistema de macros intuitivo e que permita aos usuários com algum tipo de deficiência utilizarem uma macro predefinida que o auxilie na programação. Por exemplo, pessoas com deficiência visual podem optar por ter os símbolos importantes da linguagem, como palavras chaves e operadores, escritos por extenso, objetivando-se o uso de ferramentas que transformem texto escrito em áudio. Assim, algumas macros pré-definidas poderiam ser disponibilizadas para que a pessoa possa importar e usar.

Os arquivos de entrada para o pré-processador têm o seguinte formato:

```
#MACROS
    <KEY (Chameleon)> : <Value (Custom)>
    … (mais pares key-value, ordem não importa)
#ENDMACROS
#COD
    <Código do usuário>
#ENDCOD
```

No arquivo compactado, entregue junto deste trabalho, o código do preprocessador se encontra em um arquivo chamado *preprocessador.py*. Execute-o passando um arquivo de entrada definido de acordo com o padrão acima:

python preprocessador.py NOME_ARQUIVO

Um arquivo com o mesmo nome do arquivo, mas com um .out ao final de seu nome será gerado. O arquivo gerado irá conter o código equivalente para a linguagem Chameleon.

A Fig. 2.3.1 a seguir representa exemplo possível de macro para auxiliar pessoas com deficiência visual. Como mencionado acima, os símbolos são escritos por extenso. A Fig. 2.3.2 mostra este mesmo exemplo com as macros aplicadas, convertendo o código informado para a linguagem Chameleon.

```
#MACROS
        farewell : retorna
        task : funcao
        stop : pare
        jumpto : vaipara
        say : fala
        listen : escuta
        if : se
        elif : senao
        real : real
        word : texto
        integer : inteiro
        vector : vetor
        squad : esquadrao
        + : mais
        - : menos
        * : multiplica
        / : divide
        % : modulo
        = : recebe
        == : igual
        ^ : elevadoa
        ! : fatorial
        != : diferente
        >= : maiorouigual
        <= : menorouigual
        > : maior
        < : menor
        and  : E
        or :  OU
        , : virgula
        . : .
```

```
33              for : para
34              while : enquanto
35              // : iniciacomentario
36    \\ : terminacomentario
37              def : 1 : Verdadeiro
38              def : 0 : Falso
39              block : C
40  #ENDMACROS
41  #COD
42  iniciacomentario Codificacao para ajudar algumas classes de PCD/iniciantes terminacomentario
43  funcao p{
44              retorna 0
45  }
46  squad pessoa {
47              texto nome
48              inteiro idade
49  }
50  inteiro num
51  num recebe Falso
52  inteiro valor
53  valor recebe 5
54  vetor vet 5
55  texto txt "aprendendo a programar"
56
57  se valor menor 3{
58              num recebe True
59  }senao valor menor 2 e valor diferente 0{
60              num recebe 5 divide 2fatorial
61  }
62
63  enquanto valor maiorouigual 0{
64              valor recebe valor menos 1
65              se valor menor 2{
66                      pare
67              }
68  }
69
70  for valor recebe 0, valor maior 5, valor recebe valor mais 1{
71              num recebe num elevadoa 2
72              vet[4] recebe 5 multiplica 3 modulo dois
73  }
74  funcao valor
75  #ENDCOD
```

Figura 2.3.1 - Exemplo de macro e código para auxiliar pessoas com deficiência visual.

```
1   begin
2   // Codificacao for ajudar algumas clasifs de PCD/iniciantes \\
3   task p:
4   farewell 0
5   endtask
6   squad pessoa :
7   word nome
8   integer idade
9   endsquad
10  integer num
11  num = Falso
```

```
integer valor
valor = 5
vector vet 5
word txt
txt = "aprendendo a programar"

if valor < 3:
num = True
elif valor < 2 e valor != 0:
num = 5 / 2!
endelif

while valor >= 0:
valor = valor - 1
if valor < 2:
stop
endif
endwhile

for valor = 0, valor > 5, valor = valor + 1:
num = num ^ 2
vet[4] = 5 * 3 % dois
endfor
task valor
end
```

Figura 2.3.2 - Código da Fig. 2.3.1 com as macros aplicadas. Note que é lexicalmente correto conforme a linguagem Chameleon (veja Seção 4).

A implementação desse recurso, como visto na imagem, engloba:

- Renomear palavras-chave, para que o usuário possa utilizar palavras em sua língua materna ou qualquer outra de sua preferência. E também para que use termos de linguagens que ele já tem domínio, como mostrado nos exemplos a seguir, para Python:

```
#MACROS
   farewell : return
   task : function
   stop : break
   jumpto : goto
   say : print
   listen : input
   elif : else
   real : double
   word : str
   integer : int
   vector : array
   squad : class
   ^ : **
   ! : not
   block : C
#ENDMACROS
```

```
#COD
// imprime se os numeros sao pares ou impares, de 1 a 50 \\
int N
N = 50

function verificarParidade numero{
   if (numero%2 == 0){
      print numero + " eh par"
   }else {
      print numero + " eh impar"
   }
}

while N <= 50{
   function verificarParidade(N)

   N = N + 1
}
#ENDCOD
```

Figura 2.3.3 - Exemplo de macro e código para sintaxes similar a Python.

```
begin
// imprime se os numeros sao pares ou impares, de 1 a 50 \\
integer N
N = 50

task verificarParidade numero:
if (numero%2 == 0):
say numero + " eh par"
elif :
say numero + " eh impar"
endelif
endtask

while N <= 50:
task verificarParidade(N)

N = N + 1
endtask
end
```

Figura 2.3.4 - Código da Fig. 2.3.3 com as macros aplicadas. Note que é lexicalmente correto conforme a linguagem Chameleon (veja Seção 4).

- Renomear operadores, para permitir a descrição por extenso na língua falada desejada pelo programador. Esse item almeja facilitar a programação para PCD e pessoas que estão iniciando pequenas tarefas e aprendizados em lógica de programação.

- Redefinição de tipos semelhantes ao typedef de C. podem ser implementados com (def : 1 : True; def : 0 : False;), para possibilitar o tipo **boolean** que não é nativamente oferecido por Chameleon.
- Por fim, decidimos que o modo como a linguagem faz a delimitação de blocos pode ser fundamental para a adequação do usuário. Isso significa que algumas pessoas têm mais facilidade em se organizar com os esquemas de begin/end de Pascal, outras com as tabulações em Python, ou ainda, as chaves de C. Para isso, inicialmente permitimos 3 tipos de definição de blocos: o padrão Chameleon que tem variações de begin/end para cada comando, o padrão begin/end de Pascal e o padrão C.

No entanto, como o desenvolvimento tanto compilador como um todo como o sistema de macros são tarefas incrementais, ainda há muitas melhorias que podem e serão feitas. Nesta primeira versão, encontramos e reportamos 3 principais limitações do pré-processador:

- Palavras-chave não podem ser *substring* em identificadores ou em cadeias de caracteres, se não, elas também serão substituídas no pré-processador.
- O sistema de redefinições de tipo só aceita uma redefinição por padrão ou tipo, isto é, se a palavra-chave **integer** for redefinida para “biscoito”, por exemplo, ela não poderia ter uma terceira redefinição.
- Finalmente, ainda não implementamos delimitação de blocos por tabulação. Essa é uma melhoria interessante que será trabalhada nas próximas etapas deste trabalho.

3 - O Analisador Léxico - Versão preliminar - FLEX

Nesta versão, o analisador léxico para a linguagem tem como objetivo apenas fazer a impressão na tela da mesma forma como foi feito no trabalho prático 1 desta disciplina, sendo um analisador léxico *stand-alone*, demonstrando a identificação correta dos *tokens*. O código **lex.l** pode ser visualizado no apêndice A. Ele define os padrões de lexemas dos *tokens* que podem ser aceitos. Além disso, ele define outras

regras que tornam possível algumas operações curiosas, como por exemplo ignorar o não casamento de certos padrões de *tokens*, finalizando com os *tokens* reconhecidos, apenas, sem acusar erros. Isso é feito se houver um casamento da *string* SUPRESS_ERRORS no arquivo de entrada. Mais detalhes a respeito de algumas curiosidades a respeito do analisador léxico são descritas ao longo desta seção. Para gerar o analisador léxico com a ferramenta FLEX, basta passar o arquivo de acordo com a especificação do LEX para o comando *flex*. O arquivo gerado, *lex.yy.c*, pode ser compilado para gerar o arquivo *a.out*. Com isso, pode-se utilizar este novo arquivo gerado para que se faça a análise léxica de um outro arquivo de entrada. Em resumo, os comandos a serem executados sequencialmente são:

flex lex.l

gcc lex.yy.c

a.out < arquivo_entrada_analise_lexica

O seguinte trecho de código (Fig. 3.1) ilustra uma das decisões que permitiram a simplificação do uso de *strings* para serem imprimidas no código. Cada pedaço de *string* diz respeito a uma macro pelo uso de #define e, para concatenar duas *strings* criadas dessa maneira, basta colocar uma macro ao lado da outra. Uma macro PRINT foi criada para facilitar o uso do comando *printf* e o uso de ponto e vírgula [4]. Essa macro define que os argumentos passados para ela serão aplicados ao comando *printf*. A função *remover_espacoes_e_print* na linha 12 tem como objetivo remover espaços em branco, tabulações e quebras de linha, de modo a permitir entradas de valores como se o texto fosse contínuo em uma só linha, por exemplo. As linhas 13 e 14 definem constantes auxiliares para identificação de valores do tipo **real** e **integer**, e as linhas 16 e 17 definem variáveis auxiliares para marcar se a supressão de erros deverá ser considerada e se erros foram encontrados, respectivamente. Este último é particularmente útil para, ao final, retornar um erro se erros léxicos forem encontrados, quando a *flag* de supressão de erros estiver desligada (valor 0). Isso permite que mais de um erro possa ser identificado, antes de encerrar a função do analisador léxico, deixando o programador ciente de demais erros para serem resolvidos.

```
1  %{
2  #include <string.h>
3  #define PRINT_ERROR "---------- Erro encontrado na linha %d ----------\n"
4  #define PRINT_ERROR_EOF "-> Um ou mais erros foram encontrados. Corrija-os!\n"
5  #define PRINT_PREFIX "Line %d: I've found a"
6  #define PRINT(args) printf args ;
7  #define PRINT_LEXEME " LEXEME: %s\n"
8  #define PRINT_REAL_NUMBER PRINT_PREFIX " real number."
9  #define PRINT_NUMBER PRINT_PREFIX "n integer number."
10 #define PRINT_WORD PRINT_PREFIX " word."
11 /* outros #define... */
12 void remover_espacos_e_print(int t);
13 #define _REAL 1
14 #define _NUMBER 2
15
16 int supress_errors_flag = 0;
17 int erro_encontrado = 0;
18 %}
```

Figura 3.1 - Seção de Declarações/Definições do arquivo lex.l (parte 1).

No trecho de código abaixo (Fig. 3.2), as linhas 2 e 3 definem regras exclusivas, isto é, quando ativadas, todas as regras que não possuem essa condição de ativamento não são consideradas pelo FLEX a partir do momento de sua ativação. A linha 5, por sua vez, define uma regra inclusiva, isto é, quando ativada, as regras que não a possuem continuam a funcionar. A linha 7 informa para o FLEX para contabilizar o nº de linhas à medida que o arquivo de entrada vai sendo avaliado. As linhas 8 em diante definem os padrões de lexemas para a linguagem. Em particular, a linha 8 define o padrão "SUPRESS_ERRORS", que quando lido, ativa a condição inclusiva da linha 5 e permite que códigos com erros léxicos passem para a próxima etapa do compilador, passando apenas os *tokens* identificados nesta fase. A linha 19 declara um padrão para a impressão dos créditos dos criadores da linguagem, em que são informados os nomes e matrículas de cada integrante do grupo.

```
1  /* condicao exclusiva (bloqueia as demais regras) */
2  %x comment_condition
3  %x word_condition
4  /* condicao que e ativa mas mantem as demais ativadas tambem */
5  %s supress_errors_condition
6  /* Permitir a contabilizacao de linhas */
7  %option   yylineno
8  supress_errors              "SUPRESS_ERRORS"
9  /* captura uma ocorrencia de espaco em branco, tabulacao ou quebra de linha*/
```

```
10  delim                       [ \t\n\r]
11  /* ign (ignorador) ira ignorar um ou mais delim*/
12  ign                         {delim}+
13  letter                      [A-Za-z]
14  digit                       [0-9]
15  word_value                  (\\.|[^"\\])*
16  number                      {ign}*({digit}{ign}*)+
17  type                        "integer"|"word"|"real"
18  /* OUTRAS REGRAS (VEJA APENDICE A PARA VER O RESTANTE) */
19  creditos                    "??creditos??"
20  %%
```

Figura 3.2 - Seção de Declarações/Definições do arquivo lex.l (parte 2).

A Figura 3.3 abaixo ilustra parte da seção de regras do arquivo lex.l. As linhas 2, 4 e 10 iniciam as regras anteriormente citadas. A regra ativada pela linha 2 serve para permitir códigos inválidos lexicalmente a partir do momento do casamento de padrão correspondente, como comentado anteriormente. A linha 4 inicia a condição de comentário que, em conjunto com as linhas 7 e 8, ignoram tudo entre as tags de início e fim de comentário de múltiplas linhas (nesta linguagem não há comentários de apenas uma linha). A linha 10 inicia a condição de início de *word* que, em conjunto com as linhas 11 e 12, é capaz de identificar cadeias de caracteres identificados como *word* (equivalente a *string* em linguagens como C++).

As demais regras são fáceis de serem entendidas, porém, a regra da linha 26 merece uma atenção especial: ao término da análise de todo o arquivo pelo analisador léxico, este verifica se a variável auxiliar ***erro_encontrado*** é um valor diferente de zero. Em caso positivo, isso significa que a supressão de erros está desativada e, em caso de ter sido encontrado algum erro léxico durante o processamento, uma mensagem de erro é informada pedindo para que o programador corrija seu código. Outro ponto a destacar é que os erros, quando encontrados, são informados a linha e o lexema inválido encontrado. Isso é possível graças à variável **yylineo** oferecida pela ferramenta FLEX.

```
1  %%
2  {supress_errors}          { BEGIN(supress_errors_condition); supress_errors_flag = 1; }
3  "giveup"                  {return 0;}
4  "//"                      BEGIN(comment_condition);
5  {ign}                     {}
6
7  <comment_condition>.|\n    {}
8  <comment_condition>"\\\\"  { BEGIN(INITIAL); if(supress_errors_flag) BEGIN(supress_errors_condition); }
9
```

```
10  "\""                        BEGIN(word_condition);
11  <word_condition>{word_value}        {PRINT((PRINT_WORD PRINT_LEXEME, yylineno, yytext))}
12  <word_condition>"\""                { BEGIN(INITIAL);  if(supress_errors_flag) BEGIN(supress_errors_condition); }
13
14  {creditos}              {PRINT(("Feito por:\n%s\n",
15                          "Daniel Freitas Martins - 2304\n"
16                          "João Arthur Gonçalves do Vale - 3025\n"
17                          "Maria Dalila Vieira - 3030\n"
18                          "Naiara Cristiane dos Reis Diniz - 3005"))}
19
20  {type}                  {PRINT((PRINT_TYPE PRINT_LEXEME, yylineno, yytext))}
21  /* OUTRAS REGRAS (VERIFIQUE APÊNDICE A PARA VER AS DEMAIS) */
22  {identifier}                {PRINT((PRINT_IDENTIFIER PRINT_LEXEME, yylineno, yytext))}
23  <supress_errors_condition>.     {}
24  ";"+                        {} /* ignorando ponto e virgula */
25  .                           {PRINT((PRINT_ERROR "-> %s\nint_code_s0: %d\n", yylineno, yytext, yytext[0])) erro_encontrado
26  = 1;} /* Ignorar o que nao foi definido */
27  <<EOF>>                     {
28              if(erro_encontrado){
29                  PRINT((PRINT_ERROR_EOF))
30                  exit(1);
31              }
32              return 0;
33          }
34  %%
```

Figura 3.3 - Seção de Regras do arquivo lex.l.

Por fim, a Fig. 3.4 mostra a seção de códigos do arquivo do gerador de analisador léxico. Nela está presente a função para a remoção de espaços em branco, tabulações e quebras de linha. Com ela, números podem ser escritos considerando-se estes caracteres que são ignorados, permitindo uma maior flexibilidade na escrita destes valores. Por exemplo, o número real "**842.19**" poderia ser escrito no código como "**842 . 19**", sendo aceito também dessa maneira, sem os espaços em branco. Isso serve apenas para oferecer uma maior flexibilidade, apesar de que o código em si pode ter sua legibilidade afetada, ficando a critério do programador se valer disso ou não.

```
1   %%
2   void remover_espacos_e_print(int t){
3       char* s; /* tera a nova string sem os espacos em branco */
4       int i, j, tam_yytext = strlen(yytext);
5       s = (char*) malloc(tam_yytext*sizeof(char));
6       j = 0;
7       for(i = 0; i < tam_yytext; i++){
8           if(yytext[i] == ' ' || yytext[i] == '\n' || yytext[i] == '\t')
9               continue;
10          s[j++] = yytext[i];
11      }
12      s[j] = '\0';
```

```
13
14      switch(t){
15          case _REAL:
16              PRINT((PRINT_REAL_NUMBER PRINT_LEXEME, yylineno, s))
17          break;
18          case _NUMBER:
19              PRINT((PRINT_NUMBER PRINT_LEXEME, yylineno, s))
20          break;
21      }
22      free(s);
23  }
24  int yywrap(){ return 1; } /* se EOF for encontrado, encerre. */
25  int main(){
26      yylex();
27      return 0;
28  }
```

Figura 3.4 - Seção de Códigos do arquivo lex.l.

3.1 - Palavras-chave e Palavras reservadas

Em conjunto com a gramática apresentada e o analisador léxico criado, é possível observar quais são as palavras-chave e as palavras reservadas da linguagem Chameleon. Esta seção tem como objetivo sumarizar essas informações:

- **Palavras-chave**: *begin, end, squad, endsquad, vector, for, endfor, while, endwhile, if, endif, elif, endelif, task, endtask, jumpto, farewell, say, listen, stop.*
- **Palavras reservadas**: Todas as palavras-chave e as palavras *??creditos??, SUPRESS_ERRORS* e *giveup*. A palavra chave *giveup* serve apenas para forçar o encerramento do analisador léxico em um dado momento.

3.2 - Restrições

A especificação do analisador sintático e da gramática permite realizar inferências a respeito de algumas restrições da linguagem. Uma delas é a respeito da restrição dos nomes válidos para os identificadores, que não podem corresponder às palavras reservadas da linguagem, sumarizadas na Seção 3.1. Outras restrições dizem respeito à declaração de variáveis, atribuição de valores a variáveis, cadeias de caracteres, comentários e início e fim de comandos. Em resumo:

- Os nomes dos identificadores precisam começar com uma letra do alfabeto, maiúscula ou minúscula ou começar com *underline* seguido de uma letra do alfabeto da mesma forma. Após isso, letras ou dígitos podem compor o restante do lexema do identificador.
- Em relação a declaração e atribuição de valores a variáveis em uma mesma instrução, a linguagem não suporta esse tipo de operação. É necessário que se faça primeiro a declaração e, posteriormente em uma outra instrução, realize a operação de atribuição.
- As nossas cadeias de caracteres(*words*) são quaisquer conjuntos de caracteres que são identificados apenas se estiverem entre um abre aspas duplas e um fecha aspas duplas. Uma limitação da linguagem é a impossibilidade de se usar aspas duplas dentro de uma *word*. Futuramente poderemos incluir uma possibilidade de uso de caracteres de *escape* assim como ocorre na linguagem C com o caractere '\'.
- Os comentários são delimitados por // e \\, que indicam a abertura e o fechamento de um comentário, respectivamente.
- Os comandos se iniciam com dois pontos, e se encerram com a palavra end concatenada com o nome do comando.

Apesar da existência dessas restrições para a linguagem Chameleon, é importante ressaltar que, como abordado nas seções anteriores, o uso do pré-processador pode ser capaz de trocar ou até mesmo suprimir algumas dessas restrições. Porém, o código base, gerado através do pré-processamento, ainda seguirá estas restrições impostas pela linguagem.

4 - Programas, Testes e Resultados

Esta seção apresenta programas válidos e inválidos lexicalmente para a linguagem Chameleon. Junto de cada programa, a saída do analisador léxico é mostrada.

4.1 - Programas lexicalmente válidos

Abaixo serão mostrados códigos válidos para a linguagem Chameleon seguidos de suas respectivas saídas geradas pelo analisador léxico criado. Nos códigos abaixo é possível observar alguns dos recursos que a linguagem permite. Um deles é a concatenação de cadeias de caracteres pelo operador ++, como pode ser observado na Entrada 1. Ainda, nestes exemplos pode-se observar as estruturas de repetição e estruturas condicionais que seguem estruturas similares àquelas das linguagens C, C++ e Python.

Entrada 1 - Cálculo de Fibonacci

```
1   begin
2      integer N
3      N = 20
4      vector v N
5      integer f1
6      integer f2
7      f1 = 0
8      f2 = 1
9      integer i
10     integer temp
11     // calculando Fib(N) \\
12     for i = 0, i < N, i = i+1:
13        v[i] = f1
14        temp = fib2
15        fib2 = fib1 + fib2
16        fib = temp
17     endfor
18     say "O resultado de Fib(" ++ N ++ ") = " ++ fib2 ++ "\n"
19  end
```

```
Line 1: I've found a block begin. LEXEME: begin
Line 2: I've found a type. LEXEME: integer
Line 2: I've found an identifier. LEXEME: N
Line 3: I've found an identifier. LEXEME: N
Line 3: I've found an attribution. LEXEME: =
Line 4: I've found an integer number. LEXEME: 20
Line 4: I've found a vector declaration. LEXEME: vector
Line 4: I've found an identifier. LEXEME: v
Line 4: I've found an identifier. LEXEME: N
Line 5: I've found a type. LEXEME: integer
Line 5: I've found an identifier. LEXEME: f1
Line 6: I've found a type. LEXEME: integer
Line 6: I've found an identifier. LEXEME: f2
Line 7: I've found an identifier. LEXEME: f1
Line 7: I've found an attribution. LEXEME: =
Line 8: I've found an integer number. LEXEME: 0
```

```
Line 8: I've found an identifier. LEXEME: f2
Line 8: I've found an attribution. LEXEME: =
Line 9: I've found an integer number. LEXEME: 1
Line 9: I've found a type. LEXEME: integer
Line 9: I've found an identifier. LEXEME: i
Line 10: I've found a type. LEXEME: integer
Line 10: I've found an identifier. LEXEME: temp
Line 12: I've found a for. LEXEME: for
Line 12: I've found an identifier. LEXEME: i
Line 12: I've found an attribution. LEXEME: =
Line 12: I've found an integer number. LEXEME: 0
Line 12: I've found a comma. LEXEME: ,
Line 12: I've found an identifier. LEXEME: i
Line 12: I've found a rel operator. LEXEME: <
Line 12: I've found an identifier. LEXEME: N
Line 12: I've found a comma. LEXEME: ,
Line 12: I've found an identifier. LEXEME: i
Line 12: I've found an attribution. LEXEME: =
Line 12: I've found an identifier. LEXEME: i
Line 12: I've found an add operator. LEXEME: +
Line 12: I've found an integer number. LEXEME: 1
Line 12: I've found a separator. LEXEME: :
Line 13: I've found an identifier. LEXEME: v
Line 13: I've found a vector access start. LEXEME: [
Line 13: I've found an identifier. LEXEME: i
Line 13: I've found a vector access end. LEXEME: ]
Line 13: I've found an attribution. LEXEME: =
Line 13: I've found an identifier. LEXEME: f1
Line 14: I've found an identifier. LEXEME: temp
Line 14: I've found an attribution. LEXEME: =
Line 14: I've found an identifier. LEXEME: fib2
Line 15: I've found an identifier. LEXEME: fib2
Line 15: I've found an attribution. LEXEME: =
Line 15: I've found an identifier. LEXEME: fib1
Line 15: I've found an add operator. LEXEME: +
Line 15: I've found an identifier. LEXEME: fib2
Line 16: I've found an identifier. LEXEME: fib
Line 16: I've found an attribution. LEXEME: =
Line 16: I've found an identifier. LEXEME: temp
Line 17: I've found a for end. LEXEME: endfor
Line 18: I've found a say. LEXEME: say
Line 18: I've found a word. LEXEME: O resultado de Fib(
Line 18: I've found a word concatenation operator. LEXEME: ++
Line 18: I've found an identifier. LEXEME: N
Line 18: I've found a word concatenation operator. LEXEME: ++
Line 18: I've found a word. LEXEME: ) =
Line 18: I've found a word concatenation operator. LEXEME: ++
Line 18: I've found an identifier. LEXEME: fib2
Line 18: I've found a word concatenation operator. LEXEME: ++
Line 18: I've found a word. LEXEME: \n
Line 19: I've found a block end. LEXEME: end
```

Entrada 2 - Uso de *jumpto*

```
1   // programa que incrementa uma variavel ate 5 usando jumpto \\
2   begin
3       integer i
4       i = 0
5
6       RETORNAR: // label de retorno \\
7       i = i + 1
8       if i < 5:
9           jumpto RETORNAR
10
11      say i // deve imprimir 5 \\
12  end
```

```
Line 2: I've found a block begin. LEXEME: begin
Line 3: I've found a type. LEXEME: integer
Line 3: I've found an identifier. LEXEME: i
Line 4: I've found an identifier. LEXEME: i
Line 4: I've found an attribution. LEXEME: =
Line 6: I've found an integer number. LEXEME: 0
Line 6: I've found an identifier. LEXEME: RETORNAR
Line 6: I've found a separator. LEXEME: :
Line 7: I've found an identifier. LEXEME: i
Line 7: I've found an attribution. LEXEME: =
Line 7: I've found an identifier. LEXEME: i
Line 7: I've found an add operator. LEXEME: +
Line 8: I've found an integer number. LEXEME: 1
Line 8: I've found an if. LEXEME: if
Line 8: I've found an identifier. LEXEME: i
Line 8: I've found a rel operator. LEXEME: <
Line 8: I've found an integer number. LEXEME: 5
Line 8: I've found a separator. LEXEME: :
Line 9: I've found a jumpto. LEXEME: jumpto
Line 9: I've found an identifier. LEXEME: RETORNAR
Line 11: I've found a say. LEXEME: say
Line 11: I've found an identifier. LEXEME: i
Line 12: I've found a block end. LEXEME: end
```

Entrada 3 - Operações relacionais

```
1   begin
2       integer a
3       integer b
4
5       listen a
6       listen b
7       if(((a == b) or (a/b < 1) or (b > a/2)) and
8           ((b^a != 1) or (a >= b*2) or (a <= b/2) or !(a < b))):
```

```
 9        say "Ok"
10     elif:
11        say "Ok tambem"
12     endelif
13  end
```

```
Line 1: I've found a block begin. LEXEME: begin
Line 2: I've found a type. LEXEME: integer
Line 2: I've found an identifier. LEXEME: a
Line 3: I've found a type. LEXEME: integer
Line 3: I've found an identifier. LEXEME: b
Line 5: I've found a listen. LEXEME: listen
Line 5: I've found an identifier. LEXEME: a
Line 6: I've found a listen. LEXEME: listen
Line 6: I've found an identifier. LEXEME: b
Line 7: I've found an if. LEXEME: if
Line 7: I've found an open parenthesis. LEXEME: (
Line 7: I've found an open parenthesis. LEXEME: (
Line 7: I've found an open parenthesis. LEXEME: (
Line 7: I've found an identifier. LEXEME: a
Line 7: I've found a rel operator. LEXEME: ==
Line 7: I've found an identifier. LEXEME: b
Line 7: I've found a close parenthesis. LEXEME: )
Line 7: I've found a logic operator. LEXEME: or
Line 7: I've found an open parenthesis. LEXEME: (
Line 7: I've found an identifier. LEXEME: a
Line 7: I've found a div operator. LEXEME: /
Line 7: I've found an identifier. LEXEME: b
Line 7: I've found a rel operator. LEXEME: <
Line 7: I've found an integer number. LEXEME: 1
Line 7: I've found a close parenthesis. LEXEME: )
Line 7: I've found a logic operator. LEXEME: or
Line 7: I've found an open parenthesis. LEXEME: (
Line 7: I've found an identifier. LEXEME: b
Line 7: I've found a rel operator. LEXEME: >
Line 7: I've found an identifier. LEXEME: a
Line 7: I've found a div operator. LEXEME: /
Line 7: I've found an integer number. LEXEME: 2
Line 7: I've found a close parenthesis. LEXEME: )
Line 7: I've found a close parenthesis. LEXEME: )
Line 7: I've found a logic operator. LEXEME: and
Line 8: I've found an open parenthesis. LEXEME: (
Line 8: I've found an open parenthesis. LEXEME: (
Line 8: I've found an identifier. LEXEME: b
Line 8: I've found a pow operator. LEXEME: ^
Line 8: I've found an identifier. LEXEME: a
Line 8: I've found a rel operator. LEXEME: !=
Line 8: I've found an integer number. LEXEME: 1
Line 8: I've found a close parenthesis. LEXEME: )
Line 8: I've found a logic operator. LEXEME: or
Line 8: I've found an open parenthesis. LEXEME: (
Line 8: I've found an identifier. LEXEME: a
Line 8: I've found a rel operator. LEXEME: >=
Line 8: I've found an identifier. LEXEME: b
Line 8: I've found a div operator. LEXEME: *
```

```
Line 8: I've found an integer number. LEXEME: 2
Line 8: I've found a close parenthesis. LEXEME: )
Line 8: I've found a logic operator. LEXEME: or
Line 8: I've found an open parenthesis. LEXEME: (
Line 8: I've found an identifier. LEXEME: a
Line 8: I've found a rel operator. LEXEME: <=
Line 8: I've found an identifier. LEXEME: b
Line 8: I've found a div operator. LEXEME: /
Line 8: I've found an integer number. LEXEME: 2
Line 8: I've found a close parenthesis. LEXEME: )
Line 8: I've found a logic operator. LEXEME: or
Line 8: I've found a logic operator. LEXEME: !
Line 8: I've found an open parenthesis. LEXEME: (
Line 8: I've found an identifier. LEXEME: a
Line 8: I've found a rel operator. LEXEME: <
Line 8: I've found an identifier. LEXEME: b
Line 8: I've found a close parenthesis. LEXEME: )
Line 8: I've found a close parenthesis. LEXEME: )
Line 8: I've found a close parenthesis. LEXEME: )
Line 8: I've found a separator. LEXEME: :
Line 9: I've found a say. LEXEME: say
Line 9: I've found a word. LEXEME: Ok
Line 10: I've found an elif. LEXEME: elif
Line 10: I've found a separator. LEXEME: :
Line 11: I've found a say. LEXEME: say
Line 11: I've found a word. LEXEME: Ok tambem
Line 12: I've found a endelif end. LEXEME: endelif
Line 13: I've found a block end. LEXEME: end
```

Entrada 4 - Uso de *squad* e *stop*

```
1  begin
2      squad pessoa :
3          string nome
4          integer idade
5      endsquad
6
7      listen pessoa->nome
8      listen pessoa->idade
9      say "Seu nome: " ++ pessoa->nome ++ "\n"
10     say "Sua idade: " ++ pessoa->idade ++ "\n"
11
12     while(1 == 1):
13         stop // quebrando o loop: exemplo de uso de stop \\
14     endwhile
15 end
```

```
Line 1: I've found a block begin. LEXEME: begin
Line 2: I've found a squad declaration. LEXEME: squad
Line 2: I've found an identifier. LEXEME: pessoa
```

```
Line 2: I've found a separator. LEXEME: :
Line 3: I've found an identifier. LEXEME: string
Line 3: I've found an identifier. LEXEME: nome
Line 4: I've found a type. LEXEME: integer
Line 4: I've found an identifier. LEXEME: idade
Line 5: I've found a squad end. LEXEME: endsquad
Line 7: I've found a listen. LEXEME: listen
Line 7: I've found an identifier. LEXEME: pessoa
Line 7: I've found a squad access derreference. LEXEME: ->
Line 7: I've found an identifier. LEXEME: nome
Line 8: I've found a listen. LEXEME: listen
Line 8: I've found an identifier. LEXEME: pessoa
Line 8: I've found a squad access derreference. LEXEME: ->
Line 8: I've found an identifier. LEXEME: idade
Line 9: I've found a say. LEXEME: say
Line 9: I've found a word. LEXEME: Seu nome:
Line 9: I've found a word concatenation operator. LEXEME: ++
Line 9: I've found an identifier. LEXEME: pessoa
Line 9: I've found a squad access derreference. LEXEME: ->
Line 9: I've found an identifier. LEXEME: nome
Line 9: I've found a word concatenation operator. LEXEME: ++
Line 9: I've found a word. LEXEME: \n
Line 10: I've found a say. LEXEME: say
Line 10: I've found a word. LEXEME: Sua idade:
Line 10: I've found a word concatenation operator. LEXEME: ++
Line 10: I've found an identifier. LEXEME: pessoa
Line 10: I've found a squad access derreference. LEXEME: ->
Line 10: I've found an identifier. LEXEME: idade
Line 10: I've found a word concatenation operator. LEXEME: ++
Line 10: I've found a word. LEXEME: \n
Line 12: I've found a while. LEXEME: while
Line 12: I've found an open parenthesis. LEXEME: (
Line 12: I've found an integer number. LEXEME: 1
Line 12: I've found a rel operator. LEXEME: ==
Line 12: I've found an integer number. LEXEME: 1
Line 12: I've found a close parenthesis. LEXEME: )
Line 12: I've found a separator. LEXEME: :
Line 13: I've found a stop. LEXEME: stop
Line 14: I've found a while end. LEXEME: endwhile
Line 15: I've found a block end. LEXEME: end
```

Entrada 5 - Função com retorno (*task* com *farewell*) e algumas operações

```
1  begin
2      integer a
3      real b
4      real c
5      real resultado
6
7      listen a
8      listen b
```

```
 9      c = (a - b)*5
10
11      task funcaoSomaNumeros n1, n2, n3:
12          farewell n1 + n2 + n3
13      endtask
14
15      resultado = task funcaoSomaNumeros(a, b, c)
16      say "O resultado de " ++ a ++ " + " ++ b ++ " + " ++ c ++ " = " ++ resultado
17  end
```

```
Line 1: I've found a block begin. LEXEME: begin
Line 2: I've found a type. LEXEME: integer
Line 2: I've found an identifier. LEXEME: a
Line 3: I've found a type. LEXEME: real
Line 3: I've found an identifier. LEXEME: b
Line 4: I've found a type. LEXEME: real
Line 4: I've found an identifier. LEXEME: c
Line 5: I've found a type. LEXEME: real
Line 5: I've found an identifier. LEXEME: resultado
Line 7: I've found a listen. LEXEME: listen
Line 7: I've found an identifier. LEXEME: a
Line 8: I've found a listen. LEXEME: listen
Line 8: I've found an identifier. LEXEME: b
Line 9: I've found an identifier. LEXEME: c
Line 9: I've found an attribution. LEXEME: =
Line 9: I've found an open parenthesis. LEXEME: (
Line 9: I've found an identifier. LEXEME: a
Line 9: I've found an add operator. LEXEME: -
Line 9: I've found an identifier. LEXEME: b
Line 9: I've found a close parenthesis. LEXEME: )
Line 9: I've found a div operator. LEXEME: *
Line 11: I've found an integer number. LEXEME: 5
Line 11: I've found a task. LEXEME: task
Line 11: I've found an identifier. LEXEME: funcaoSomaNumeros
Line 11: I've found an identifier. LEXEME: n1
Line 11: I've found a comma. LEXEME: ,
Line 11: I've found an identifier. LEXEME: n2
Line 11: I've found a comma. LEXEME: ,
Line 11: I've found an identifier. LEXEME: n3
Line 11: I've found a separator. LEXEME: :
Line 12: I've found a farewell. LEXEME: farewell
Line 12: I've found an identifier. LEXEME: n1
Line 12: I've found an add operator. LEXEME: +
Line 12: I've found an identifier. LEXEME: n2
Line 12: I've found an add operator. LEXEME: +
Line 12: I've found an identifier. LEXEME: n3
Line 13: I've found a task end. LEXEME: endtask
Line 15: I've found an identifier. LEXEME: resultado
Line 15: I've found an attribution. LEXEME: =
Line 15: I've found a task. LEXEME: task
Line 15: I've found an identifier. LEXEME: funcaoSomaNumeros
Line 15: I've found an open parenthesis. LEXEME: (
Line 15: I've found an identifier. LEXEME: a
Line 15: I've found a comma. LEXEME: ,
Line 15: I've found an identifier. LEXEME: b
```

```
Line 15: I've found a comma. LEXEME: ,
Line 15: I've found an identifier. LEXEME: c
Line 15: I've found a close parenthesis. LEXEME: )
Line 16: I've found a say. LEXEME: say
Line 16: I've found a word. LEXEME: O resultado de
Line 16: I've found a word concatenation operator. LEXEME: ++
Line 16: I've found an identifier. LEXEME: a
Line 16: I've found a word concatenation operator. LEXEME: ++
Line 16: I've found a word. LEXEME:  +
Line 16: I've found a word concatenation operator. LEXEME: ++
Line 16: I've found an identifier. LEXEME: b
Line 16: I've found a word concatenation operator. LEXEME: ++
Line 16: I've found a word. LEXEME:  +
Line 16: I've found a word concatenation operator. LEXEME: ++
Line 16: I've found an identifier. LEXEME: c
Line 16: I've found a word concatenation operator. LEXEME: ++
Line 16: I've found a word. LEXEME:  =
Line 16: I've found a word concatenation operator. LEXEME: ++
Line 16: I've found an identifier. LEXEME: resultado
Line 17: I've found a block end. LEXEME: end
```

Entrada 6 - Código da Fig. 2.3.2 - Convertido a partir da macro PCD

```
1  begin
2  // Codificacao for ajudar algumas clasifs de PCD/iniciantes \\
3  task p:
4  farewell 0
5  endtask
6  squad pessoa :
7  word nome
8  integer idade
9  endsquad
10 integer num
11 num = Falso
12 integer valor
13 valor = 5
14 vector vet 5
15 word txt
16 txt = "aprendendo a programar"
17
18 if valor < 3:
19 num = True
20 elif valor < 2 e valor != 0:
21 num = 5 / 2!
22 endelif
23
24 while valor >= 0:
25 valor = valor - 1
26 if valor < 2:
```

```
27  stop
28  endif
29  endwhile
30
31  for valor = 0, valor > 5, valor = valor + 1:
32  num = num ^ 2
33  vet[4] = 5 * 3 % dois
34  endfor
35  task valor
36  end
```

```
Line 1: I've found a block begin. LEXEME: begin
Line 3: I've found a task. LEXEME: task
Line 3: I've found an identifier. LEXEME: p
Line 3: I've found a separator. LEXEME: :
Line 4: I've found a farewell. LEXEME: farewell
Line 5: I've found an integer number. LEXEME: 0
Line 5: I've found a task end. LEXEME: endtask
Line 6: I've found a squad declaration. LEXEME: squad
Line 6: I've found an identifier. LEXEME: pessoa
Line 6: I've found a separator. LEXEME: :
Line 7: I've found a type. LEXEME: word
Line 7: I've found an identifier. LEXEME: nome
Line 8: I've found a type. LEXEME: integer
Line 8: I've found an identifier. LEXEME: idade
Line 9: I've found a squad end. LEXEME: endsquad
Line 10: I've found a type. LEXEME: integer
Line 10: I've found an identifier. LEXEME: num
Line 11: I've found an identifier. LEXEME: num
Line 11: I've found an attribution. LEXEME: =
Line 11: I've found an identifier. LEXEME: Falso
Line 12: I've found a type. LEXEME: integer
Line 12: I've found an identifier. LEXEME: valor
Line 13: I've found an identifier. LEXEME: valor
Line 13: I've found an attribution. LEXEME: =
Line 14: I've found an integer number. LEXEME: 5
Line 14: I've found a vector declaration. LEXEME: vector
Line 14: I've found an identifier. LEXEME: vet
Line 15: I've found an integer number. LEXEME: 5
Line 15: I've found a type. LEXEME: word
Line 15: I've found an identifier. LEXEME: txt
Line 16: I've found an identifier. LEXEME: txt
Line 16: I've found an attribution. LEXEME: =
Line 16: I've found a word. LEXEME: aprendendo a programar
Line 18: I've found an if. LEXEME: if
Line 18: I've found an identifier. LEXEME: valor
Line 18: I've found a rel operator. LEXEME: <
Line 18: I've found an integer number. LEXEME: 3
Line 18: I've found a separator. LEXEME: :
Line 19: I've found an identifier. LEXEME: num
Line 19: I've found an attribution. LEXEME: =
Line 19: I've found an identifier. LEXEME: True
Line 20: I've found an elif. LEXEME: elif
Line 20: I've found an identifier. LEXEME: valor
```

```
Line 20: I've found a rel operator. LEXEME: <
Line 20: I've found an integer number. LEXEME: 2
Line 20: I've found an identifier. LEXEME: e
Line 20: I've found an identifier. LEXEME: valor
Line 20: I've found a rel operator. LEXEME: !=
Line 20: I've found an integer number. LEXEME: 0
Line 20: I've found a separator. LEXEME: :
Line 21: I've found an identifier. LEXEME: num
Line 21: I've found an attribution. LEXEME: =
Line 21: I've found an integer number. LEXEME: 5
Line 21: I've found a div operator. LEXEME: /
Line 21: I've found an integer number. LEXEME: 2
Line 21: I've found a logic operator. LEXEME: !
Line 22: I've found a endelif end. LEXEME: endelif
Line 24: I've found a while. LEXEME: while
Line 24: I've found an identifier. LEXEME: valor
Line 24: I've found a rel operator. LEXEME: >=
Line 24: I've found an integer number. LEXEME: 0
Line 24: I've found a separator. LEXEME: :
Line 25: I've found an identifier. LEXEME: valor
Line 25: I've found an attribution. LEXEME: =
Line 25: I've found an identifier. LEXEME: valor
Line 25: I've found an add operator. LEXEME: -
Line 26: I've found an integer number. LEXEME: 1
Line 26: I've found an if. LEXEME: if
Line 26: I've found an identifier. LEXEME: valor
Line 26: I've found a rel operator. LEXEME: <
Line 26: I've found an integer number. LEXEME: 2
Line 26: I've found a separator. LEXEME: :
Line 27: I've found a stop. LEXEME: stop
Line 28: I've found an if end. LEXEME: endif
Line 29: I've found a while end. LEXEME: endwhile
Line 31: I've found a for. LEXEME: for
Line 31: I've found an identifier. LEXEME: valor
Line 31: I've found an attribution. LEXEME: =
Line 31: I've found an integer number. LEXEME: 0
Line 31: I've found a comma. LEXEME: ,
Line 31: I've found an identifier. LEXEME: valor
Line 31: I've found a rel operator. LEXEME: >
Line 31: I've found an integer number. LEXEME: 5
Line 31: I've found a comma. LEXEME: ,
Line 31: I've found an identifier. LEXEME: valor
Line 31: I've found an attribution. LEXEME: =
Line 31: I've found an identifier. LEXEME: valor
Line 31: I've found an add operator. LEXEME: +
Line 31: I've found an integer number. LEXEME: 1
Line 31: I've found a separator. LEXEME: :
Line 32: I've found an identifier. LEXEME: num
Line 32: I've found an attribution. LEXEME: =
Line 32: I've found an identifier. LEXEME: num
Line 32: I've found a pow operator. LEXEME: ^
Line 33: I've found an integer number. LEXEME: 2
Line 33: I've found an identifier. LEXEME: vet
Line 33: I've found a vector access start. LEXEME: [
Line 33: I've found an integer number. LEXEME: 4
Line 33: I've found a vector access end. LEXEME: ]
```

```
Line 33: I've found an attribution. LEXEME: =
Line 33: I've found an integer number. LEXEME: 5
Line 33: I've found a div operator. LEXEME: *
Line 33: I've found an integer number. LEXEME: 3
Line 33: I've found a div operator. LEXEME: %
Line 33: I've found an identifier. LEXEME: dois
Line 34: I've found a for end. LEXEME: endfor
Line 35: I've found a task. LEXEME: task
Line 35: I've found an identifier. LEXEME: valor
Line 36: I've found a block end. LEXEME: end
```

Entrada 7 - Código da Fig. 2.3.4 - Convertido a partir da macro Python

```
1  begin
2  // imprime se os numeros sao pares ou impares, de 1 a 50 \\
3  integer N
4  N = 50
5
6  task verificarParidade numero:
7  if (numero%2 == 0):
8  say numero + " eh par"
9  elif :
10 say numero + " eh impar"
11 endelif
12 endtask
13
14 while N <= 50:
15 task verificarParidade(N)
16
17 N = N + 1
18 endtask
19 end
```

```
Line 1: I've found a block begin. LEXEME: begin
Line 3: I've found a type. LEXEME: integer
Line 3: I've found an identifier. LEXEME: N
Line 4: I've found an identifier. LEXEME: N
Line 4: I've found an attribution. LEXEME: =
Line 6: I've found an integer number. LEXEME: 50
Line 6: I've found a task. LEXEME: task
Line 6: I've found an identifier. LEXEME: verificarParidade
Line 6: I've found an identifier. LEXEME: numero
Line 6: I've found a separator. LEXEME: :
Line 7: I've found an if. LEXEME: if
Line 7: I've found an open parenthesis. LEXEME: (
Line 7: I've found an identifier. LEXEME: numero
Line 7: I've found a div operator. LEXEME: %
Line 7: I've found an integer number. LEXEME: 2
Line 7: I've found a rel operator. LEXEME: ==
Line 7: I've found an integer number. LEXEME: 0
```

```
Line 7: I've found a close parenthesis. LEXEME: )
Line 7: I've found a separator. LEXEME: :
Line 8: I've found a say. LEXEME: say
Line 8: I've found an identifier. LEXEME: numero
Line 8: I've found an add operator. LEXEME: +
Line 8: I've found a word. LEXEME:  eh par
Line 9: I've found an elif. LEXEME: elif
Line 9: I've found a separator. LEXEME: :
Line 10: I've found a say. LEXEME: say
Line 10: I've found an identifier. LEXEME: numero
Line 10: I've found an add operator. LEXEME: +
Line 10: I've found a word. LEXEME:  eh impar
Line 11: I've found a endelif end. LEXEME: endelif
Line 12: I've found a task end. LEXEME: endtask
Line 14: I've found a while. LEXEME: while
Line 14: I've found an identifier. LEXEME: N
Line 14: I've found a rel operator. LEXEME: <=
Line 14: I've found an integer number. LEXEME: 50
Line 14: I've found a separator. LEXEME: :
Line 15: I've found a task. LEXEME: task
Line 15: I've found an identifier. LEXEME: verificarParidade
Line 15: I've found an open parenthesis. LEXEME: (
Line 15: I've found an identifier. LEXEME: N
Line 15: I've found a close parenthesis. LEXEME: )
Line 17: I've found an identifier. LEXEME: N
Line 17: I've found an attribution. LEXEME: =
Line 17: I've found an identifier. LEXEME: N
Line 17: I've found an add operator. LEXEME: +
Line 18: I've found an integer number. LEXEME: 1
Line 18: I've found a task end. LEXEME: endtask
Line 19: I've found a block end. LEXEME: end
```

Os testes acima ilustram alguns exemplos de programas válidos da linguagem Chameleon. Tais exemplos tentam mostrar o uso da grande maioria dos comandos disponíveis na linguagem. Além disso, o analisador léxico conseguiu reconhecer todos os tokens válidos, conforme as impressões realizadas para cada entrada.

Note que na Entrada 7, o código é lexicalmente correto, mas sintaticamente incorreto, uma vez que a concatenação de cadeias de caracteres é dado pelo operador ++, e foi utilizado o operador + sem convertê-lo para ++.

4.2 - Programas lexicalmente inválidos

Abaixo serão mostrados códigos lexicalmente inválidos para a linguagem Chameleon seguidos de suas respectivas saídas geradas pelo analisador léxico criado.

No exemplo de entrada Entrada E1 a seguir é mostrado um código que tenta usar nomes para identificadores começados com o caractere **$**. O analisador léxico identifica todos os erros e por fim encerra, informando que o código está com erro léxico.

Em contrapartida, a Entrada E2 corresponde ao mesmo código de Entrada E1, mas com o uso de SUPRESS_ERRORS. Todos os *tokens* possíveis de serem identificados são considerados válidos, e caracteres inválidos são descartados. Com isso, o código estaria apto a passar para a próxima etapa, mesmo com erros léxicos, descartados durante o processo de análise léxica. Com este exemplo, é possível notar que a supressão de erros é um recurso interessante para este trabalho e para esta linguagem que tem como um dos objetivos oferecer flexibilidades ao programador.

Entrada E1 - Soma de dois números - Erro léxico

```
1   begin
2      real g
3      g = 5.3
4      integer $a
5      real $b
6
7      $a = 2
8      listen $b;
9      $b = $a + $b; // soma 2 ao numero lido \\
10     say $b;
11  end
```

```
Line 1: I've found a block begin. LEXEME: begin
Line 2: I've found a type. LEXEME: real
Line 2: I've found an identifier. LEXEME: g
Line 3: I've found an identifier. LEXEME: g
Line 3: I've found an attribution. LEXEME: =
Line 4: I've found a real number. LEXEME: 5.3
Line 4: I've found a type. LEXEME: integer
---------- Erro encontrado na linha 4 ----------
-> $
int_code_s0: 36
Line 4: I've found an identifier. LEXEME: a
Line 5: I've found a type. LEXEME: real
---------- Erro encontrado na linha 5 ----------
-> $
int_code_s0: 36
Line 5: I've found an identifier. LEXEME: b
---------- Erro encontrado na linha 7 ----------
-> $
int_code_s0: 36
Line 7: I've found an identifier. LEXEME: a
```

```
Line 7: I've found an attribution. LEXEME: =
Line 8: I've found an integer number. LEXEME: 2
Line 8: I've found a listen. LEXEME: listen
---------- Erro encontrado na linha 8 ----------
-> $
int_code_s0: 36
Line 8: I've found an identifier. LEXEME: b
---------- Erro encontrado na linha 9 ----------
-> $
int_code_s0: 36
Line 9: I've found an identifier. LEXEME: b
Line 9: I've found an attribution. LEXEME: =
---------- Erro encontrado na linha 9 ----------
-> $
int_code_s0: 36
Line 9: I've found an identifier. LEXEME: a
Line 9: I've found an add operator. LEXEME: +
---------- Erro encontrado na linha 9 ----------
-> $
int_code_s0: 36
Line 9: I've found an identifier. LEXEME: b
Line 10: I've found a say. LEXEME: say
---------- Erro encontrado na linha 10 ----------
-> $
int_code_s0: 36
Line 10: I've found an identifier. LEXEME: b
Line 11: I've found a block end. LEXEME: end
-> Um ou mais erros foram encontrados. Corrija-os!
```

Entrada E2 - Soma de dois números - Erro léxico - Supressão de erros

```
1  SUPRESS_ERRORS
2  begin
3     integer $a
4     real $b
5
6     $a = 2
7     listen $b
8     $b = $a + $b // soma 2 ao numero lido \\
9     say $b
10 end
```

```
Line 2: I've found a block begin. LEXEME: begin
Line 3: I've found a type. LEXEME: integer
Line 3: I've found an identifier. LEXEME: a
Line 4: I've found a type. LEXEME: real
Line 4: I've found an identifier. LEXEME: b
Line 6: I've found an identifier. LEXEME: a
Line 6: I've found an attribution. LEXEME: =
Line 7: I've found an integer number. LEXEME: 2
Line 7: I've found a listen. LEXEME: listen
Line 7: I've found an identifier. LEXEME: b
```

```
Line 8: I've found an identifier. LEXEME: b
Line 8: I've found an attribution. LEXEME: =
Line 8: I've found an identifier. LEXEME: a
Line 8: I've found an add operator. LEXEME: +
Line 8: I've found an identifier. LEXEME: b
Line 9: I've found a say. LEXEME: say
Line 9: I've found an identifier. LEXEME: b
Line 10: I've found a block end. LEXEME: end
```

5 - Testes no JFlap

Com o objetivo de validar a gramática da linguagem criada e descrita anteriormente, foi utilizada a ferramenta JFlap (versão beta 8.0). Esta ferramenta contém uma opção para validar a gramática denominada *Grammar,* por meio dela é possível inserir as produções e verificar se as entradas corretas estão sendo aceitas. Outrossim, o programa em si realiza a construção de uma árvore de derivação considerando, por padrão, a derivação mais à esquerda. Vale ressaltar, que os testes realizados dentro da ferramenta JFLAP, foram realizados a partir de uma versão preliminar da gramática, mas capaz de representar o comportamento da mesma.

Por conseguinte, uma vez que as produções possuem exclusivamente o formato: letra maiúscula deriva em uma ou mais variáveis ou terminais (ex.: A→ B, A→ 1, A→ b, A→ B C), foi necessário refatorar os nomes das produções já criadas, para um que se adequasse ao formato aceito pela ferramenta.

Ademais, os testes foram realizados nas produções mais relevantes e mais propensas a erros, que são: *expressions* e *commands*. Abaixo, nas Tabelas 5.1 e 5.2 seguem as conversões dos nomes utilizados para os testes.

Produções *Expressions*	
Produção original	**Produção correspondente**
expression → *add_expression* | *div_expression* | *pow_expression* | *logic_expression* | **(***expression***)**	*E* → *A* | *V* | *P* | *L* | **(***E***)**

| number | real_number	| N | R
add_expression → **-** *expression* |**+** *expression* |*expression add_operator expression*	*A* → -E | +E | E + E
div_expression → *expression div_operator expression*	V → E T E
pow_expression → *expression pow_operator expression*	*P* → E Z E
logic_expression → *expression logic_operator expression* | *neg_operator expression*	*L* → E O E | G E
number → digit | digit number	*N* → D | D N
real_number → number**.**number	R → N.N
add_operator → + | -	X → + | -
div_operator → / | ***</br>| **%**	T → / | ***</br>| **%**
pow_operator → ^	Z → ^
logic_operator → **and** | **or** **(&** | **)**	O → & ||
neg_operator → **!**	G → !
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**	*D* → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Tabela 5.1 - Conversões para as produções de *expressions.*

Produções *Commands*	
Produções originais	**Produções equivalentes**
command → *variable attribution expression*	C → V A E

| *if_command* | *for_command* | *while_command* | *farewell_command* | *stop_command* | *jumpto_command* | *say_command* | *listen_command* | *task_command* | *label*	| I | F | *W* | *R* | *S* | *J* | *Y* | *L* | *T* | *B*
variable → *identifier* | *vector_access* | *squad_access*	*V* → *D* | *H* | *Q*
identifier → *letter identifier_complement* | *underline letter identifier_complement*	*D* → *O N* |*U O N*
letter → **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**	*O* → **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**
identifier_complement → ε | *digit identifier_complement* | *letter identifier_complement* | *underline identifier_complement*	*N* → ε | *E N* | *O N* | *U N*
underline → _	*U* → _
vector_access → *identifier***[***number***]**	H→ D[E]
squad_access → *identifier* **->** identifier	*Q* → *D*-> *D*
attribution → =	*A* → =
E → … todo o corpo de expression está descrito na tabela 3.1, verifique observações após esta tabela.	E → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
if_command → **if** *expression* **:** *statement* **endif** | **if** *expression* **:** *statement* **elif** *statement* **endelif**	I → Z E : X Z | Z E : X Z X Z
Todas as palavras reservadas derivam em 0, como: IF, ENDIF, FOR etc.	Z→ 0

statement → **ε** | *command statement*	X→ **ε** | *C X*
for_command → **for** *expression*, *expression*, *expression*: *statement* **endfor**	F → Z E, E, E: X Z
while_command → **while** *expression* : *statement* **endwhile**	W → Z E: X Z
farewell_command → **farewell**	R → Z
stop_command → **stop**	S → Z
jumpto_command → **jumpto** *identifier*	*J* → *Z D*
say_command → **say** *expression*	*Y* → *Z E*
listen_command → **listen** *variable*	L → *Z V*
task_command → **task** *identifier task_parameters* : *expression* **taskend**	*T* → *Z D K: E Z*
task_parameters → **ε** | *task_parameter*	*K* → **ε** | *G*
task_parameter → *expression* | *expression task_parameter*	G → E | E G
label → *identifier* : *command*	*B* → *D : C*

Tabela 5.2 - Conversões para as produções de *commands.*

Observações: após construirmos a tabela e refatorar as produções foi possível observar três limitações no uso da ferramenta que estão listadas abaixo.

- Devido a produção *expressions* ser bem extensa, ela teve de ser representada como apenas números naturais na Tabela 5.2, visto que as letras do alfabeto haviam se esgotado;
- As letras maiúsculas não foram representadas visto que a ferramenta entende que são expressões e não terminais;

- Todas as palavras reservadas foram representadas pelo número 0 visto que não seria possível representar todas distintamente (novamente pela limitação de quantidade de letras disponíveis).

5.1 - Testes para as derivações de *expressions*

Foram realizados testes a fim de validar a gramática das derivações de *expressions*. Para isso inseriu-se no JFlap as produções da Tabela 5.1 e, em seguida, testou-se com entradas diversas descritas abaixo (cada tópico refere-se a uma derivação em específico).

- ***add_expression***

É notório que somas simples são válidas como: 1+1, 10+5 etc. Para comprovar a Fig. 5.1.1 mostra o resultado das derivações para a entrada 2+2. Também é possível realizar subtrações como 1-1, bem como é mostrado na Fig. 5.1.2.

Derivation Tree | Derivation Table

Production	Derivation
	E
E->N	N
N->D N	D N
D->2	2 N
N->+ N	2 + N
N->D	2 + D
D->2	2 + 2

Figura 5.1.1 - Entrada 2+2.

Production	Derivation
	E
E->A	A
A->E X E	E X E
E->N	N X E
N->D	D X E
D->1	1 X E
X->-	1 - E
E->N	1 - N
N->D	1 - D
D->1	1 - 1

Figura 5.1.2 - Entrada 1-1.

A Fig. 5.1.3 mostra que é possível realizar a subtração por meio da adição de números negativos, como: 1+-1.

Production	Derivation
	E
E->A	A
A->E + E	E + E
E->N	N + E
N->D	D + E
D->1	1 + E
E->A	1 + A
A->- E	1 + - E
E->N	1 + - N
N->D	1 + - D
D->1	1 + - 1

Figura 5.1.3 - Entrada 1+-1.

Ou ainda, utilizar 1+(-1), todavia o uso de parênteses demanda um maior esforço computacional, como mostrado na Fig. 5.1.4a.

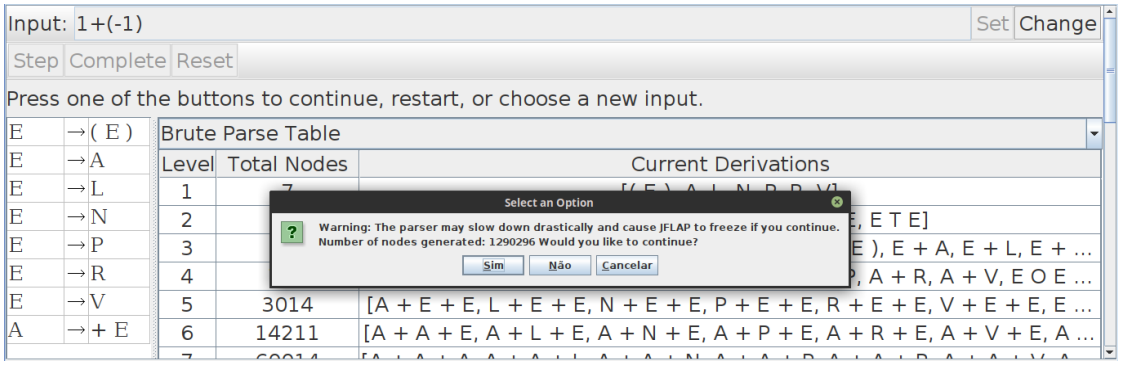

Figura 5.1.4a - Esforço computacional para a entrada 1+(-1)

Mas é possível realizar essa computação, como observado na Fig. 5.1.4b.

.

Production	Derivation
	E
E->A	A
A->E + E	E + E
E->N	N + E
N->D	D + E
D->1	1 + E
E->(E)	1 + (E)
E->A	1 + (A)
A->- E	1 + (- E)
E->N	1 + (- N)
N->D	1 + (- D)
D->1	1 + (- 1)

Figura 5.1.4b - Entrada 1+(-1)

Uma observação feita pelo grupo é que números de 5 dígitos demoram muito para serem computados, pois demanda processamento da máquina. Um teste com a entrada 65445+2 foi realizada e este demorou mais de 10 minutos de espera sem resposta, onde 3 milhões de nós foram criados para representar a árvore de derivações possíveis e por isso não finalizou.

Expressões algébricas com soma e subtração também são válidas, como 3+5+-7, como observado na Fig. 5.1.5. Entretanto, somas maiores que esta representação também demandam muito esforço computacional.

Production	Derivation
	E
E->A	A
A->E + E	E + E
E->A	A + E
A->E + E	E + E + E
E->N	N + E + E
N->D	D + E + E
D->3	3 + E + E
E->N	3 + N + E
N->D	3 + D + E
D->5	3 + 5 + E
E->A	3 + 5 + A
A->- E	3 + 5 + - E
E->N	3 + 5 + - N
N->D	3 + 5 + - D
D->7	3 + 5 + - 7

Figura 5.1.5 - Entrada 3+5+-7.

Tal esforço computacional é demonstrado na Fig. 5.1.6, onde a única alteração entre a expressão anterior e esta, foi o 1° número que se tornou negativo.

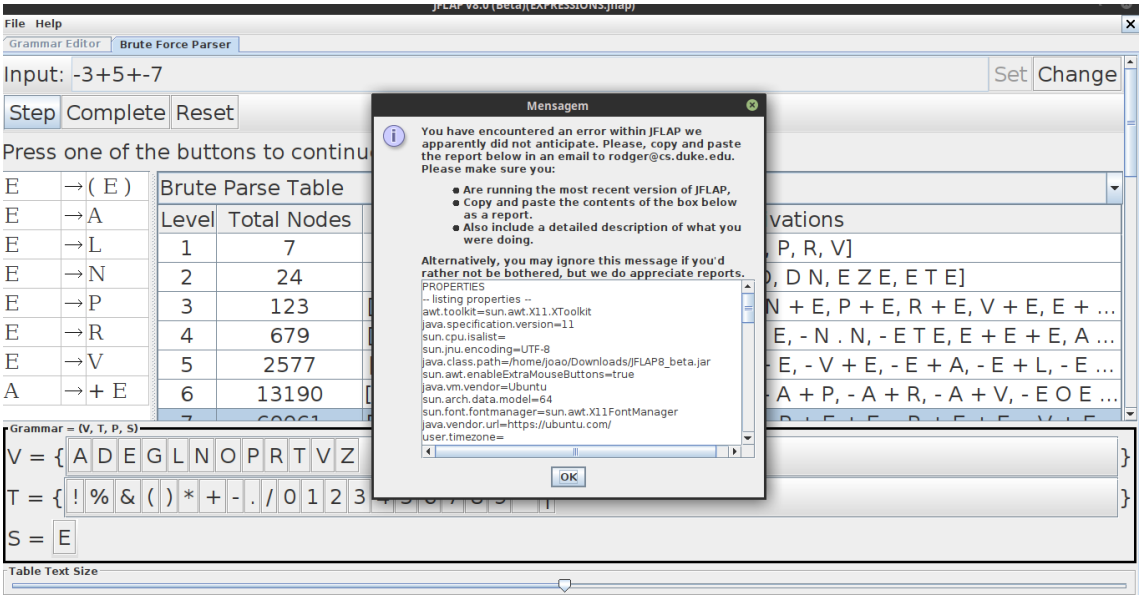

Figura 5.1.6 - Problema de grande esforço computacional para a entrada 3+5+-7.

- ***div_expression***

Quanto às expressões que envolvem multiplicação e divisão, verificou-se que funcionam perfeitamente. Os testes realizados podem ser vistos nas Figs. 5.1.7 a 5.1.10, os mesmos envolvem divisão simples, divisão com números negativos e somas.

Production	Derivation
	E
E->V	V
V->E T E	E T E
E->N	N T E
N->D	D T E
D->4	4 T E
T->/	4 / E
E->N	4 / N
N->D	4 / D
D->2	4 / 2

Figura 5.1.7 - Entrada 4/2.

Production	Derivation
	E
E->A	A
A->- E	- E
E->V	- V
V->E T E	- E T E
E->N	- N T E
N->D	- D T E
D->5	- 5 T E
T->/	- 5 / E
E->N	- 5 / N
N->D N	- 5 / D N
D->1	- 5 / 1 N
N->D	- 5 / 1 D
D->0	- 5 / 1 0

Figura 5.1.8 - Entrada -5/10.

Production	Derivation
	E
E->A	A
A->- E	- E
E->V	- V
V->E T E	- E T E
E->N	- N T E
N->D	- D T E
D->4	- 4 T E
T->/	- 4 / E
E->A	- 4 / A
A->- E	- 4 / - E
E->N	- 4 / - N
N->D	- 4 / - D
D->2	- 4 / - 2

Figura 5.1.9 - Entrada -4/-2.

Production	Derivation
	E
E->A	A
A->E + E	E + E
E->V	V + E
V->E T E	E T E + E
E->N	N T E + E
N->D	D T E + E
D->4	4 T E + E
T->/	4 / E + E
E->N	4 / N + E
N->D	4 / D + E
D->2	4 / 2 + E
E->N	4 / 2 + N
N->D	4 / 2 + D
D->1	4 / 2 + 1

Figura 5.1.10 - Entrada 4/2+1.

Para realizar operações envolvendo potenciação de dois elevado a dois por exemplo, deve-se utilizar a forma 2^2. Vários testes foram realizados, os principais estão descritos nas Figs. 5.1.11 a 5.1.13, onde é possível ver uma operação simples, uma com números negativos e outra envolvendo as operações anteriormente citadas nesta seção.

Production	Derivation
	E
E->P	P
P->E Z E	E Z E
E->N	N Z E
N->D	D Z E
D->2	2 Z E
Z->^	2 ^ E
E->N	2 ^ N
N->D	2 ^ D
D->2	2 ^ 2

Figura 5.1.11 - Entrada 2^2.

Production	Derivation
	E
E->P	P
P->E Z E	E Z E
E->N	N Z E
N->D	D Z E
D->2	2 Z E
Z->^	2 ^ E
E->A	2 ^ A
A->- E	2 ^ - E
E->N	2 ^ - N
N->D	2 ^ - D
D->2	2 ^ - 2

Figura 5.1.12 - Entrada 2^-2.

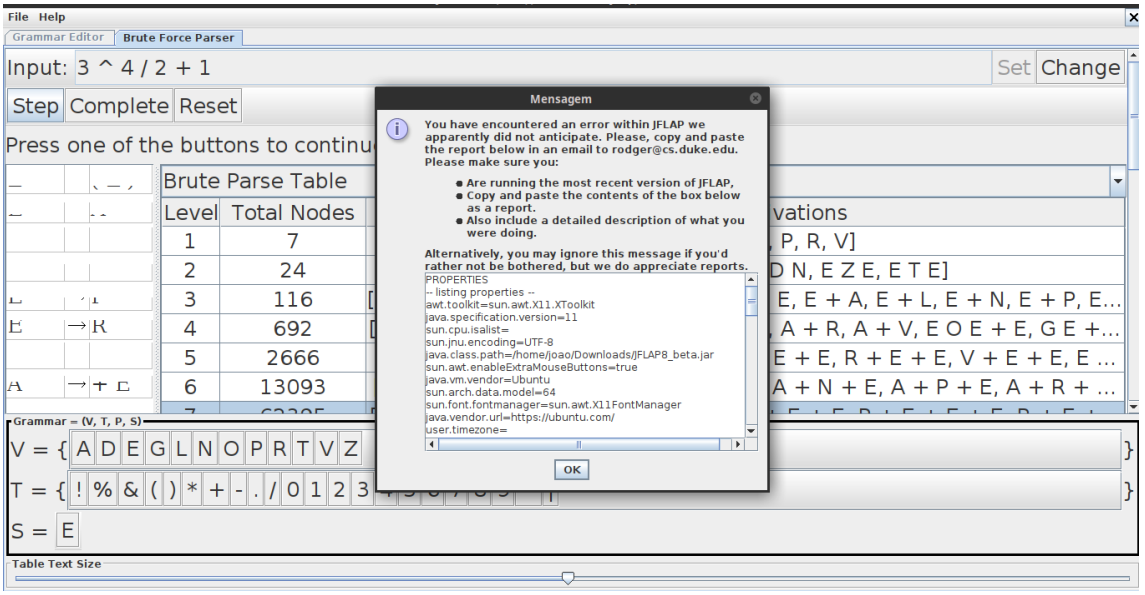

Figura 5.1.13 - Problema de grande esforço computacional para a entrada *3 ^ 4 / 2 + 1*.

Expressões lógicas podem ser construídas com os caracteres '&' e '|', representando respectivamente as operações **and** e **or**. Segue abaixo quatro testes relevantes para se verificar o funcionamento correto desta etapa. Vale ressaltar que testes com valores com muitos algarismos demandam muito poder computacional e não pode ser processado, como: *5 | 3 ^ 4 / 2 + 1*.

Production	Derivation
	E
E->L	L
L->E O E	E O E
E->N	N O E
N->D	D O E
D->3	3 O E
O->&	3 & E
E->N	3 & N
N->D	3 & D
D->4	3 & 4

Figura 5.1.14 - Entrada 3 & 4.

Derivation Tree | Derivation Table

Production	Derivation
	E
E->L	L
L->E O E	E O E
E->N	N O E
N->D	D O E
D->5	5 O E
O->|	5 | E
E->N	5 | N
N->D	5 | D
D->8	5 | 8

Figura 5.1.15 - Entrada *5 | 8.*

	E
E->L	L
L->E O E	E O E
E->N	N O E
N->D	D O E
D->5	5 O E
O->|	5 | E
E->P	5 | P
P->E Z E	5 | E Z E
E->N	5 | N Z E
N->D	5 | D Z E
D->3	5 | 3 Z E
Z->^	5 | 3 ^ E
E->N	5 | 3 ^ N
N->D	5 | 3 ^ D
D->4	5 | 3 ^ 4

Figura 5.1.16 - Entrada *5 | 3 ^ 4.*

- **(*expression*), number e real_number**

Qualquer expressão pode vir entre parênteses e isso é demonstrado na Fig. 5.1.17. Ademais, os números compostos de mais de um algarismo são construídos por meio de chamadas à produção *number*, como é possível ver na figura 5.1.18. Outrossim, os números reais também são construídos com o auxílio da produção number, bem como mostram as Figs. 5.1.19 e 5.1.20. Vale ressaltar que entradas como *2.1 + 3.2* e 2.37+56.4 devido ao alto poder computacional demandado, não puderam ser computadas.

Production	Derivation
	E
E->(E)	(E)
E->A	(A)
A->E + E	(E + E)
E->N	(N + E)
N->D	(D + E)
D->2	(2 + E)
E->N	(2 + N)
N->D	(2 + D)
D->2	(2 + 2)

Figura 5.1.17 - Entrada (2+2).

	E
E->N	N
N->D N	D N
D->1	1 N
N->D N	1 D N
D->0	1 0 N
N->D N	1 0 D N
D->0	1 0 0 N
N->D	1 0 0 D
D->0	1 0 0 0

Figura 5.1.18 - Entrada 1000.

Production	Derivation
	E
E->R	R
R->N . N	N . N
N->D	D . N
D->2	2 . N
N->D	2 . D
D->1	2 . 1

Figura 5.1.19 - Entrada 2.1.

	E
E->A	A
A->E + E	E + E
E->R	R + E
R->N . N	N . N + E
N->D	D . N + E
D->2	2 . N + E
N->D	2 . D + E
D->1	2 . 1 + E
E->N	2 . 1 + N
N->D	2 . 1 + D
D->1	2 . 1 + 1

Figura 5.1.20 - Entrada 2.1 + 1.

5.2 - Testes para as derivações de *commands*

Foram realizados testes a fim de validar a gramática das derivações de *commands*. Para isso inseriu-se no JFlap as produções da Tabela 5.2, e em seguida, testou-se com entradas diversas descritas abaixo (cada tópico refere-se a uma derivação em específico).

- **Comando de atribuição**

O primeiro comando a ser testado foi o de atribuição, na Fig. 5.2.1 é possível ver a derivação à esquerda da atribuição x=1. Um outro tipo de atribuição muito importante na linguagem é o vetor, no qual tem o seguinte formato v[1] = 1 (Fig. 5.2.2). Ademais, a atribuição s→ v=1 também é permitida (Fig. 5.2.3).

Production	Derivation
	C
C->V A E	V A E
V->D	D A E
D->O N	O N A E
O->x	x N A E
N->λ	x A E
A->=	x = E
E->1	x = 1

Figura 5.2.1 - Entrada x=1.

Production	Derivation
	C
C->V A E	V A E
V->H	H A E
H->D [E]	D [E] A E
D->O N	O N [E] A E
O->v	v N [E] A E
N->λ	v [E] A E
E->1	v [1] A E
A->=	v [1] = E
E->1	v [1] = 1

Figura 5.2.2 - Entrada v[1] = 1.

Production	Derivation
	C
C->V A E	V A E
V->Q	Q A E
Q->D - > D	D - > D A E
D->O N	O N - > D A E
O->s	s N - > D A E
N->λ	s - > D A E
D->O N	s - > O N A E
O->v	s - > v N A E
N->λ	s - > v A E
A->=	s - > v = E
E->1	s - > v = 1

Figura 5.2.3 - Entrada s→ v=1.

- ***Comandos if, for e while***

As palavras reservadas foram representadas nas entradas com o numeral 0, logo, em todas as ocorrências de palavras significativas como for, while etc, se encontra o n° 0. A fim de facilitar a refatoração das produções commands a produção expressions foi reduzida a dígitos de 1 a 9 uma vez que todas as expressões resultam em valores numéricos. Abaixo segue um exemplo dessas representações:

- ➔ Comando if: **0 1 : x = 1**. Neste exemplo nota-se a ocorrência de uma palavra reservada seguida de uma expressão, um delimitador ‘:’ e uma atribuição.
- ➔ Comando for: ***0 1, 1, 1: X=1 0.*** Neste exemplo nota-se a ocorrência de uma palavra reservada, 3 expressões separadas por vírgula seguidas de um delimitador ‘:’ e por fim uma atribuição com uma palavra reservada ao final indicando que o comando for acabou.
- ➔ Comando while: ***0 1 : x=1 0.*** Neste exemplo nota-se a ocorrência de uma palavra reservada seguida de uma expressão, com um delimitador ‘:’ que separa a atribuição e a palavra que sinaliza fim do comando while.

Ademais, os testes destes comandos demandaram um grande poder computacional, levando a uma aparente tela congelada. Isso, como já explicado anteriormente, ocasionou-se pelo fato de que a entrada possuía muitos algarismos a serem processados.

- ***Comandos diversos***

Tanto o *farewell_command* quanto o *stop_command* são comandos de apenas uma palavra reservada, por isso eles possuem a mesma formatação de acordo com nossas representações, que no caso é o numeral 0, Fig. 5.2.4.

O *jumpto_command* é basicamente um comando de uma palavra reservada seguida de um valor de desvio (*label*), isso pode ser visto na Fig. 5.2.5. Ademais, os comandos *say_command, listen_command, task_command, e label* estão representados nas Figs. 5.2.4 a 5.2.9 respectivamente.

Production	Derivation
	C
C->R	R
R->Z	Z
Z->0	0

Figura 5.2.4 - Entrada 0.

Production	Derivation
	C
C->J	J
J->Z D	Z D
Z->0	0 D
D->O N	0 O N
O->v	0 v N
N->λ	0 v

Figura 5.2.5 - Entrada 0 v.

Production	Derivation
	C
C->Y	Y
Y->Z E	Z E
Z->0	0 E
E->1	0 1

Figura 5.2.6 - Entrada 0 1.

Production	Derivation
	C
C->J	J
J->Z D	Z D
Z->0	0 D
D->O N	0 O N
O->a	0 a N
N->λ	0 a

Figura 5.2.7 - Entrada 0 a.

Production	Derivation
	C
C->T	T
T->Z D K : E Z	Z D K : E Z
Z->0	0 D K : E Z
D->O N	0 O N K : E Z
O->v	0 v N K : E Z
N->λ	0 v K : E Z
K->G	0 v G : E Z
G->E	0 v E : E Z
E->1	0 v 1 : E Z
E->1	0 v 1 : 1 Z
Z->0	0 v 1 : 1 0

Figura 5.2.8 - Entrada 0 v 1 : 1 0.

Production	Derivation
	C
C->B	B
B->D : C	D : C
D->O N	O N : C
O->v	v N : C
N->λ	v : C
C->V A E	v : V A E
V->D	v : D A E
D->O N	v : O N A E
O->x	v : x N A E
N->λ	v : x A E
A->=	v : x = E

Figura 5.2.9 - Entrada v : x = E.

6 - Considerações Finais

Neste trabalho foi apresentada uma nova linguagem de programação denominada Chameleon, sua gramática, seu pré-processador, suas particularidades, o arquivo para a geração do analisador léxico em sua forma preliminar de impressão de casamentos, testes com programas válidos e inválidos lexicalmente e testes da gramática no JFLAP.

A linguagem tem como um dos objetivos oferecer maior flexibilidade ao programador, e daí veio seu nome e a construção de seu pré-processador. Elaborar essa linguagem permitiu um maior ganho e reforço de conhecimento do grupo acerca dos projetos de linguagens de programação. Durante o processo de sua construção, a gramática passou por diversas modificações e correções, sendo que uma dessas versões, capaz de representar bem a linguagem como ela é agora, foi validada através da ferramenta JFLAP. Apesar desta ferramenta ter apresentado algumas limitações, ela foi suficiente para avaliar as principais partes da gramática da linguagem.

O pré-processador da linguagem permite a customização de códigos, permitindo que pessoas com deficiência visual, por exemplo, consigam ter uma maior facilidade na programação pela maior legibilidade proporcionada pela macro disponível para esta linguagem e apresentada neste trabalho. Sua customização não se limita a isso, podendo até mesmo ter códigos com nomes de comandos similares a linguagens conhecidas como C, Python, ou qualquer outra que o programador se sinta mais confortável.

Pelos testes do analisador léxico com exemplos de programas escritos na linguagem Chameleon, foi possível observar que a construção do que fora proposto está adequada para as próximas etapas deste trabalho prático. O mecanismo de supressão de erros adicionada à linguagem e ao seu analisador léxico é um recurso interessante, pois permite que programas lexicalmente inválidos prossigam para as próximas etapas do compilador, captando apenas as partes lexicalmente válidas do programa.

Referências Bibliográficas

[1] AHO, A.V.; LAM, M.S.; SETHI, R.; ULLMAN, J.D. Compiladores: Princípios, técnicas e ferramentas. Segunda Edição. Pearson Addison-Wesley, 2008.

[2] Gramática para a linguagem C (YACC). Disponível em: <https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#relational-expression> Acesso em: 01 de novembro de 2020.

[3] Flex. Disponível em: https://github.com/westes/flex

[4] “How to wrap printf() into a function or macro?”. Disponível em: https://stackoverflow.com/questions/20639632/how-to-wrap-printf-into-a-function-or-macro

[5] Flex - Documentação. Disponível em: https://github.com/westes/flex/blob/master/doc/flex.texi

[6] “How to wrap printf() into a function or macro?”. Disponível em: https://www.quora.com/What-is-use-of-yywrap-in-LEX

Apêndice A - Código em *lex.l* para a linguagem Chameleon

```
%{
#include <string.h>

#define PRINT_ERROR "---------- Erro encontrado na linha %d ----------\n"
#define PRINT_ERROR_EOF "-> Um ou mais erros foram encontrados. Corrija-os!\n"
#define PRINT_PREFIX "Line %d: I've found a"
#define PRINT(args) printf args ;
#define PRINT_LEXEME " LEXEME: %s\n"
#define PRINT_REAL_NUMBER PRINT_PREFIX " real number."
#define PRINT_NUMBER PRINT_PREFIX "n integer number."
#define PRINT_WORD PRINT_PREFIX " word."
#define PRINT_TYPE PRINT_PREFIX " type."
#define PRINT_SQUAD_DECLARATION PRINT_PREFIX " squad declaration."
#define PRINT_VECTOR_DECLARATION PRINT_PREFIX " vector declaration."
#define PRINT_SQUAD_END PRINT_PREFIX " squad end."
#define PRINT_BLOCK_BEGIN PRINT_PREFIX " block begin."
#define PRINT_BLOCK_END PRINT_PREFIX " block end."

#define PRINT_FOR PRINT_PREFIX " for."
#define PRINT_FOR_END PRINT_PREFIX " for end."
#define PRINT_WHILE PRINT_PREFIX " while."
#define PRINT_WHILE_END PRINT_PREFIX " while end."
#define PRINT_IF PRINT_PREFIX "n if."
#define PRINT_IF_END PRINT_PREFIX "n if end."
#define PRINT_ELIF PRINT_PREFIX "n elif."
#define PRINT_ELIF_END PRINT_PREFIX " endelif end."
#define PRINT_TASK PRINT_PREFIX " task."
#define PRINT_TASK_END PRINT_PREFIX " task end."

#define PRINT_JUMPTO PRINT_PREFIX " jumpto."
#define PRINT_FAREWELL PRINT_PREFIX " farewell."
#define PRINT_SAY PRINT_PREFIX " say."
#define PRINT_LISTEN PRINT_PREFIX " listen."
#define PRINT_STOP PRINT_PREFIX " stop."

#define PRINT_COMMA PRINT_PREFIX " comma."
#define PRINT_OPEN_PARENTHESIS PRINT_PREFIX "n open parenthesis."
#define PRINT_CLOSE_PARENTHESIS PRINT_PREFIX " close parenthesis."

#define PRINT_IDENTIFIER PRINT_PREFIX "n identifier."
#define PRINT_VECTOR_ACCESS PRINT_PREFIX " vector access."
#define PRINT_VECTOR_ACCESS_START PRINT_PREFIX " vector access start."
#define PRINT_VECTOR_ACCESS_END PRINT_PREFIX " vector access end."


#define PRINT_SQUAD_ACCESS_DERREFERENCE PRINT_PREFIX " squad access derreference."
#define PRINT_SEPARATOR PRINT_PREFIX " separator."
#define PRINT_WORD_CONCAT_OPERATOR PRINT_PREFIX " word concatenation operator."
#define PRINT_ADD_OPERATOR PRINT_PREFIX "n add operator."
#define PRINT_DIV_OPERATOR PRINT_PREFIX " div operator."
#define PRINT_POW_OPERATOR PRINT_PREFIX " pow operator."
#define PRINT_REL_OPERATOR PRINT_PREFIX " rel operator."
#define PRINT_LOGIC_OPERATOR PRINT_PREFIX " logic operator."
#define PRINT_ATTRIBUTION PRINT_PREFIX "n attribution."

void remover_espacos_e_print(int t);
```

```
57  #define _REAL 1
58  #define _NUMBER 2
59
60  int supress_errors_flag = 0;
61  int erro_encontrado = 0;
62  %}
63  /* condicao exclusiva (bloqueia as demais regras) */
64  %x comment_condition
65  %x word_condition
66  /* condicao que e ativa mas mantem as demais ativadas tambem */
67  %s supress_errors_condition
68  /* Permitir a contabilizacao de linhas */
69  %option   yylineno
70
71  supress_errors            "SUPRESS_ERRORS"
72
73  /* captura uma ocorrencia de espaco em branco, tabulacao ou quebra de linha*/
74  delim                     [ \t\n\r]
75  /* ign (ignorador) ira ignorar um ou mais delim*/
76  ign                       {delim}+
77  letter                    [A-Za-z]
78  digit                     [0-9]
79  underline                 _
80  word_value                (\\.|[^"\\])*
81  number                    {ign}*({digit}{ign}*)+
82  type                      "integer"|"word"|"real"
83  squad_declaration         "squad"
84  vector_declaration        "vector"
85  end_squad                 "endsquad"
86  block_begin               "begin"
87  block_end                 "end"
88  for                       "for"
89  end_for                   "endfor"
90  while                     "while"
91  end_while                 "endwhile"
92  if                        "if"
93  end_if                    "endif"
94  elif                      "elif"
95  end_elif                  "endelif"
96  task                      "task"
97  end_task                  "endtask"
98
99  jumpto                    "jumpto"
100 farewell                  "farewell"
101 say                       "say"
102 listen                    "listen"
103 stop                      "stop"
104 comma                     ","
105 open_parenthesis          "("
106 close_parenthesis         ")"
107 identifier_complement     ({digit}|{letter}|{underline})*
108 identifier                ({letter}|{underline}{letter}){identifier_complement}
109 vector_access_start       "["
110 vector_access_end         "]"
111 squad_access_derreference "->"
112 separator                 ":"
113 real_number               {ign}*({digit}{ign}*)+{ign}*(\.{ign}*({digit}{ign}*)+)
114 word_concat_operator      "++"
115 add_operator              [+-]
```

```
116 div_operator              [/*\%]
117 pow_operator              [\^]
118 rel_operator              "=="|"!="|">="|"<="|[><]
119 logic_operator            "and"|"or"|"!"
120 attribution               "="
121 creditos                  "??creditos??"
122
123 %%
124 {supress_errors}           { BEGIN(supress_errors_condition); supress_errors_flag = 1; }
125 "giveup"                   {return 0;}
126 "//"                       BEGIN(comment_condition);
127 {ign}                      {}
128
129 <comment_condition>.|\n     {}
130 <comment_condition>"\\\\"   { BEGIN(INITIAL); if(supress_errors_flag) BEGIN(supress_errors_condition); }
131
132 "\""                       BEGIN(word_condition);
133 <word_condition>{word_value}        {PRINT((PRINT_WORD PRINT_LEXEME, yylineno, yytext))}
134 <word_condition>"\""                { BEGIN(INITIAL);  if(supress_errors_flag) BEGIN(supress_errors_condition); }
135
136 {creditos}              {PRINT(("Feito por:\n%s\n",
137                         "Daniel Freitas Martins - 2304\n"
138                         "João Arthur Gonçalves do Vale - 3025\n"
139                         "Maria Dalila Vieira - 3030\n"
140                         "Naiara Cristiane dos Reis Diniz - 3005"))}
141
142 {type}                  {PRINT((PRINT_TYPE PRINT_LEXEME, yylineno, yytext))}
143 {squad_declaration}     {PRINT((PRINT_SQUAD_DECLARATION PRINT_LEXEME, yylineno, yytext))}
144 {vector_declaration}    {PRINT((PRINT_VECTOR_DECLARATION PRINT_LEXEME, yylineno, yytext))}
145
146 {end_squad}             {PRINT((PRINT_SQUAD_END PRINT_LEXEME, yylineno, yytext))}
147 {block_begin}           {PRINT((PRINT_BLOCK_BEGIN PRINT_LEXEME, yylineno, yytext))}
148 {block_end}             {PRINT((PRINT_BLOCK_END PRINT_LEXEME, yylineno, yytext))}
149
150 {for}                   {PRINT((PRINT_FOR PRINT_LEXEME, yylineno, yytext))}
151 {end_for}               {PRINT((PRINT_FOR_END PRINT_LEXEME, yylineno, yytext))}
152 {while}                 {PRINT((PRINT_WHILE PRINT_LEXEME, yylineno, yytext))}
153 {end_while}             {PRINT((PRINT_WHILE_END PRINT_LEXEME, yylineno, yytext))}
154 {if}                    {PRINT((PRINT_IF PRINT_LEXEME, yylineno, yytext))}
155 {end_if}                {PRINT((PRINT_IF_END PRINT_LEXEME, yylineno, yytext))}
156 {elif}                  {PRINT((PRINT_ELIF PRINT_LEXEME, yylineno, yytext))}
157 {end_elif}              {PRINT((PRINT_ELIF_END PRINT_LEXEME, yylineno, yytext))}
158 {task}                  {PRINT((PRINT_TASK PRINT_LEXEME, yylineno, yytext))}
159 {end_task}              {PRINT((PRINT_TASK_END PRINT_LEXEME, yylineno, yytext))}
160
161 {jumpto}                {PRINT((PRINT_JUMPTO PRINT_LEXEME, yylineno, yytext))}
162 {farewell}              {PRINT((PRINT_FAREWELL PRINT_LEXEME, yylineno, yytext))}
163 {say}                   {PRINT((PRINT_SAY PRINT_LEXEME, yylineno, yytext))}
164 {listen}                {PRINT((PRINT_LISTEN PRINT_LEXEME, yylineno, yytext))}
165 {stop}                  {PRINT((PRINT_STOP PRINT_LEXEME, yylineno, yytext))}
166
167 {comma}                 {PRINT((PRINT_COMMA PRINT_LEXEME, yylineno, yytext))}
168 {open_parenthesis}      {PRINT((PRINT_OPEN_PARENTHESIS PRINT_LEXEME, yylineno, yytext))}
169 {close_parenthesis}     {PRINT((PRINT_CLOSE_PARENTHESIS PRINT_LEXEME, yylineno, yytext))}
170
171 {vector_access_start}   {PRINT((PRINT_VECTOR_ACCESS_START PRINT_LEXEME, yylineno, yytext))}
172 {vector_access_end}     {PRINT((PRINT_VECTOR_ACCESS_END PRINT_LEXEME, yylineno, yytext))}
173
174
```

```
{squad_access_derreference}     {PRINT((PRINT_SQUAD_ACCESS_DERREFERENCE PRINT_LEXEME, yylineno,
yytext))}
{separator}            {PRINT((PRINT_SEPARATOR PRINT_LEXEME, yylineno, yytext))}
{word_concat_operator}      {PRINT((PRINT_WORD_CONCAT_OPERATOR PRINT_LEXEME, yylineno, yytext))}
{add_operator}               {PRINT((PRINT_ADD_OPERATOR PRINT_LEXEME, yylineno, yytext))}
{div_operator}               {PRINT((PRINT_DIV_OPERATOR PRINT_LEXEME, yylineno, yytext))}
{pow_operator}               {PRINT((PRINT_POW_OPERATOR PRINT_LEXEME, yylineno, yytext))}
{rel_operator}               {PRINT((PRINT_REL_OPERATOR PRINT_LEXEME, yylineno, yytext))}
{logic_operator}             {PRINT((PRINT_LOGIC_OPERATOR PRINT_LEXEME, yylineno, yytext))}
{attribution}                {PRINT((PRINT_ATTRIBUTION PRINT_LEXEME, yylineno, yytext))}
{real_number}                {remover_espacos_e_print(_REAL);}
{number}                     {remover_espacos_e_print(_NUMBER);}

{identifier}                 {PRINT((PRINT_IDENTIFIER PRINT_LEXEME, yylineno, yytext))}
<supress_errors_condition>.     {}
";"+                         {} /* ignorando ponto e virgula */
.                            {PRINT((PRINT_ERROR "-> %s\nint_code_s0: %d\n", yylineno, yytext, yytext[0])) erro_encontrado = 1;}
/* Ignorar o que nao foi definido */
<<EOF>>                      {
            if(erro_encontrado){
               PRINT((PRINT_ERROR_EOF))
               exit(1);
            }
            return 0;
         }

%%
void remover_espacos_e_print(int t){
   char* s; /* tera a nova string sem os espacos em branco */
   int i, j, tam_yytext = strlen(yytext);
   s = (char*) malloc(tam_yytext*sizeof(char));
   j = 0;
   for(i = 0; i < tam_yytext; i++){
      if(yytext[i] == ' ' || yytext[i] == '\n' || yytext[i] == '\t')
         continue;
      s[j++] = yytext[i];
   }
   s[j] = '\0';

   switch(t){
      case _REAL:
         PRINT((PRINT_REAL_NUMBER PRINT_LEXEME, yylineno, s))
      break;
      case _NUMBER:
         PRINT((PRINT_NUMBER PRINT_LEXEME, yylineno, s))
      break;
   }
   free(s);
}
int yywrap(){ return 1; } /* se EOF for encontrado, encerre. */
int main(){
   yylex();
   return 0;
}
```

Apêndice B - Código em Python correspondente ao pré-processador da linguagem Chameleon

```
import sys

def match(term,lst,used):
    if lst[0].strip() == term:
       # print(lst)
        if lst[1].strip() in keywords.values():
            print("palavras chave repetidas!!!")
            exit(1)
        s = lst[1].strip()
        used.append(s)
        keywords[term] = s
        return True

def ordenarVetorPeloTamanho(used):
    return used.sort(key=len, reverse=True)

#return, func, goto, break, print, input, float, string, const, if/else
keywords = {}
defs = {}
used = []
block = "Chameleon"

keywords["farewell"] = None
keywords["task"] = None
keywords["stop"] = None
keywords["jumpto"] = None
keywords["say"] = None
keywords["listen"] = None
keywords["if"] = None
keywords["elif"] = None
keywords["for"] = None
keywords["while"] = None

keywords["real"] = None
keywords["word"] = None
keywords["integer"] = None
keywords["vector"] = None
keywords["squad"] = None

keywords["+"] = None
keywords["-"] = None
keywords["*"] = None
keywords["/"] = None
keywords["%"] = None
keywords["="] = None
keywords["=="] = None
keywords["^"] = None
keywords["!"] = None
keywords["!="] = None
keywords[">="] = None
keywords["<="] = None
keywords[">"] = None
keywords["<"] = None
keywords["and"] = None
```

```
55  keywords["or"] = None
56  keywords[","] = None
57  keywords["."] = None
58  keywords["//"] = None
59  keywords['\\\\'] = None
60
61
62  name = ""
63
64  #MAIN
65  if len(sys.argv) > 1:
66      name = sys.argv[1]
67  else:
68      print("n")
69      exit(1)
70
71  arquivo_in = open(name, 'r')
72  arquivo_out = open(name+'.out', 'w')
73
74  init = False
75  code = False
76  tokens = []
77  pilha = []
78  arquivo_out.write('begin\n')
79  for line in arquivo_in:
80      #print(line)
81      if line.find("#MACROS") != -1:
82          print("bloco de macros")
83          init = True
84
85      if line.find("#ENDMACROS") != -1:
86          init = False
87
88      done = False
89      if init:
90          lst = line.split(":")
91          if lst[0].strip() == 'def':
92              defs[lst[1].strip()] = lst[2].strip()
93          elif lst[0].strip() == 'block':
94              block = lst[1].strip()
95              if block == "Pascal":
96                  keywords[":"] = "begin"
97                  used.append("begin")
98
99              elif block == "C":
100                 keywords[":"] = "{"
101                 used.append("{")
102
103         for term in keywords.keys():
104             if done:
105                 break
106             done = match(term,lst,used)
107
108     if line.find("#ENDCOD") != -1:
109         code = False
110
111     if code:
112         line2 = line.strip()
113         lst = line2.split(" ")
```

```
114
115         print(lst)
116
117         for i in range(len(lst)):
118             for item in used:
119                 if lst[i].find(item) != -1:
120                    #print(item)
121                    for k in keywords.keys():
122                        if item == keywords[k]:
123                            lst[i] = lst[i].replace(item,k)
124                    break
125         #print(lst)
126         tokens.append(lst)
127
128         for l in lst:
129             if(block == "C"):
130                 commands = ["while", "if", "task", "squad", "for"]
131
132                 print("LLLLLLLL",l)
133                 if l in commands:
134                     pilha.append(l)
135                 elif l == "}elif":
136                     pilha.pop()
137                     l = "elif"
138                     pilha.append(l)
139                 elif l == "}":
140                     l = "end"+pilha.pop()
141
142             elif(block == "Pascal"):
143                 commands = ["while", "if", "task", "squad", "for"]
144
145                 print("LLLLLLLL",l)
146                 if l in commands:
147                     pilha.append(l)
148                 elif l == "elif":
149                     pilha.pop()
150                     pilha.append(l)
151                 elif l == "end":
152                     l = "end"+pilha.pop()
153
154             print("PILHAAAA",pilha)
155             arquivo_out.write(l+" ")
156         arquivo_out.write("\n")
157        # print(line)
158
159     if line.find("#COD") != -1:
160         ordenarVetorPeloTamanho(used)
161         print(used)
162         print("bloco de macros")
163         code = True
164 arquivo_out.write('end')
165
166 #print(keywords)
167 print("TOKENS",tokens)
168 #print(defs)
169 #print(block)
```