

# Tradutor da Linguagem C-IPL

Pedro Vitor Valença Mizuno<sup>[17/0043665]</sup>

Universidade de Brasília, DF, Brasil

## 1 Motivação

Linguagens de programação são formas de se descrever, por meio de termos, uma computação de forma mais simples e clara para pessoas. No entanto, para que os comandos contidos nessa linguagem sejam compreendidos por uma máquina, é necessária a utilização de um tradutor, um programa que traduz as linhas de código para um formato capaz de ser executado pelo computador. [ALSU06]

Como projeto da disciplina Tradutores, foi proposta a criação de um tradutor para a linguagem C-IPL, uma linguagem de programação baseada em C. O C-IPL apresenta como seu maior diferencial a existência da primitiva *list*, que permite criar uma lista de inteiros ou números de ponto flutuante. A primitiva *list* é uma ferramenta poderosa devido ao fato de criar uma estrutura de dados em formato de lista que permite adicionar e remover elementos de forma simples. Além disso, as operações *map* e *filter* (*>>* e *<<*, respectivamente) geram uma forma efetiva de operar com funções sobre as listas utilizadas.

## 2 Descrição

A análise léxica consiste na primeira fase do tradutor, em que caracteres são casados com um padrão definido por um *token*, sendo identificado como uma instância desse *token*. Como saída, o analisador léxico produz um *token*, o qual é composto por um nome e um valor, no formato (nome-token, valor-token), ou apenas o nome do *token* caso não exista um valor, como para palavras reservadas e símbolos como chaves e parênteses.

A tabela de símbolos consiste em uma estrutura de dados que é usada pelo tradutor para guardar informações acerca dos símbolos que venham a aparecer no código a ser traduzido. Para tal, planeja-se que a tabela de símbolos seja construída a partir de uma lista encadeada e que contenha os campos de lexema, tipo, endereço e informações a cerca de seu escopo. A medida que os símbolos são identificados, seus atributos serão armazenados na tabela e, além disso, serão implementadas novas tabelas de símbolos para os escopos que surjam.

A partir do *software FLEX* [EP01], é possível utilizar expressões regulares para casar os lexemas com os padrões dos *tokens* da linguagem C-IPL, como tipos, identificadores, constantes, etc. Ao ocorrer uma correspondência entre lexema e expressão regular, é impresso o *token* correspondente, assim como a linha e coluna onde ela ocorreu. Caso venha a ocorrer um erro léxico, ele é anunciado a partir de uma mensagem que aponta a linha e coluna onde houve o

erro. Foram também tratados dois erros em específico, o comentário em múltiplas linhas sem fechamento e a string em múltiplas linhas.

### 3 Arquivos Teste

A fim de testar as funcionalidades do analisador léxico, foram criados quatro arquivos de entrada em linguagem C-IPL, localizados no diretório *tests*, para serem avaliados pelo programa:

1. *teste1\_correto.ci*: programa sem erros léxicos que recebe uma lista e retorna se a soma dos elementos contidos nessa lista é positiva ou negativa;
2. *teste2\_correto.ci*: programa sem erros léxicos que monta em uma lista a sequência de *Fibonacci* até uma posição *n*;
3. *teste1\_errado.ci*: programa com erros léxicos que calcula e apresenta a classificação do IMC de acordo com os dados inseridos. Os erros presentes são os pontos utilizados na linha 13, a string não fechada na linha 14, o *AND* com apenas um *&* na linha 17 e o comentário sem fechamento na linha 43;
4. *teste2\_errado.ci*: programa com erros léxicos que encontra o menor número de ponto flutuante de uma lista. Os erros presentes são as strings sem fechamento nas linha 14 e 30, o símbolo barra invertida na linha 25 e o símbolo *.* na linha 34.

### 4 Instrução de Compilação

Versões das ferramentas utilizadas:

- Ubuntu 21.04
- FLEX 2.6.4
- GCC 11.1.0
- Make 4.3
- Kernel 5.11.0-25-generic

Para facilitar a compilação do analisador léxico criado, foi criado um *makefile* que sintetiza os passos de compilação em apenas um comando. Por meio de um terminal, no diretório do projeto (i.e. 17\_0043665), onde está o arquivo *makefile*, execute o comando: **make**. Compilado o projeto, basta apenas executá-lo com o comando: **./tradutor tests/nome\_do\_arquivo.ci**.

### Referências

- [ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 75 Arlington Street Suite 300 Boston MA 02116 USA, 2 edition, 2006.
- [EP01] Will Estes and Vern Paxson. The fast lexical analyzer - scanner generator for lexing in c and c++. <https://github.com/westes/flex>, 2001. [Accessed 08-August-2021].

## A Léxico

**Tabela 1.** Tabela que apresenta o léxico da linguagem C-IPL.

Lexema	Nome do Token	Valor do Atributo
Espaço	-	-
Comentário	-	-
<i>if</i>	<b><i>if</i></b>	-
<i>else</i>	<b><i>else</i></b>	-
<i>for</i>	<b><i>for</i></b>	-
<i>return</i>	<b><i>return</i></b>	-
<i>read</i>	<b><i>read</i></b>	-
<i>write</i>	<b><i>write</i></b>	-
<i>writeln</i>	<b><i>writeln</i></b>	-
Identificador	<b><i>IDENTIFICADOR</i></b>	Apontador para a entrada da tabela
Constante	<b><i>CONSTANTE</i></b>	Apontador para a entrada da tabela
String	<b><i>STRING</i></b>	Apontador para a entrada da tabela
int	<b><i>TIPO</i></b>	int
float	<b><i>TIPO</i></b>	float
int list	<b><i>TIPO</i></b>	int list
float list	<b><i>TIPO</i></b>	float list
+	<b><i>ARITOP_BAIXA</i></b>	+
-	<b><i>ARITOP_BAIXA</i></b>	-
*	<b><i>ARITOP_ALTA</i></b>	*
/	<b><i>ARITOP_ALTA</i></b>	/
&&	<b><i>LOGOP_AND</i></b>	&&
	<b><i>LOGOP_OR</i></b>	
!	<b><i>LOGOP_NEG</i></b>	!
<=	<b><i>RELOP_ALTA</i></b>	<=
<	<b><i>RELOP_ALTA</i></b>	<
>=	<b><i>RELOP_ALTA</i></b>	>=
>	<b><i>RELOP_ALTA</i></b>	>
==	<b><i>RELOP_BAIXA</i></b>	==
!=	<b><i>RELOP_BAIXA</i></b>	!=
:	<b><i>LISTOP</i></b>	:
?	<b><i>LISTOP</i></b>	?
!	<b><i>LISTOP</i></b>	!
%	<b><i>LISTOP</i></b>	%
>>	<b><i>LISTOP</i></b>	>>
<<	<b><i>LISTOP</i></b>	<<

**Tabela 2.** Segunda parte da tabela que apresenta o léxico da linguagem C-IPL.

Lexema	Nome do Token	Valor do Atributo
(	(	-
)	)	-
{	{	-
}	}	-
[	[	-
]	]	-
,	,	-
;	;	-

## B Gramática

A gramática utilizada no tradutor da linguagem C-ILP é apresentada nas seguintes expressões (a fim de facilitar a leitura, foram adotados ID e CONST, que correspondem a IDENTIFICADOR e CONSTANTE, respectivamente):

1.  $program \rightarrow program\ declaration \mid declaration$
2.  $declaration \rightarrow function \mid variable$
3.  $function \rightarrow \mathbf{TIPO\ ID}\ (parameters)\ \{moreStmt\}$
4.  $parameters \rightarrow parameters,\ variable \mid variable \mid \varepsilon$
5.  $moreStmt \rightarrow moreStmt\ stmt \mid stmt$
6.  $stmt \rightarrow oneLineStmt \mid multLineStmt$
7.  $oneLineStmt \rightarrow variable; \mid expression; \mid call; \mid IO; \mid return$
8.  $multLineStmt \rightarrow conditional \mid iteration$
9.  $IO \rightarrow in \mid out$
10.  $in \rightarrow \mathbf{read(ID)}$
11.  $out \rightarrow \mathbf{write(CONST)} \mid \mathbf{writeln(CONST)} \mid \mathbf{write(ID)} \mid \mathbf{writeln(ID)} \mid \mathbf{write(expression)} \mid \mathbf{writeln(expression)} \mid \mathbf{write(STRING)} \mid \mathbf{writeln(STRING)}$
12.  $variable \rightarrow \mathbf{TIPO\ ID}$
13.  $conditional \rightarrow \mathbf{if}\ (expLogic)\ bracesStmt \mid \mathbf{if}\ (expLogic)\ bracesStmt\ \mathbf{else}\ bracesStmt$

14.  $bracesStmt \rightarrow \{moreStmt\} \mid oneLineStmt$
15.  $iteration \rightarrow \mathbf{for}(expIte; expIte; expIte) bracesStmt$
16.  $expIte \rightarrow expression \mid \varepsilon$
17.  $expression \rightarrow \mathbf{ID} = expression \mid \mathbf{ID} \mid expLogic \mid expList$
18.  $expList \rightarrow !\mathbf{ID} \mid ?\mathbf{ID} \mid \% \mathbf{ID} \mid \mathbf{ID} < < \mathbf{ID} \mid \mathbf{ID} > > \mathbf{ID} \mid expression : \mathbf{ID}$
19.  $expLogic \rightarrow expLogic \parallel andLogic \mid andLogic$
20.  $andLogic \rightarrow andLogic \&\& expComp \mid expComp$
21.  $expComp \rightarrow expComp == expRel \mid expComp != expRel \mid expRel$
22.  $expRel \rightarrow expRel < expArit \mid expRel > expArit \mid expRel <= expArit \mid expRel >= expArit \mid expArit$
23.  $expArit \rightarrow expArit + expMul \mid expArit - expMul \mid expMul$
24.  $expMul \rightarrow expMul * negElement \mid expMul / negElement \mid negElement$
25.  $negElement \rightarrow !element \mid element$
26.  $element \rightarrow \mathbf{ID} \mid (expression) \mid call \mid \mathbf{CONST}$
27.  $call \rightarrow \mathbf{ID} (emptyArg)$
28.  $emptyArg \rightarrow arguments \mid \varepsilon$
29.  $arguments \rightarrow arguments, arg \mid arg$
30.  $arg \rightarrow \mathbf{ID} \mid expression$
31.  $return \rightarrow \mathbf{return CONST}; \mid \mathbf{return ID};$