

# Tradutor da Linguagem C-IPL

Pedro Vitor Valença Mizuno - 17/0043665

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 expressões regulares, sendo classificado como um *token*. Como saída, o analisador léxico produz um *token*, o qual é composto por um nome e um valor, no formato (nome-token, linha, coluna).

Por meio do programa *FLEX* [EP01], são criados autômatos a partir das expressões regulares. Por meio desses autômatos, é possível reconhecer os lexemas e classificá-los a partir do *token* gerado. Na primeira fase do projeto, ao ocorrer uma correspondência entre lexema e expressão regular, era impresso o *token* correspondente, assim como a linha e a coluna onde ela ocorreu, no entanto, durante a realização da segunda etapa do projeto, esse processo foi substituído pelo retorno do token gerado. Porém, ainda se mantém que caso ocorra um erro léxico, o lexema é anunciado a partir de uma mensagem que aponta a linha e a coluna onde houve o erro. Além disso, foi mantido também o tratamento de dois erros em específico, o comentário em múltiplas linhas sem fechamento e a string em múltiplas linhas.

A segunda etapa do projeto da disciplina consiste em implementar a análise sintática. Essa fase do tradutor recebe os *tokens* obtidos pelo analisador léxico, que representam o programa fonte. Então, é verificado se as cadeias de *token* correspondem a alguma regra da gramática [ALSU06]. Adicionalmente, a análise

sintática tem como responsabilidade gerar a árvore sintática abstrata e, opcionalmente, a tabela de símbolos, visto que também pode ser montada pela análise léxica.

Durante a segunda fase do projeto, foi utilizado o programa *Bison* [Cor14] para ser feita a análise sintática do tradutor. Para tal, foi utilizado o padrão LR(1) canônico e foi adaptada a gramática apresentada no apêndice B. Como citado anteriormente, o formato dos *tokens* retornados pelo analisador léxico consiste em uma *struct* que contém nome-token, linha e coluna, que são o lexema que representa o *token*, a linha e a coluna onde ele foi encontrado, respectivamente. A partir dos tokens retornados e as regras que fazem parte da gramática, são feitas reduções para identificar se a entrada é aceita ou não pela linguagem C-IPL. Caso seja encontrada alguma construção recusada pela gramática, é destacada a linha e a coluna onde ela ocorreu.

Para a criação da árvore sintática abstrata, foi definido que o tipo das reduções da gramática é a *struct* que define os nós da árvore e, ao encontrar o fim de uma produção, é criado um nó da árvore. O nó consiste em uma *struct* composta por um nome, que o identifica, um inteiro que representa o escopo ao qual pertence, três ponteiros para nós filhos, um ponteiro para uma lista de nós, que é utilizado para simplificar a árvore criada, e um ponteiro para a tabela de símbolos, para caso o símbolo do nó esteja presente nela. A fim de simplificar a árvore sintática gerada, foram adotadas duas técnicas. A primeira consiste em passar adiante o ponteiro do nó caso seja uma regra composta por apenas um não-terminal ou similar, como em:  $expLog \rightarrow expArit$ . Para a segunda técnica utiliza-se o ponteiro que aponta para uma lista de nós. A lista de nós evita que hajam nós redundantes na árvore abstrata, como em  $moreStmt \rightarrow moreStmt$  *block*, regra essa que poderia gerar vários nós desnecessários *moreStmt*. No entanto, por meio da lista de nós, é possível que todos os *stmt* sejam encadeados em uma mesma estrutura, cada qual podendo gerar sua própria árvore.

A tabela de símbolos consiste em uma estrutura de dados usada pelo tradutor para guardar informações acerca das variáveis e funções que venham a aparecer no código a ser traduzido. Para tal, foi criada uma lista encadeada em que cada elemento consiste em uma *struct* composta por: símbolo, que identifica o termo, *varOuFunc*, que define se é uma variável, função ou constante, *tipo*, que contém o tipo do termo, *valor*, que armazenará o valor atrelado a variável, *numArgs*, que armazena o número de argumentos da função, *escopo*, que identifica o escopo do termo, *linha* e *coluna*, que identificam o local onde foi declarado, e o ponteiro para o próximo elemento da lista. Foi também necessário criar uma lista que auxilia o processo de definir os escopos presentes nos programas. Nas figuras 1 e 2 são exemplificadas a árvore sintática e a tabela de símbolos de um exemplo.

### 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:



**Figura 1.** Árvore sintática abstrata gerada em um exemplo.

===== TABELA DE SIMBOLOS =====						
Símbolo	Var/Func/Const	Tipo	Num Args	Escopo	Linha	Coluna
oi	variavel	int	0	0	2	1
_write list	funcao	int	2	0	4	1
lista	variavel	int list	0	1	4	17
tamanho	variavel	int	0	1	4	33
i	variavel	int	0	1	5	5
header	variavel	int	0	1	6	5
" "	constante	string	0	2	11	15
soma	funcao	int	2	0	18	1

**Figura 2.** Tabela de símbolos gerada em um exemplo.

1. *teste1\_correto.ci*: programa sem erros sintáticos que recebe uma lista e retorna se a soma dos elementos contidos nesta lista é positiva ou negativa;
2. *teste2\_correto.ci*: programa sem erros sintáticos que monta em uma lista a sequência de *Fibonacci* até uma posição *n*;
3. *teste1\_errado.ci*: programa com erros sintáticos que calcula e apresenta a classificação do IMC de acordo com os dados inseridos. Os erros presentes são a falta de ponto e vírgula na linha 6 e coluna 5, o tipo errado na linha 21 e coluna 14, e problemas de atribuição na linha 27 e coluna 32, assim como na linha 32 e coluna 20;
4. *teste2\_errado.ci*: programa com erros sintáticos que encontra o menor número de ponto flutuante de uma lista. Os erros presentes são a função declarada com tipo errado na linha 1 e coluna 1, o número .9 após o número 9999.9 na linha 3 e coluna 18, o símbolo > na linha 17 e coluna 14 e o *for* com apenas uma expressão na linha 25 e coluna 18.

## 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
- Bison (GNU Bison) 3.7.5

Para compilar o programa no diretório do projeto é necessário executar os seguintes comandos:

```
bison -defines= ./src/sintaxe.tab.h -output= ./src/sintaxe.tab.c -v ./src/sintaxe.y
flex ./src/lexico.l
gcc ./src/sintaxe.tab.c ./src/lex.yy.c ./src/arvore.c ./src/lista.c -Wall -o
tradutor -lf
```

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 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.
- [Cor14] Robert Corbett. Gnu bison. <https://www.gnu.org/software/bison/>, 2014. [Accessed 02-September-2021].
- [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>	<i>if</i>	-
<i>else</i>	<i>else</i>	-
<i>for</i>	<i>for</i>	-
<i>return</i>	<i>return</i>	-
<i>read</i>	<b>ENTRADA</b>	-
<i>write</i>	<b>SAIDA</b>	-
<i>writeln</i>	<b>SAIDA</b>	-
Identificador	<b>ID</b>	Apontador para a entrada da tabela
Constante Int	<b>CONST_INT</b>	Apontador para a entrada da tabela
Constante Float	<b>CONST_FLOAT</b>	Apontador para a entrada da tabela
String	<b>STRING</b>	Apontador para a entrada da tabela
NIL	<b>NIL</b>	NIL
int	<b>TIPO</b>	int
float	<b>TIPO</b>	float
list	<b>LIST</b>	list
+	<b>ARIT_OP MAIS</b>	+
-	<b>ARIT_OP MENOS</b>	-
*	<b>ARITOP_ALTA</b>	*
/	<b>ARITOP_ALTA</b>	/
&&	<b>LOG_OP_E</b>	&&
	<b>LOG_OP_OU</b>	
!	<b>LOG_OP_UN</b>	!
<=	<b>RELOP_ALTA</b>	<=
<	<b>RELOP_ALTA</b>	<
>=	<b>RELOP_ALTA</b>	>=
>	<b>RELOP_ALTA</b>	>
==	<b>RELOP_BAIXA</b>	==
!=	<b>RELOP_BAIXA</b>	!=

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

Lexema	Nome do Token	Valor do Atributo
?	<b>LIST_OP_UN</b>	?
!	<b>LIST_OP_UN</b>	!
%	<b>LIST_OP_UN</b>	%
:	<b>LIST_OP_BIN</b>	:
>>	<b>LIST_OP_BIN</b>	>>
<<	<b>LIST_OP_BIN</b>	<<
(	(	-
)	)	-
{	{	-
}	}	-
,	<b>VIRG</b>	-
;	<b>PV</b>	-

## B Gramática

A gramática utilizada no tradutor da linguagem C-IPL é apresentada nas seguintes expressões:

1.  $program \rightarrow declarations \mid \varepsilon$
2.  $declarations \rightarrow declarations\ declaration \mid declaration$
3.  $declaration \rightarrow function \mid variable;$
4.  $function \rightarrow funcDecl\ (parameters)\ \{moreStmt\} \mid funcDecl\ ()\ \{moreStmt\}$
5.  $funcDecl \rightarrow \mathbf{TIPO\ ID} \mid \mathbf{TIPO\ LIST\ ID}$
6.  $parameters \rightarrow parameters,\ varDecl \mid varDecl$
7.  $moreStmt \rightarrow moreStmt\ block \mid block$
8.  $block \rightarrow stmt \mid \{moreStmt\}$
9.  $stmt \rightarrow conditional \mid iteration \mid varDecl; \mid attribution; \mid io; \mid ret;$
10.  $conditional \rightarrow \mathbf{IF}\ (attribution)\ bracesStmt \mid \mathbf{IF}\ (attribution)\ bracesStmt\ \mathbf{ELSE}\ bracesStmt\ \mathbf{ELSE}$
11.  $bracesStmt \rightarrow \{moreStmt\} \mid stmt$
12.  $iteration \rightarrow \mathbf{FOR}\ (iteArgs)\ bracesStmt$

13.  $iteArgs \rightarrow expIte; expIte; expIte$
14.  $expIte \rightarrow attribution \mid \varepsilon$
15.  $io \rightarrow \mathbf{READ}(\mathbf{ID}) \mid \mathbf{WRITE}(attribution) \mid \mathbf{WRITE}(\mathbf{STRING})$
16.  $varDecl \rightarrow \mathbf{TIPO ID} \mid \mathbf{TIPO LIST ID}$
17.  $attribution \rightarrow \mathbf{ID} = expLogic \mid expLogic$
18.  $expLogic \rightarrow expLogic \parallel andLogic \mid andLogic$
19.  $andLogic \rightarrow andLogic \&\& expComp \mid expComp$
20.  $expComp \rightarrow expComp == expRel \mid expComp != expRel \mid expRel$
21.  $expRel \rightarrow expRel < expArit \mid expRel > expArit \mid expRel <= expArit \mid expRel >= expArit \mid expArit$
22.  $expArit \rightarrow expArit + expMul \mid expArit - expMul \mid expMul$
23.  $expMul \rightarrow expMul * negElement \mid expMul / negElement \mid negElement$
24.  $negElement \rightarrow !expList \mid -expList \mid expList$
25.  $expList \rightarrow !element \mid ?element \mid \%element \mid expList << element \mid expList >> element \mid expList : element$
26.  $element \rightarrow \mathbf{ID} \mid (attribution) \mid \mathbf{ID}(arguments) \mid \mathbf{ID}() \mid \mathbf{CONST\_INT} \mid \mathbf{CONST\_FLOAT} \mid \mathbf{NIL}$
27.  $arguments \rightarrow arguments, attribution \mid attribution$
28.  $return \rightarrow \mathbf{RETURN} attribution$