

# Análise Léxica da Linguagem C-IPL

Lucas Dalle Rocha

17/0016641

Universidade de Brasília - UnB

170016641@aluno.unb.br

## 1 Motivação

A disciplina propõe a implementação das etapas de análise léxica, sintática, semântica e de geração de código intermediário para um subconjunto da linguagem C, denominado *C-IPL*, com o intuito de ressaltar a importância do processo de tradução de uma linguagem em alto nível para linguagem de máquina, bem como o impacto do desenvolvimento do projeto no sistema como um todo. Assim, pela linguagem *C-IPL* busca-se facilitar a manipulação de listas para programas escritos em C, visto que são utilizadas frequentemente para armazenamento de dados em sequência.

## 2 Descrição

### 2.1 Análise léxica

A etapa de análise léxica visa reconhecer e classificar lexemas válidos para dada linguagem, além de estabelecer a tabela de símbolos para satisfazer a comunicação com o analisador sintático.

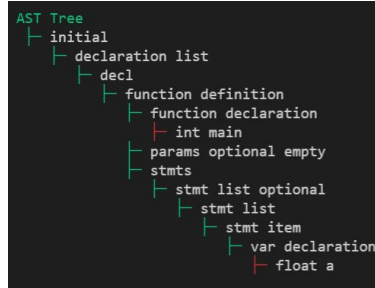
Dito isso, houve o uso de variáveis com a finalidade de exibir a linha e coluna em que o lexema analisado se encontra, bem como expressões regulares para buscar padrões que condizem com a descrição da linguagem *C-IPL*, de modo a atribuir *tokens* válidos para identificadores, dígitos, tipos de dado, palavras reservadas, números inteiros e reais, operadores aritméticos, relacionais, lógicos, e as operações sobre listas, implementadas pela linguagem *C-IPL*. Ademais, comentários não são tratados como lexemas, *strings* são sinalizadas como um único *token* e os símbolos de pontuação pareada também serão lexemas.

Dessa forma, quaisquer *lexemas* que não foram representados pelas expressões regulares são considerados erros léxico, assim como comentários multilinhas e *strings*, ambos não finalizados. Por fim, delimitadores como ponto e vírgula serão aprofundados na etapa de análise sintática, em conjunto com a tabela de símbolos, que será composta pelos atributos de identificadores (nome e tipo), além das suas respectivas posições em linha e coluna de código, e escopo.

## 2.2 Análise sintática

Já a etapa de análise sintática enfoca a sintaxe de uma gramática, isto é, a validade de uma sentença escrita na linguagem *C-IPL*. Para isso, foi-se utilizado o gerador de analisador sintático ascendente *Bison*, pelo padrão LR(1) canônico, de modo que o analisador léxico comunica-se com o analisador sintático através de *tokens* válidos da linguagem e, esse último, por sua vez, utiliza os *tokens* recebidos para esquematização de uma árvore sintática abstrata, que é a estrutura utilizada para validar a sintaxe em nossa gramática.

Desse modo, para cada regra válida da gramática, dado o conjunto de *tokens*, é criado um nó na estrutura da árvore, e para um terminal, é atribuído à estrutura da árvore esse conjunto de *tokens*, a fim de que possam ser impressos igualmente. Portanto, a estrutura utilizada para os nós da árvore possui o nome do próprio nó, ponteiros para seus nós filhos e, como supracitado, o conjunto de *tokens*.



**Figura 1.** Árvore sintática abstrata de *int main () {float a;}*, de acordo com o apêndice B.

Não obstante, a tabela de símbolos é uma estrutura implementada em conjunto das análises léxica e sintática, de modo que a primeira se ocupa em atribuir a posição do *token* em código (linha e coluna), além do escopo inicial, baseado nos *tokens* '{' e '}'. Já a análise sintática contribui para a construção da tabela de símbolos com a concatenação de *tokens* que especificam tipo e identificador, em nossa gramática, e a atualização do escopo, haja vista símbolos já existentes na tabela. Assim, um símbolo possui os atributos: nome, tipo, escopo, linha e coluna de ocorrência, e são armazenados dinamicamente em uma lista encadeada.

Symbol Table				
Name	Type	Line	Column	Scope
main	int	1	1	0
a	float	2	1	1

**Figura 2.** Tabela de símbolos de *int main () {float a;}*, de acordo com o apêndice B.

### 3 Testes

Existem dois arquivos de teste que se encontram corretos (representados pela cor verde abaixo), de acordo com a análise sintática, uma vez que não apresentam sintaxes irreconhecíveis pela gramática da linguagem *C-IPL*. Em contrapartida, existem outros dois arquivos de teste que possuem erros (representados pela cor vermelha abaixo), que serão listados abaixo. Todos os testes encontram-se na pasta *tests*.

```
17_0016641
└─ tests
    ├── correct1.c
    ├── correct2.c
    ├── incorrect1.c
    └── incorrect2.c
```

1. *incorrect1.c* apresenta erros de sintaxe não reconhecidas pela gramática da linguagem, como atribuição de valores dentro da função *if* (linha 2, coluna 6) - em que se espera uma expressão - e comando não finalizado com ponto e vírgula (linha 4, coluna 1), uma vez que se esperava o *token* `;`, e não `}`.
2. *incorrect2.c* similarmente possui sintaxe não reconhecidas pela *C-IPL*, tais como função de escrita atribuída a função de lista *map* `'>>'` (linha 7, coluna 16) - em que se espera uma expressão matemática de soma, no mínimo - e atribuição de valores dentro da função *for* (linha 10, coluna 21), uma vez que se esperava uma expressão simples.

### 4 Instruções para compilação e execução

Para a compilação e execução, foi utilizado o sistema operacional Ubuntu 20.04.2 LTS, além do *gcc* versão 11.1.0, *flex* versão 2.6.4, *bison* versão 3.7.4 e GNU Make 4.2.1. O programa pode ser compilado pela utilização do *Makefile*, com o seguinte comando em seu terminal:

```
$ make
```

Ademais, pode-se compilar e executar o *Valgrind* para checagem de memória dos arquivos de teste, em conjunto com a utilização do *Makefile*:

```
correct1.c  -> $ make valgrind1
correct2.c  -> $ make valgrind2
incorrect1.c -> $ make valgrind3
incorrect2.c -> $ make valgrind4
```

Caso seja de seu interesse compilá-lo manualmente, execute a sequência de instruções:

```
$ bison -o src/syntax.tab.c -d src/syntax.y -Wcounterexamples
$ flex -o src/lex.yy.c src/lexical.l
$ gcc-11 -g -c src/structures.c -o obj/structures.o
$ gcc-11 -g src/syntax.tab.c src/lex.yy.c obj/structures.o
-I lib -o tradutor -Wall -Wpedantic
```

Por fim, para execução do analisador sintático nos arquivos teste:

```
$ ./tradutor tests/<file>.c
```

## Referências

- [ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.
- [Hec21] Robert Heckendorn. A grammar for the C- programming language. <http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf>, 2021. [Online; Last accessed 19-August-2021].
- [Lee85] Jeff Lee. ANSI C Yacc grammar. <https://www.lysator.liu.se/c/ANSI-C-grammar-y.html>, 1985. [Online; Last accessed 2-September-2021].
- [LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., 2nd edition, 1992.
- [PEM16] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex, for flex 2.6.2. <https://westes.github.io/flex/manual/>, 2016. [Online; Last accessed 19-August-2021].

## A Estruturação do *token*

Para estruturação do *token*, isto é, seus padrões, nomes e atributos, foi utilizado como referência a figura 3.12 do livro-texto [ALSU06], abordado na página 130, e atualizado para a análise sintática.

Lexema	Nome do <i>token</i>	Atributo
Espaços em branco	–	–
Qualquer <i>id</i>	<b>ID</b>	Ponteiro p/ tabela de símbolos
Qualquer <i>integer_number</i>	<b>INTEGER</b>	Ponteiro p/ tabela de símbolos
Qualquer <i>real_number</i>	<b>REAL</b>	Ponteiro p/ tabela de símbolos
if	<b>IF</b>	–
else	<b>ELSE</b>	–
for	<b>FOR</b>	–
return	<b>RETURN</b>	–
read	<b>INPUT</b>	–
write, writeln	<b>OUTPUT</b>	–
NIL	<b>NIL_CONSTANT</b>	–
int, float	<b>SIMPLE_TYPE</b>	–
list	<b>COMPOUND_TYPE</b>	–
<, <=, >, >=, ==, !=	<b>RELATIONAL</b>	–
+, -	<b>ADD</b>	–
*, /	<b>MUL</b>	–
	<b>LOGICAL_OR</b>	–
&&	<b>LOGICAL_AND</b>	–
:	<b>LIST_CONSTR</b>	–
?	<b>LIST_OP</b>	–
%	<b>LIST_DESTR</b>	–
>>, <<	<b>LIST_FUNC</b>	–
!	<b>OP_OVERLOAD</b>	–

**Tabela 1.** Padrões de *tokens*, seus nomes e atributos.

## B Gramática da linguagem

Para a montagem da gramática, foi utilizado como referência [Hec21], a fim de estimar a transição da análise léxica para a sintática. As adaptações foram feitas para que a gramática seja compatível com a linguagem *C-IPL* e correções para a análise sintática.

1.  $initial \rightarrow declaration\_list$
2.  $declaration\_list \rightarrow declaration\_list\ decl \mid decl$
3.  $decl \rightarrow var\_declaration \mid func\_definition$
4.  $var\_declaration \rightarrow \mathbf{SIMPLE\_TYPE}\ ID; \mid \mathbf{SIMPLE\_TYPE}\ \mathbf{COMPOUND\_TYPE}\ ID;$
5.  $var\_definition \rightarrow \mathbf{SIMPLE\_TYPE}\ ID \mid \mathbf{SIMPLE\_TYPE}\ \mathbf{COMPOUND\_TYPE}\ ID$
6.  $func\_definition \rightarrow func\_declaration\ (params.opt)\ stmts$
7.  $func\_declaration \rightarrow \mathbf{SIMPLE\_TYPE}\ ID \mid \mathbf{SIMPLE\_TYPE}\ \mathbf{COMPOUND\_TYPE}\ ID$
8.  $params.opt \rightarrow params \mid \epsilon$
9.  $params \rightarrow param\_list$
10.  $param\_list \rightarrow param\_list,\ param \mid param$
11.  $param \rightarrow var\_definition$
12.  $stmts \rightarrow \{stmt\_list.opt\}$
13.  $stmt\_list.opt \rightarrow stmt\_list \mid \epsilon$
14.  $stmt\_list \rightarrow stmt\_list\ stmt.item \mid stmt.item$
15.  $stmt.item \rightarrow stmt \mid var\_declaration$
16.  $stmt \rightarrow return\_stmt \mid select\_stmt \mid iter\_stmt \mid io\_stmt \mid exp\_stmt \mid stmts$
17.  $select\_stmt \rightarrow \mathbf{IF}\ (simple\_exp)\ stmt \mid \mathbf{IF}\ (simple\_exp)\ stmt\ \mathbf{ELSE}\ stmt$
18.  $iter\_stmt \rightarrow \mathbf{FOR}\ (assign\_exp;\ simple\_exp;\ exp)\ stmt$
19.  $exp\_stmt \rightarrow assign\_exp \mid simple\_exp$
20.  $return\_stmt \rightarrow \mathbf{RETURN}\ exp;$
21.  $io\_stmt \rightarrow in\_stmt \mid out\_stmt$
22.  $in\_stmt \rightarrow \mathbf{INPUT}\ (ID);$
23.  $out\_stmt \rightarrow \mathbf{OUTPUT}\ (STRING); \mid \mathbf{OUTPUT}\ (exp);$
24.  $assign\_exp \rightarrow ID = exp$
25.  $simple\_exp \rightarrow simple\_exp\ \mathbf{LOGICAL\_OR}\ and\_exp \mid and\_exp$
26.  $and\_exp \rightarrow and\_exp\ \mathbf{LOGICAL\_AND}\ rel\_exp \mid rel\_exp$
27.  $rel\_exp \rightarrow rel\_exp\ \mathbf{RELATIONAL}\ list\_exp \mid list\_exp$
28.  $list\_exp \rightarrow sum\_exp\ \mathbf{LIST\_CONSTR}\ list\_exp \mid sum\_exp\ \mathbf{LIST\_FUNC}\ list\_exp \mid sum\_exp$
29.  $sum\_exp \rightarrow sum\_exp\ \mathbf{ADD}\ mul\_exp \mid mul\_exp$
30.  $mul\_exp \rightarrow mul\_exp\ \mathbf{MUL}\ unary\_exp \mid unary\_exp$
31.  $unary\_exp \rightarrow factor \mid \mathbf{LIST\_OP}\ unary\_exp \mid \mathbf{LIST\_DESTR}\ unary\_exp \mid \mathbf{OP\_OVERLOAD}\ unary\_exp \mid \mathbf{ADD}\ unary\_exp$
32.  $factor \rightarrow ID \mid (simple\_exp) \mid func\_call \mid constant$
33.  $func\_call \rightarrow ID\ (func\_params)$
34.  $func\_params \rightarrow simple\_exp \mid func\_params,\ simple\_exp \mid \epsilon$
35.  $constant \rightarrow \mathbf{INTEGER} \mid \mathbf{REAL} \mid \mathbf{NIL\_CONSTANT}$

## C Léxico da linguagem

Para a esquematização do léxico em expressões regulares, foi utilizado como referência o livro a respeito do analisador léxico [LMB92], bem como o manual do flex [PEM16], disponibilizado *online*.

Macro do token	Definição <i>regex</i>
comment_line	"//".*
delim	[ \t\r]
ws	[delim]+
newline	[\n]
brace_opening & brace_closing	"{" "}"
parenthese_opening & parenthese_closing	"(" ")"
letter	[A-Za-z_]
digit	[0-9]
keywords	"if" "else" "for" "return"
data_type	"int list" "int" "float list" "float"
input_command	"read"
output_command	"write" "writeln"
nil_constant	"NIL"
id	{letter}({letter} {digit})*
integer_number	-?{digit}+
real_number	-?{integer_number}(\.{digit}+)(E[+-]?{digit}+)?
arithmetic_operators	"+" "-" "*" "/"
relational_operators	"<" "<=" ">" ">=" "==" "!="
logical_operators	"  " "&&"
assignment	"="
separator	" "
flow_control	"."
list_constructor	"."
list_operator	"?"
list_destructor	"%"
list_functions	">>" "<<"
string_literal	\ "([^\n\\] \\.)*\\"
operator_overload	!

**Tabela 2.** Tabela de lexemas e suas regras.