

# Analizador Sintático - Trabalho Prático

João Lucas Azevedo Yamin Rodrigues da Cunha - 17/0013731

Universidade de Brasília  
Brasília - DF, CEP 70910-900  
jlyaminc@gmail.com

## 1 Motivação

A disciplina de Tradutores é uma das últimas da cadeia obrigatória do curso de Ciência da Computação. Ela aborda, em sua ementa, conteúdos de diversos períodos do curso. Por isso, a implementação do analisador léxico, bem como os demais componentes do projeto da disciplina a serem implementados futuramente, permite a assimilação da união do aprendizado acumulado ao longo do curso.

Para este projeto será utilizada a linguagem C-IPL, um subconjunto da linguagem C com adição de tratamento de listas. Dentro da computação, a lista é uma das estruturas de dados mais utilizadas e a adição dessa primitiva à linguagem já as disponibiliza de forma simples ao usuário, facilitando a programação. Elas possuem diversas aplicações práticas dentro e fora da matemática, além de servir como base para a implementação de outros conceitos.

## 2 Descrição da Análise Léxica

Para montagem do analisador léxico foi utilizada a ferramenta de geração de analisador léxico Flex. Redigiu-se um arquivo com extensão .l que serve de entrada para a ferramenta, descrevendo como deve ser feita a geração do analisador. Nele foram adicionadas diversas capturas de caracteres por meio de expressões regulares.

Os lexemas reconhecidos foram enviados como *tokens* para a etapa de análise sintática, com atributos adicionais de linha e coluna. As sequências de caracteres não identificadas são impressas na tela como erros léxicos. Também no escaneamento léxico, foi realizada uma implementação parcial de escopo: a cada { encontrado, é incrementado um contador. Este serve como identificador para referenciar o escopo dos símbolos na tabela de símbolos. Posteriormente será desenvolvida uma estrutura de árvore para determinar a hierarquia entre os escopos.

No anexo, a tabela da Seção B apresenta a descrição dos *tokens* utilizados.

## 3 Descrição da Análise Sintática

A partir da gramática presente na Seção A do anexo e por meio da ferramenta Bison, implementou-se um analisador sintático. Integrado com o analisador léxico,

o sintático identifica quaisquer erros sintáticos presentes no código analisado e, com o auxílio de funções adicionais implementadas, permite a criação de uma árvore sintática abstrata e de uma tabela de símbolos simples.

A árvore sintática foi implementada utilizando *structs* compostas por um *token* (caso seja um nó-folha), um nome e um vetor de filhos (ambos caso seja um nó-pai) para representar os elementos. Os nós são adicionados quando se identificam regras da gramática que possuem terminais (nós-folha) ou ramificações (nós-pai). Sua impressão é feita de forma recursiva.

No caso da tabela de símbolos, a inserção é feita sempre que uma declaração de variável ou de função é encontrada. Uma entrada com identificador, tipo, número de escopo e um indicador de função é inserida nesses casos. A tabela de símbolos foi construída utilizando uma lista encadeada.

Ao fim da análise sintática são impressos: os erros encontrados, a árvore sintática e a tabela de símbolos, nesta ordem.

## 4 Arquivos de Teste

Dentro do diretório `tests/`, foram criados dois arquivos de teste corretos:

1. `correct_test_01.c`;
2. `correct_test_02.c`, .

E dois arquivos de teste contendo erros:

1. `incorrect_test_01.c`:
  - Erro léxico: Caracter `@` encontrado na linha 1, coluna 7;
  - Erro sintático: Atribuição junto de declaração na linha 1, coluna 10; e
  - Erro sintático: Falta de ponto e vírgula na linha 4, coluna 1.
2. `incorrect_test_02.c`:
  - Erro sintático: Declaração de variável com número antes do identificador na linha 1, coluna 7;
  - Erro léxico: Caracter `^` encontrado na linha 5, coluna 14; e
  - Erro sintático: Expressão de soma dentro de `writeln` na linha 6, coluna 24.

## 5 Compilação e Organização

O código referente a este projeto foi desenvolvido e executado em um sistema com as seguintes características:

- Sistema Operacional: Manjaro 21.10;
- Kernel: 5.10 LTS;
- gcc: 11.1.0;
- ld: 2.36.1;
- flex: 2.6.4;
- make: 4.3.

Para execução o analisador léxico e sintático, o comando

```
$ make tradutor
```

compilará o código e gerará um executável de nome ‘tradutor’. Este comando é equivalente a executar

```
$ bison -d ./src/cipl_syn.y -Wcounterexamples
$ flex ./src/cipl_lex.l
$ gcc -c ./src/symbol.c
$ gcc -c ./src/tree.c
$ gcc -o tradutor cipl_syn.tab.c lex.yy.c symbol.o tree.o -g
-Wall -Wextra -Wpedantic
```

Com isto, execute utilizando

```
$ ./tradutor <caminho_para_o_arquivo>
```

Como dito acima, os testes se encontram no diretório `./tests/`.

## Referências

- [ALSU06] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2006.
- [Fou] Free Software Foundation. GNU Bison - The Yacc-compatible Parser Generator. <https://www.gnu.org/software/bison/manual/>. Acessado por último em 1 Set 2021.
- [Jon] D. Jones. The New C Standard. <http://www.coding-guidelines.com/cbook/cbook1.1.pdf>. Acessado por último em 18 Ago 2021.
- [Nac] V. Nachiappan. USING LEX. <https://silcnitc.github.io/lex.html>. Acessado por último em 10 Ago 2021.
- [Pol] B. Pollack. BNF Grammar for C-Minus. <http://www.csci-snc.com/ExamplesX/C-Syntax.pdf>. Acessado por último em 10 Ago 2021.
- [SK] K. Slonneger and B Kurtz. Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach. <https://homepage.divms.uiowa.edu/~slonnegr/plf/Book/Chapter1.pdf>. Acessado por último em 2 Set 2021.

## A Gramática

A Gramática a seguir foi gerada utilizando como base a gramática C-Minus [Pol]. Os rótulos em letras todas maiúsculas são equivalentes aos apresentados na tabela B.

1.  $program \rightarrow declarationList$
2.  $declarationList \rightarrow declarationList\ declaration \mid declaration$

3.  $declaration \rightarrow variableDeclaration \mid functionDeclaration$
4.  $variableDeclaration \rightarrow \mathbf{TYPE\ ID} ;$
5.  $functionDeclaration \rightarrow \mathbf{TYPE\ ID} ( params )$   
 $compoundStmt$
6.  $params \rightarrow paramList \mid \epsilon$
7.  $paramList \rightarrow paramList , param \mid param$
8.  $param \rightarrow \mathbf{TYPE\ ID}$
9.  $compoundStmt \rightarrow \{ statementList \}$
10.  $statementList \rightarrow statementList statement \mid \epsilon$
11.  $statement \rightarrow expressionStmt \mid compoundStmt \mid conditionalStmt \mid$   
 $loopStmt \mid returnStmt \mid variableDeclaration \mid inOutStmt$
12.  $expressionStmt \rightarrow expression ; ;$
13.  $conditionalStmt \rightarrow \mathbf{if} ( expression ) statement \mid$   
 $\mathbf{if} ( expression ) statement \mathbf{else} statement$
14.  $loopStmt \rightarrow \mathbf{for} ( expression ; logicExpression ; expression ) statement$
15.  $returnStmt \rightarrow \mathbf{return} expression ;$
16.  $inOutStmt \rightarrow \mathbf{INPUT} ( \mathbf{ID} ); \mid \mathbf{OUTPUT} ( outputArgs );$
17.  $expression \rightarrow \mathbf{ID} = expression \mid logicExpression$
18.  $logicExpression \rightarrow logicExpression \mathbf{OP\_LOGIC} relatExpression$   
 $\mid relatExpression$
19.  $relatExpression \rightarrow relatExpression \mathbf{OP\_RELAT} listExpression$   
 $\mid listExpression$
20.  $listExpression \rightarrow addExpression \mathbf{OP\_LIST} listExpression$   
 $\mid addExpression$
21.  $addExpression \rightarrow addExpression \mathbf{OP\_ADD} mulExpression$   
 $\mid mulExpression$
22.  $mulExpression \rightarrow mulExpression \mathbf{OP\_MUL} factor$   
 $\mid factor$
23.  $factor \rightarrow ( expression ) \mid unaryExpression \mid call \mid \mathbf{ID} \mid$   
 $\mathbf{FLOAT} \mid \mathbf{INT} \mid \mathbf{NIL}$
24.  $unaryExpression \rightarrow \mathbf{UN\_OP} factor \mid \mathbf{OP\_ADD} factor$
25.  $call \rightarrow \mathbf{ID} ( args )$
26.  $outputArgs \rightarrow \mathbf{STRING} \mid factor$
27.  $args \rightarrow argList \mid \epsilon$
28.  $argList \rightarrow argList , expression \mid expression$

## B Tokens e Lexemas

| Rótulo do Token | Padrão do Lexema (RegEx)                                     | Lexema de Exemplo |
|-----------------|--|-------------------|
| ID              | <code>[_a-zA-Z][_a-zA-Z0-9]*</code>                          | num               |
| DIGIT           | <code>[0-9]</code>   | 88                |
| FLOAT           | <code>(-)?{DIGIT}*.{DIGIT}+</code>                           | -402.3            |
| INT             | <code>(-)?{DIGIT}+</code>                                    | 25                |
| OP_ADD          | <code>[+-]</code>  | +                 |
| OP_MUL          | <code>[*/]</code>  | /                 |
| OP_LOGIC        | <code>(&amp;&amp;) (  )</code>                               | &&                |
| OP_RELAT        | <code>(&lt;) (&lt;=) (&gt;) (&gt;=) (&amp;=) (&amp;=)</code> | >=                |
| OP_ASSIG        | <code>(=)</code>   | =                 |
| UN_OP           | <code>(!) (%) (?)</code>                                     | !                 |
| OP_LIST         | <code>(&gt;&gt;) (&lt;&lt;) (:)</code>                       | >>                |
| TYPE            | <code>(int) (float) (int list) (float list)</code>           | int list          |
| NIL             | <code>(NIL)</code>   | NIL               |
| IF              | <code>(if)</code>  | if                |
| ELSE            | <code>(else)</code>  | else              |
| FOR             | <code>(for)</code>   | for               |
| RETURN          | <code>(return)</code>  | return            |
| INPUT           | <code>(read)</code>  | read              |
| OUTPUT          | <code>(write) (writeln)</code>                               | writeln           |
| SEMICOLON       | <code>(;)</code>   | ;                 |
| COMMA           | <code>(,)</code>   | ,                 |
| CURLYB          | <code>[{ }]</code>   | {                 |
| PARENTHESIS     | <code>[ ( ) ]</code>   | (                 |
| STRING          | <code>(" .*") ( ' .*')</code>                                | "string"          |

**Tabela 1.** Tokens e lexemas de exemplo