

# Analizador Semântico

Lucas Mafra Chagas<sup>[12/0126443]</sup>

Prof.<sup>a</sup> Dra. Cláudia Nalon

**Resumo** Esse projeto tem como intuito construir um analisador sintático para um subconjunto da linguagem C, mostrando sua nova primitiva de linguagem, sua descrição, instrução e gramática.

**Keywords:** Tradutores · Compiladores · Mini-C.

## 1 Objetivo

Tendo como base a metodologia de ensino baseada em projeto, determinou-se algumas diretrizes que o discente da Ciência da Computação necessita seguir para alcançar um melhor entendimento do conteúdo apresentado. Dessa forma, as diretrizes definidas consistem na implementação de um analisador de um subconjunto da linguagem C. Por fim, este projeto é constituído por quatro etapas: Implementação do analisador léxico; Implementação do analisador sintático; Implementação do analisador semântico; e Implementação do gerador de código intermediário.

## 2 Motivação

Um compilador é um programa que lê um programa escrito numa linguagem, conhecida como linguagem fonte, e o traduz num programa equivalente numa outra linguagem, conhecida como linguagem alvo. Essa ação é feita observando possíveis erros presentes no programa fonte[1].

Sendo assim, a linguagem projetada para o desenvolvimento do trabalho final segue as tratativas básicas da linguagem C e a adição de uma nova primitiva de dados, que é declarado como *set*. A declaração de uma variável desse tipo segue o molde de C, sem ter um tipo associado aos seus elementos, onde o *set* tem como papel ser composto por *elem*, onde *elem* é polimórfico, podendo assumir o papel de *int*, *float* ou de *set*[5].

Além disso, o novo tipo de dado da linguagem, o *set* possui operações específicas, como verificação de tipagem, *is\_set*, a inclusão de elemento em um conjunto, *add*, a remoção de elemento de um conjunto, *remove*, a seleção dentro desse conjunto, *exists*, e a iteração dentro desse conjunto, *forall*[5].

### 3 Descrição da Análise Léxica

A análise léxica é a primeira fase de um projeto de compilador. O analisador léxico é responsável por converter uma sequência de caracteres em uma sequência de lexemas, removendo qualquer comentário ou espaço em branco. O analisador léxico é responsável por varrer o código fonte do programa e identificar cada token, um por um[6]. Durante essa fase, para uma melhor compreensão da linguagem desenvolvida, é necessário construir uma gramática livre de contexto que determina o seu funcionamento. Além da gramática, é necessário ter uma tabela de símbolos classificando os nomes das variáveis, as funções, as classes e as constantes[4].

No projeto desenvolvido, além das regras e expressões regulares, o analisador léxico necessitou da estruturação de algumas funcionalidades para a preparação do algoritmo. Essas novas estruturas servem de base para a construção da tabela de símbolos na próxima etapa. Para o tratamento de erros, foi construída uma função que destaca um token inválido ao ser encontrado, mostrando a linha e a coluna onde esse erro aconteceu. Foram adicionadas também sub-rotinas auxiliares para impressão colorida de texto e controle de linhas e colunas lidas.

### 4 Descrição da Análise Sintática

A análise sintática é a segunda fase de um projeto de compilador. O analisador sintático é responsável por apanhar os lexemas identificados pelo analisador léxico e descrevê-los de forma adequada aos seus programas[2]. Os tokens, que vem agrupados nos lexemas, possuem dois componentes: os seus nomes e seus valores de atributos. Os nomes dos tokens aparecem como símbolos terminais da gramática da linguagem, e seus atributos direcionam para a tabela de símbolos que possuem mais informações sobre os tokens[1].

Nessa segunda etapa, além da revisão do analisador léxico, foi necessário construir uma árvore sintática, que representa a estrutura sintática hierárquica do programa, e a estrutura da tabela de símbolos, que são estruturas de dados utilizados pelo compilador para obter informações sobre as construções do programa-fonte.

### 5 Descrição da Análise Semântica

A análise semântica é a terceira fase de um projeto de compilador. O analisador semântico é responsável por coletar as informações dos tokens e colocar em tabela de símbolos, performar verificação de tipo, verificação de etiqueta e controle de fluxo. O analisador semântico produz algum tipo de representação do programa, seja o código do objeto ou uma representação intermediária do programa[2]. [1].

Nessa terceira etapa, além da revisão do analisador sintático, foi necessário otimizar a tabela de símbolos, para fazer a verificação de tipo, de etiqueta e um controle melhor do fluxo. Além disso, o programa foi otimizado para um compilador de duas etapas. Para isso, o código foi construído para construção da árvore abstrata apenas ao fim das passagens de leitura do código.

## 6 Descrição dos Arquivos de Teste

Os arquivos de teste buscam apresentar códigos sem erros e códigos que possuem algum erro para o analisador sintático. Os códigos inválidos buscam mostrar erros de um caractere ou string colocado incorretamente em um comando ou instrução que causa uma falha na execução, desrespeitando a sintaxe do programa.

No código `incorreto1.mf`, o código apresenta o erro de não ter uma chave para fechar o laço `for`, iniciado na linha 4.

No código `incorreto2.mf`, o código apresenta o erro de não ter um ponto e vírgula na chamada da função `contador`, na linha 10.

## 7 Instruções para Compilação

Para compilar, é necessário executar os seguintes comandos:

1. `$ make`
2. `$ ./mafralang tests/corretoX.mf`, onde X representa o número do exemplo, para rodar os códigos corretos;
3. `$ ./mafralang tests/incorreto.mf`, onde X representa o número do exemplo, para rodar os códigos incorretos.

Depois de compilar o código, ele gerará no próprio terminal a árvore sintática abstrata e a tabela de símbolos. Os erros sintáticos apareceram no terminal, informando a linha onde ocorreu.

## Referências

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
2. Grune, D., Jacobs, C.J.H.: *Parsing Techniques: A Practical Guide*. Ellis Horwood, USA (1990)
3. Kernighan, B.W., Ritchie, D.M.: *The C programming language*. Prentice hall of India (1973)
4. Loudon, K.C.: *Compiler construction*. Cengage Learning (1997)
5. Nalon, C.: Trabalho prático - descrição da linguagem (2021), <https://aprender3.unb.br/mod/page/view.php?id=294131>, visitou em 26-02-2021
6. Ritchie, D.M., Kernighan, B.W., Lesk, M.E.: *The C programming language*. Prentice Hall Englewood Cliffs (1988)

## A Gramática

A gramática foi feita a partir das adaptações da gramática apresentada por Brian W. Kernighan e Dennis M. Ritchie no livro K&R[3]. Os terminais são apresentados entre aspas simples e em caixa alta, enquanto as variáveis são apresentadas entre parênteses.

|  |   |
|--|---|
| $\langle program \rangle$              | $::= \langle declarations \rangle$  |
| $\langle declarations \rangle$         | $::= \langle declarations \rangle \langle declaration \rangle$<br>  $\langle declaration \rangle$   |
| $\langle declaration \rangle$          | $::= \langle var\_declaration \rangle$<br>  $\langle func\_declaration \rangle$   |
| $\langle var\_declaration \rangle$     | $::= 'TYPE' 'ID' ';'$   |
| $\langle func\_declaration \rangle$    | $::= 'TYPE' 'ID' '(' \langle paramaters \rangle ')' \langle compound\_statement \rangle$  |
| $\langle paramaters \rangle$           | $::= \langle parameters \rangle ',' \langle paramater \rangle$<br>  $\langle parameter \rangle$<br>   |
| $\langle paramater \rangle$            | $::= 'TYPE' 'ID'$   |
| $\langle block\_struct \rangle$        | $::= \langle compound\_statement \rangle$<br>  $\langle block\_statement \rangle$   |
| $\langle coumpound\_statement \rangle$ | $::= '{' \langle statement\_list \rangle '}'$   |
| $\langle statement\_list \rangle$      | $::= \langle statement\_list \rangle \langle block\_statement \rangle$<br>  $\langle block\_statement \rangle$  |
| $\langle block\_statement \rangle$     | $::= \langle var\_declaration \rangle$<br>  $\langle set\_statement \rangle ';'$<br>  $\langle statement \rangle$<br>  $\langle return\_statement \rangle$<br>  $\langle input\_statement \rangle ';'$<br>  $\langle output\_statement \rangle ';'$<br>  $\langle function\_call \rangle ';'$ |

|  |  |
|--|--|
| $\langle function\_call \rangle$         | $::= 'ID' '(' \langle local\_parameters \rangle ')'$   |
| $\langle local\_parameters \rangle$      | $::= \langle local\_parameters \rangle ',' \langle local\_parameter \rangle$<br>$  \langle local\_parameter \rangle$   |
| $\langle local\_parameter \rangle$       | $::= 'ID'$   |
| $\langle input\_statement \rangle$       | $::= 'READ' '(' \langle expression \rangle ')'$  |
| $\langle output\_statement \rangle$      | $::= 'WRITE' '(' \langle expression \rangle ')'$<br>$  'WRITELN' '(' \langle expression \rangle ')'$   |
| $\langle return\_statement \rangle$      | $::= 'RETURN' '{' \langle expression \rangle '}' ';' ;$<br>$  'RETURN' \langle expression \rangle ';' ;$   |
| $\langle statement \rangle$              | $::= \langle expression\_statement \rangle$<br>$  \langle conditional\_statement \rangle$<br>$  \langle iteration\_statement \rangle$  |
| $\langle expression\_statement \rangle$  | $::= \langle expression \rangle ';' ;$   |
| $\langle conditional\_statement \rangle$ | $::= 'IF' '(' \langle expression \rangle ') ' \langle block\_struct \rangle$<br>$  'IF' '(' \langle expression \rangle ') ' \langle block\_struct \rangle 'else' \langle block\_struct \rangle$                                    |
| $\langle iteration\_statement \rangle$   | $::= 'FOR' '(' \langle expression \rangle ';' \langle expression \rangle ';' \langle expression \rangle ') ' \langle block\_struct \rangle$<br>$  'FORALL' '(' \langle set\_expression \rangle ') ' \langle block\_struct \rangle$ |
| $\langle set\_statement \rangle$         | $::= \langle is\_set\_statement \rangle$<br>$  \langle remove\_statement \rangle$<br>$  \langle add\_statement \rangle$<br>$  \langle exists\_statement \rangle$   |
| $\langle is\_set\_statement \rangle$     | $::= 'IS\_SET' '(' 'ID' ')'$   |
| $\langle remove\_statement \rangle$      | $::= 'REMOVE' '(' \langle set\_expression \rangle ')'$   |

|   |  |
|---|--|
| $\langle add\_statement \rangle$        | $::= 'ADD' '(' \langle set\_expression \rangle ')'$  |
| $\langle exists\_statement \rangle$     | $::= 'EXISTS' '(' \langle set\_expression \rangle ')'$   |
| $\langle set\_expression \rangle$       | $::= \langle expression \rangle 'IN' \langle expression \rangle$   |
| $\langle expression \rangle$            | $::= \langle operation \rangle$<br>$  \langle expression \rangle ',' \langle operation \rangle$<br>$  \langle function\_expression \rangle$  |
| $\langle function\_expression \rangle$  | $::= \langle set\_statement \rangle$<br>$  \langle function\_call \rangle$   |
| $\langle operation \rangle$             | $::= \langle arithmetic\_operation \rangle$<br>$  \langle variable \rangle '=' \langle operation \rangle$  |
| $\langle arithmetic\_operation \rangle$ | $::= \langle logical\_operation \rangle$<br>$  \langle arithmetic\_operation \rangle '+' \langle logical\_operation \rangle$<br>$  \langle arithmetic\_operation \rangle '-' \langle logical\_operation \rangle$<br>$  \langle arithmetic\_operation \rangle '*' \langle logical\_operation \rangle$<br>$  \langle arithmetic\_operation \rangle '/' \langle logical\_operation \rangle$   |
| $\langle logic\_operation \rangle$      | $::= \langle relational\_operation \rangle$<br>$  '!' \langle logic\_operation \rangle$<br>$  \langle relational\_operation \rangle '&&' \langle logic\_operation \rangle$<br>$  \langle relational\_operation \rangle '  ' \langle logic\_operation \rangle$  |
| $\langle relational\_operation \rangle$ | $::= \langle variable \rangle$<br>$  \langle variable \rangle '==' \langle relational\_operation \rangle$<br>$  \langle variable \rangle '<' \langle relational\_operation \rangle$<br>$  \langle variable \rangle '\leq' \langle relational\_operation \rangle$<br>$  \langle variable \rangle '>' \langle relational\_operation \rangle$<br>$  \langle variable \rangle '\geq' \langle relational\_operation \rangle$<br>$  \langle variable \rangle '!=' \langle relational\_operation \rangle$ |
| $\langle variable \rangle$              | $::= \langle constant \rangle$<br>$  ``\langle string \rangle ``$<br>$  'ID'$  |

$$\langle constant \rangle ::= \text{'INTEGER'}$$
$$| \text{'FLOAT'}$$
$$| \text{'EMPTY'}$$