

# Tradutores - Analisador Léxico e Sintático

Marcus Vinícius da Silva Borges<sup>[130032891]</sup>

Departamento de Ciência da Computação, Universidade de Brasília  
130032891@aluno.unb.br  
cic.unb.br

## 1 Motivação

Os conceitos de teoria de conjuntos são muito importantes na matemática contemporânea [2], mas não se limitam somente a essa área. Esses conceitos são base dentro da Ciência da Computação já que a Teoria da Computação tem bastante de suas definições baseadas em conjuntos como, por exemplo, as máquinas de Turing e os autômatos finitos.

Este trabalho propõe a construção de um compilador para a linguagem Bemolang, que é um subconjunto da linguagem C com a adição de primitivas que dão suporte para o tratamento de conjuntos matemáticos. Nessa etapa do trabalho o foco é uma dar uma breve descrição sobre a linguagem, a implementação de seu analisador léxico e a implementação de seu analisador sintático.

## 2 Primitivas da Linguagem

Esta seção descreve brevemente as novas primitivas adicionadas a linguagem Bemolang. Todas as primitivas extras foram criadas para facilitar o tratamento de conjuntos matemáticos nessa linguagem e sua gramática completa está descrita no Anexo A.

Para trabalhar com conjuntos a Bemolang disponibiliza as primitivas do tipo **set** e **elem**. O **set** é um tipo que representa um conjunto matemático e o **elem** representa uma variável polimórfica, ou seja, pode assumir a forma dos outros três tipos disponíveis na linguagem: **set**, **int** ou **float**. Além disso, a constante **EMPTY** representa o conjunto vazio nessa linguagem.

Na Bemolang os operadores responsáveis por verificar a pertinência de um elemento em um conjunto são: **in** e **exists**. Os operadores responsáveis pela adição e remoção de um elemento em um **set** são **add** e **remove**, respectivamente. Além disso, o operador **is\_set** é responsável por verificar se uma variável do tipo **elem** é um **set**.

Para facilitar a iteração entre elementos de um **set** pode-se usar o comando **forall**. Por fim, os comandos responsáveis por lidar com entrada e saída da linguagem são: **read**, **write** e **writeln**.

### 3 Análise Léxica

A principal tarefa de um analisador léxico é ler os caracteres de entrada de um programa fonte, agrupá-los em lexemas e produzir como saída uma sequência de *tokens* para cada lexema no programa fonte [1]. Nessa etapa do projeto é papel do analisador léxico informar para o usuário sobre erros léxicos num dado programa fonte.

O Flex [3] é um programa criado por Vern Paxson para geração de analisadores léxicos e foi a ferramenta utilizada para gerar o analisador léxico deste trabalho. Dentre os arquivos desta entrega, o `bemolang.1` é o arquivo utilizado pelo Flex para criar o analisador léxico, onde são definidas as regras de casamento de padrão entre os caracteres de um programa e a geração de tokens para cada lexema encontrado.

#### 3.1 Implementação do Analisador Léxico

Funções extras foram criadas para serem utilizadas durante o processo de análise léxica, são essas: as funções `count` e `handle_unrecognized_symbol`. A `count` é responsável pela contagem das linhas e colunas de um programa, enquanto que a `handle_unrecognized_symbol` é chamada sempre que um símbolo não definido na linguagem é encontrado para gerar uma mensagem de erro informativa para o usuário.

#### 3.2 Testes de Lexema

Arquivos de testes foram disponibilizados junto a este trabalho para validar o funcionamento do analisador léxico gerado. Os arquivos de testes estão na pasta `tests` e seus nomes indicam se possuem símbolos inválidos ou não.

No arquivo `tests/lexical/invalid_symbols_1.bml` é esperado que o analisador léxico aponte erros na linha 10 - coluna 15 e linha 11 - coluna 16, sobre os caracteres `%` e `^`. Já no arquivo `tests/lexical/invalid_symbols_2.bml` é esperado que os erros apontados sejam sobre os caracteres `@` na linha 2 - coluna 8 e `^` e `#` na linha 4 - coluna 9. É esperado que a mensagem de erro contenha o número da linha e a coluna relativa ao caracter que não está incluído na gramática descrita no Anexo A.

### 4 Análise Sintática

O papel de um analisador sintático, também chamado de *parser*, é obter a cadeia de tokens gerados pelo analisador léxico e verificar se esses nomes das cadeias podem ser gerados pela gramática da linguagem fonte [1]. Também é esperado do *parser* que esse consiga reportar qualquer erro de sintaxe de maneira inteligível.

O Bison [3] é um gerador de *parser* que converte uma gramática livre de contexto dentro de uma notação pré-determinada em um *parser* determinístico LR empregando tabelas de *parser* LALR(1). Como funcionalidade experimental, o Bison pode também gerar tabelas *parser* canônicas LR(1), que foi a opção escolhida para o desenvolvimento do analisador sintático do trabalho da disciplina.

#### 4.1 Implementação do Analisador Sintático

Para a implementação do analisador sintático novas estruturas de dados precisaram ser adicionadas, sendo as mais notáveis a *Abstract Syntax Tree (AST)* e a tabela de símbolos. Uma *AST* é a representação de uma estrutura sintática abstrata do código fonte de uma linguagem, onde cada nó da árvore representa uma construção existente no código fonte de um programa. A tabela de símbolos é uma estrutura de dados que é utilizada pelos compiladores para carregar informações a respeito do programa fonte e é utilizada incrementalmente na fase de síntese para gerar o programa alvo.

As principais funções e a própria representação da *AST* nesse trabalho estão na pasta `src/syntax/ast.c`. A função `create_ast_node` é responsável por criar um nó de uma *AST* que será adicionado a árvore que é preenchida conforme os *tokens* são passados do analisador léxico para o sintático. Já as funções `print_ast` e `free_ast` são responsáveis por mostrar a *AST* formatada no terminal e por liberar a memória alocada durante a utilização do programa.

#### 4.2 Testes de Sintaxe

Arquivos de testes foram disponibilizados junto a este trabalho para validar o funcionamento do analisador sintático gerado. Os arquivos de testes estão na pasta `tests/syntax`.

### 5 Instruções de uso

Dentre os arquivos entregues existe um *Makefile* para facilitar a geração e o uso dos programas relativos ao trabalho da disciplina. Uma vez que o flex e o bison estejam instalado no ambiente, basta executar o comando `make compile` no terminal.

Isso é o suficiente para gerar o arquivo executável necessário para executar o analisador léxico e o analisador sintático. Para fazer o teste em um dos arquivos de testes basta executar o comando `make run` alterando o arquivo de teste para o desejado no *Makefile*.

### 6 Limitações da entrega atual

Devido a problemas pessoais a implementação dessa etapa do projeto não foi finalizada por completa, porém é importante explicitar quais pontos deste trabalho ainda precisam ser finalizados e melhorados. Um ponto importante que ficou incompleto foi a parte referente a mostrar na tela a *AST*. A função `print_ast` está funcional e a lógica para construção dos nós dela também está funcionando corretamente. O que faltou nesse caso foi finalizar as chamadas para construir todos os nós da *AST* de um programa dado.

Outro ponto que não pôde ser concluído foi a tabela de símbolos. Entretanto, assim como na *AST* o código está bem construído para armazenar futuramente

as informações de tipo valores e escopos de variáveis, assim como para as funções de um programa fonte dado. Um último ponto importante citar é que não pude atualizar a gramática descrita no Anexo A. Entretanto a gramática utilizada no Bison é a versão mais atualizada e evoluída em relação a última entrega desse trabalho.

## Referências

1. Alfred V. Aho e Monica S. Lam e Ravi Sethi e Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd edition)*, Addison Wesley Longman Publishing Co., 2006.
2. Domingues, H.H. and Iezzi, G. *ALGEBRA MODERNA: EDIÇÃO REFORMULADA (4ª edição)*, Editora Atual, 2003.
3. Vern Paxson, *Lexical Analysis with flex*. <https://westes.github.io/flex/manual/>, último acesso em 17 de fevereiro de 2021 às 20:53.
4. Troy D. Hanson, *a hash table for c structures* <https://troydhanson.github.io/uthash/>, último acesso em 26 de fevereiro de 2021 às 16:52.
5. Cláudia Nalon, *T01 - Trabalho Prático - Analisador Léxico* <https://aprender3.unb.br/mod/page/view.php?id=294131>, último acesso em 26 de fevereiro de 2021 às 17:23.
6. Robert Cobbert, *Bison 3.7.1*, <https://www.gnu.org/software/bison/manual/>, último acesso em 05 de abril de 2021 às 10:22. Publicado em 08 de Março de 2021.
7. Ajay Gupta, *The syntax of C in Backus-Naur Form*. [https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The syntax of C in Backus-Naur form.htm](https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm), último acesso em 17 de fevereiro de 2021 às 20:59. Publicado em 1999.

## A Gramática da Linguagem Bemolang

Abaixo é apresentada a gramática da linguagem Bemolang na forma *Backus-Naur*. Para definição da Bemolang foi utilizado a gramática da linguagem C adaptada pelo Professor Ajay Gupta [7] da *Purdue University*.

$\langle \text{translation-unit} \rangle$	$::= \langle \text{external-declaration} \rangle^*$
$\langle \text{external-declaration} \rangle$	$::= \langle \text{function-definition} \rangle$   $\langle \text{declaration} \rangle$
$\langle \text{function-definition} \rangle$	$::= \langle \text{type-specifier} \rangle \langle \text{declarator} \rangle \langle \text{compound-statement} \rangle$
$\langle \text{type-specifier} \rangle$	$::= \text{int}$   $\text{float}$   $\text{elem}$   $\text{set}$
$\langle \text{declarator} \rangle$	$::= \langle \text{identifier} \rangle ( \langle \text{parameter-list} \rangle^* )$
$\langle \text{parameter-list} \rangle$	$::= \langle \text{parameter-declaration} \rangle$   $\langle \text{parameter-list} \rangle , \langle \text{parameter-list} \rangle$
$\langle \text{parameter-declaration} \rangle$	$::= \langle \text{type-specifier} \rangle \langle \text{identifier} \rangle$
$\langle \text{conditional-expression} \rangle$	$::= \langle \text{logical-or-expression} \rangle$
$\langle \text{logical-or-expression} \rangle$	$::= \langle \text{logical-and-expression} \rangle$   $\langle \text{logical-or-expression} \rangle \text{    } \langle \text{logical-and-expression} \rangle$
$\langle \text{logical-and-expression} \rangle$	$::= \langle \text{equality-expression} \rangle$   $\langle \text{logical-and-expression} \rangle \text{ \&\& } \langle \text{equality-expression} \rangle$
$\langle \text{equality-expression} \rangle$	$::= \langle \text{relational-expression} \rangle$   $\langle \text{equality-expression} \rangle == \langle \text{relational-expression} \rangle$   $\langle \text{equality-expression} \rangle != \langle \text{relational-expression} \rangle$
$\langle \text{relational-expression} \rangle$	$::= \langle \text{additive-expression} \rangle$   $\langle \text{relational-expression} \rangle \langle < \text{additive-expression} \rangle$   $\langle \text{relational-expression} \rangle \langle > \text{additive-expression} \rangle$

	$\langle \text{relational-expression} \rangle \langle = \rangle \langle \text{additive-expression} \rangle$
	$\langle \text{relational-expression} \rangle \langle \geq \rangle \langle \text{additive-expression} \rangle$
$\langle \text{additive-expression} \rangle$	::= $\langle \text{multiplicative-expression} \rangle$
	$\langle \text{additive-expression} \rangle + \langle \text{multiplicative-expression} \rangle$
	$\langle \text{additive-expression} \rangle - \langle \text{multiplicative-expression} \rangle$
$\langle \text{multiplicative-expression} \rangle$	::= $\langle \text{unary-expression} \rangle$
	$\langle \text{multiplicative-expression} \rangle * \langle \text{unary-expression} \rangle$
	$\langle \text{multiplicative-expression} \rangle / \langle \text{unary-expression} \rangle$
$\langle \text{unary-expression} \rangle$	::= $\langle \text{postfix-expression} \rangle$
	$\langle \text{unary-operator} \rangle \langle \text{unary-expression} \rangle$
$\langle \text{postfix-expression} \rangle$	::= $\langle \text{primary-expression} \rangle$
	$\langle \text{postfix-expression} \rangle ( \langle \text{assignment-expression} \rangle )$
$\langle \text{primary-expression} \rangle$	::= $\langle \text{identifier} \rangle$
	$\langle \text{constant} \rangle$
	$( \langle \text{expression} \rangle )$
$\langle \text{constant} \rangle$	::= $\langle \text{integer-constant} \rangle$
	$\langle \text{character-constant} \rangle$
	$\langle \text{floating-constant} \rangle$
	$\langle \text{empty-constant} \rangle$
	$\langle \text{string} \rangle$
$\langle \text{expression} \rangle$	::= $\langle \text{assignment-expression} \rangle$
	$\langle \text{type-check-expression} \rangle$
	$\langle \text{expression} \rangle , \langle \text{assignment-expression} \rangle$
$\langle \text{assignment-expression} \rangle$	::= $\langle \text{conditional-expression} \rangle$
	$\langle \text{unary-expression} \rangle = \langle \text{assignment-expression} \rangle$
$\langle \text{unary-operator} \rangle$	::= $+$
	$-$
	$!$

$\langle \text{compound-statement} \rangle ::= \langle \text{declaration} \rangle^* \langle \text{statement} \rangle^*$

$\langle \text{declaration} \rangle ::= \langle \text{type-qualifier} \rangle \langle \text{identifier} \rangle ;$

$\langle \text{statement} \rangle ::= \langle \text{expression-statement} \rangle$   
 $\quad | \langle \text{compound-statement} \rangle$   
 $\quad | \langle \text{selection-statement} \rangle$   
 $\quad | \langle \text{iteration-statement} \rangle$   
 $\quad | \langle \text{inclusion-statement} \rangle ;$   
 $\quad | \langle \text{removal-statement} \rangle$   
 $\quad | \langle \text{io-statement} \rangle$   
 $\quad | \langle \text{jump-statement} \rangle$

$\langle \text{expression-statement} \rangle ::= \langle \text{expression} \rangle ? ;$

$\langle \text{membership-expression} \rangle ::= \langle \text{expression} \rangle \text{ in } \langle \text{expression} \rangle$

$\langle \text{type-check-expression} \rangle ::= \text{is\_set} ( \langle \text{identifier} \rangle )$

$\langle \text{selection-statement} \rangle ::= \text{if} ( \langle \text{expression} \rangle ) \langle \text{statement} \rangle$   
 $\quad | \text{if} ( \langle \text{expression} \rangle ) \langle \text{statement} \rangle \text{ else } \langle \text{statement} \rangle$   
 $\quad | \langle \text{existence-statement} \rangle$

$\langle \text{existence-statement} \rangle ::= \text{exists} ( \langle \text{membership-expression} \rangle )$   
 $\quad | \text{exists} ( \langle \text{identifier} \rangle \text{ in } \langle \text{identifier} \rangle )$

$\langle \text{iteration-statement} \rangle ::= \text{for} ( \langle \text{expression} \rangle ? ; \langle \text{expression} \rangle ? ; \langle \text{expression} \rangle ?$   
 $\quad ) \langle \text{statement} \rangle$   
 $\quad | \text{forall} ( \langle \text{membership-expression} \rangle ) \langle \text{statement} \rangle$

$\langle \text{inclusion-statement} \rangle ::= \langle \text{inclusion-expression} \rangle ;$

$\langle \text{inclusion-expression} \rangle ::= \text{add} ( \langle \text{membership-expression} \rangle )$   
 $\quad | \text{add} ( \langle \text{expression} \rangle \text{ in } \langle \text{inclusion-statement} \rangle )$

$\langle \text{removal-statement} \rangle$	$::= \text{remove} ( \langle \text{membership-expression} \rangle ) ;$
$\langle \text{io-statement} \rangle$	$::= \text{write} ( \langle \text{expression} \rangle ) ;$ $\quad   \quad \text{writeln} ( \langle \text{expression} \rangle ) ;$ $\quad   \quad \text{read} ( \langle \text{identifier} \rangle ) ;$
$\langle \text{jump-statement} \rangle$	$::= \text{return} \langle \text{expression} \rangle ? ;$
$\langle \text{identifier} \rangle$	$::= \langle \text{letter} \rangle \langle \text{letter} \rangle$ $\quad   \quad \langle \text{digit} \rangle ^*$
$\langle \text{integer-constant} \rangle$	$::= \langle \text{digit} \rangle ^+$
$\langle \text{character-constant} \rangle$	$::= ' \langle \text{letter} \rangle '$
$\langle \text{floating-constant} \rangle$	$::= \langle \text{digit} \rangle ^+ . \langle \text{digit} \rangle ^+$
$\langle \text{empty-constant} \rangle$	$::= \text{EMPTY}$
$\langle \text{string} \rangle$	$::= \text{“} \langle \text{character} \rangle ^* \text{”}$
$\langle \text{character} \rangle$	$::= \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \mid \langle \text{symbol} \rangle$
$\langle \text{letter} \rangle$	$::= \_ \mid \text{A} \mid \text{B} \mid \text{C} \mid \text{D} \mid \text{E} \mid \text{F} \mid \text{G} \mid \text{H} \mid \text{I} \mid \text{J} \mid \text{K} \mid \text{L} \mid \text{M} \mid \text{N} \mid \text{O} \mid$ $\quad \text{P} \mid \text{Q} \mid \text{R} \mid \text{S} \mid \text{T} \mid \text{U} \mid \text{V} \mid \text{W} \mid \text{X} \mid \text{Y} \mid \text{Z} \mid \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \mid \text{e} \mid$ $\quad \text{f} \mid \text{g} \mid \text{h} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{l} \mid \text{m} \mid \text{n} \mid \text{o} \mid \text{p} \mid \text{q} \mid \text{r} \mid \text{s} \mid \text{t} \mid \text{u} \mid$ $\quad \text{v} \mid \text{w} \mid \text{x} \mid \text{y} \mid \text{z}$
$\langle \text{digit} \rangle$	$::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$\langle \text{symbol} \rangle$	$::= \mid \mid \mid ! \mid \# \mid \$ \mid \% \mid \& \mid ( \mid ) \mid * \mid + \mid , \mid - \mid . \mid / \mid : \mid$ $\quad ; \mid > \mid = \mid < \mid ? \mid @ \mid [ \mid \backslash \mid ] \mid ^ \mid _ \mid ' \mid \{ \mid \} \mid \sim$



## B Exemplos de construção da linguagem Bemolang

Nesta seção são mostrados alguns exemplos de construções possíveis utilizando a linguagem Bemolang. Os exemplos foram criados a partir da linguagem descrita no Anexo A. Nas seções B.1 e B.2 são mostrados exemplos de iteração em conjuntos, adição e remoção de elementos em conjuntos, pertinência de elementos em conjunto, dentre o uso de outras primitivas mostradas na seção 2. Os exemplos foram disponibilizados pela professora Cláudia Nalon [5] da Universidade de Brasília no ambiente de ensino da disciplina de Tradutores.

### B.1 Exemplo de iteração em conjuntos

```
int main() {
    set s;
    s = EMPTY;

    add(1 in s);
    add(2 in s);
    add(5 in s);
    add(8 in s);

    set possibleSums;
    possibleSums = EMPTY;
    int x;

    forall (x in s) {
        set sumsWithX;
        sumsWithX = EMPTY;
        int val;
        forall (val in possibleSums) add((x + val) in sumsWithX);
        forall (val in sumsWithX) add (val in possibleSums);
        if (13 in possibleSums) writeln('y'); else writeln('n');
    }
}
```

### B.2 Exemplo de função que encontra um subconjunto cuja soma dos elementos seja igual a um valor dado

```
set subsum(set s, int target, int cur_sum, set ans) {
    if(target == cur_sum) return ans;
    else if (s == EMPTY) return EMPTY;
    else {
        int el;
        remove((exists (el in s)) in s);
        if(subsum(s, target, cur_sum, ans)) return ans;
        cur_sum = cur_sum + el;
    }
}
```

```
        add(el in ans);
        if(subsum(s, target, cur_sum, ans)) return ans;
        add(el in s);
        remove (el in s);
        return EMPTY;
    }
}
```