

Analizador Léxico - Tradutores

Marcus Vinícius da Silva Borges^[130032891]

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 primeira etapa do trabalho o foco é dar uma breve descrição sobre a linguagem e da implementação de seu analisador léxico.

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 **writeIn**.

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 de gramática 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.

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.

Embora esteja fora do escopo da implementação dessa etapa do projeto, o Anexo C promove uma discussão inicial sobre a implementação da tabela de símbolos a ser implementada na próxima etapa.

4 Testes

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/invalid_symbols_1.bml` é esperado que o analisador léxico aponte erros nas linhas 10 e 11 sobre os caracteres `%` e `^`, respectivamente. Já no arquivo `tests/invalid_symbols_2.bml` é esperado que os erros apontados sejam sobre os caracteres `@` na linha 2 e `^` e `#` na linha 4. 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.

5 Instruções de uso

Dentre os arquivos entregues existe um *Makefile* para facilitar a geração e o uso do analisador léxico. Uma vez que o flex esteja instalado no ambiente, basta rodar o comando `make flex` no terminal.

A saída esperada são dois arquivos: `lex.yy.c` e `bemolang.out`. Para fazer a análise léxica de um arquivo de teste basta rodar o comando `make run` no terminal passando o argumento `P` com o caminho de um dos arquivos de teste, como mostrado abaixo:

```
make run P="./tests/invalid_symbols_2.bml"
```

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 [6] da *Purdue University*.

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

$$\begin{array}{l}
| \langle \textit{relational-expression} \rangle \langle = \rangle \langle \textit{additive-expression} \rangle \\
| \langle \textit{relational-expression} \rangle \langle \geq \rangle \langle \textit{additive-expression} \rangle
\end{array}$$

$$\begin{array}{l}
\langle \textit{additive-expression} \rangle ::= \langle \textit{multiplicative-expression} \rangle \\
| \langle \textit{additive-expression} \rangle + \langle \textit{multiplicative-expression} \rangle \\
| \langle \textit{additive-expression} \rangle - \langle \textit{multiplicative-expression} \rangle
\end{array}$$

$$\begin{array}{l}
\langle \textit{multiplicative-expression} \rangle ::= \langle \textit{unary-expression} \rangle \\
| \langle \textit{multiplicative-expression} \rangle * \langle \textit{unary-expression} \rangle \\
| \langle \textit{multiplicative-expression} \rangle / \langle \textit{unary-expression} \rangle
\end{array}$$

$$\begin{array}{l}
\langle \textit{unary-expression} \rangle ::= \langle \textit{postfix-expression} \rangle \\
| \langle \textit{unary-operator} \rangle \langle \textit{unary-expression} \rangle
\end{array}$$

$$\begin{array}{l}
\langle \textit{postfix-expression} \rangle ::= \langle \textit{primary-expression} \rangle \\
| \langle \textit{postfix-expression} \rangle (\langle \textit{assignment-expression} \rangle)
\end{array}$$

$$\begin{array}{l}
\langle \textit{primary-expression} \rangle ::= \langle \textit{identifier} \rangle \\
| \langle \textit{constant} \rangle \\
| (\langle \textit{expression} \rangle)
\end{array}$$

$$\begin{array}{l}
\langle \textit{constant} \rangle ::= \langle \textit{integer-constant} \rangle \\
| \langle \textit{character-constant} \rangle \\
| \langle \textit{floating-constant} \rangle \\
| \langle \textit{empty-constant} \rangle \\
| \langle \textit{string} \rangle
\end{array}$$

$$\begin{array}{l}
\langle \textit{expression} \rangle ::= \langle \textit{assignment-expression} \rangle \\
| \langle \textit{type-check-expression} \rangle \\
| \langle \textit{expression} \rangle , \langle \textit{assignment-expression} \rangle
\end{array}$$

$$\begin{array}{l}
\langle \textit{assignment-expression} \rangle ::= \langle \textit{conditional-expression} \rangle \\
| \langle \textit{unary-expression} \rangle = \langle \textit{assignment-expression} \rangle
\end{array}$$

$$\begin{array}{l}
\langle \textit{unary-operator} \rangle ::= + \\
| - \\
| !
\end{array}$$

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

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

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

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

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

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

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

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

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

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

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

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

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 calcula a soma de um subconjunto

```
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;
    }
}

```

C Projeto de tabela de símbolos

Essa seção tem como intenção a expor uma ideia inicial de implementação da tabela de símbolos que será utilizada na próxima etapa do projeto junto ao analisador sintático. A tabela de símbolos é uma estrutura de dados que é usado pelos compiladores para carregar informações a respeito do programa fonte e é utilizada incrementalmente na fase de síntese para gerar o programa alvo [1].

As entradas da tabela de símbolos são criadas durante a fase de análise e usadas pelo analisador léxico, *parser* e analisador semântico. Normalmente, o *parser* está em melhor posição do que o analisador léxico para distinguir entre diferentes declarações de um identificador [1].

Para o desenvolvimento da tabela de símbolos deste projeto será utilizado a biblioteca *uthash* [4] criada por Troy D. Hanson. Essa biblioteca facilita a criação e manuseio de uma hash table na linguagem C, estrutura de dados na qual a tabela de símbolos será implementada.

O analisador léxico será adaptado para retornar, para cada lexema encontrado, um *token* que será representado por uma estrutura de dados através do tipo *struct* em C. Esse token terá consigo campo *type* que identifica o tipo de informação que aquele *token* carrega, se é um variável ou uma constante, por exemplo, e o campo *value* que representa o valor daquele *token* no código fonte. A hash table será composta por todos os *tokens* retornados pelo analisador léxico.

Certamente esse planejamento inicial não vai endereçar toda a informação necessária pelo analisador sintático. Entretanto, a intenção nesse primeiro momento é ter uma estrutura básica que será melhorada através do aprofundamento do conhecimento do analisador sintático.

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 - Analizador Léxico* <https://aprender3.unb.br/mod/page/view.php?id=294131>, último acesso em 26 de fevereiro de 2021 às 17:23.
6. 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.