Analisador 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 uma 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.l` 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 [**?**] 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 seu possuem símbolos inválidos ou não.

No arquivo `tests/invalid_symbols_1.bml` é esperado que o análisador 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 a 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 [5] da *Purdue University*.

⟨*translation-unit*⟩ ::= ⟨*external-declaration*⟩*

⟨*external-declaration*⟩ ::= ⟨*function-definition*⟩
| ⟨*declaration*⟩

⟨*function-definition*⟩ ::= ⟨*type-specifier*⟩ ⟨*declarator*⟩ ⟨*compound-statement*⟩

⟨*type-specifier*⟩ ::= `int`
| `float`
| `elem`
| `set`

⟨*declarator*⟩ ::= ⟨*identifier*⟩ `(` ⟨*parameter-list*⟩* `)`

⟨*parameter-list*⟩ ::= ⟨*parameter-declaration*⟩
| ⟨*parameter-list*⟩ `,` ⟨*parameter-list*⟩

⟨*parameter-declaration*⟩ ::= ⟨*type-specifier*⟩ ⟨*identifier*⟩

⟨*conditional-expression*⟩ ::= ⟨*logical-or-expression*⟩

⟨*logical-or-expression*⟩ ::= ⟨*logical-and-expression*⟩
| ⟨*logical-or-expression*⟩ `||` ⟨*logical-and-expression*⟩

⟨*logical-and-expression*⟩ ::= ⟨*equality-expression*⟩
| ⟨*logical-and-expression*⟩ `&&` ⟨*equality-expression*⟩

⟨*equality-expression*⟩ ::= ⟨*relational-expression*⟩
| ⟨*equality-expression*⟩ `==` ⟨*relational-expression*⟩
| ⟨*equality-expression*⟩ `!=` ⟨*relational-expression*⟩

⟨*relational-expression*⟩ ::= ⟨*additive-expression*⟩
| ⟨*relational-expression*⟩ ⟨ <⟨*additive-expression*⟩
| ⟨*relational-expression*⟩ > ⟨*additive-expression*⟩

| ⟨*relational-expression*⟩ ⟨= <*additive-expression*⟩
| ⟨*relational-expression*⟩ >= ⟨*additive-expression*⟩

⟨*additive-expression*⟩ ::= ⟨*multiplicative-expression*⟩
| ⟨*additive-expression*⟩ `+` ⟨*multiplicative-expression*⟩
| ⟨*additive-expression*⟩ `-` ⟨*multiplicative-expression*⟩

⟨*multiplicative-expression*⟩ ::= ⟨*unary-expression*⟩
| ⟨*multiplicative-expression*⟩ `*` ⟨*unary-expression*⟩
| ⟨*multiplicative-expression*⟩ `/` ⟨*unary-expression*⟩

⟨*unary-expression*⟩ ::= ⟨*postfix-expression*⟩
| ⟨*unary-operator*⟩ ⟨*unary-expression*⟩

⟨*postfix-expression*⟩ ::= ⟨*primary-expression*⟩
| ⟨*postfix-expression*⟩ `(` ⟨*assignment-expression*⟩ `)`

⟨*primary-expression*⟩ ::= ⟨*identifier*⟩
| ⟨*constant*⟩
| `(` ⟨*expression*⟩ `)`

⟨*constant*⟩ ::= ⟨*integer-constant*⟩
| ⟨*character-constant*⟩
| ⟨*floating-constant*⟩
| ⟨*empty-constant*⟩
| ⟨*string*⟩

⟨*expression*⟩ ::= ⟨*assignment-expression*⟩
| ⟨*type-check-expression*⟩
| ⟨*expression*⟩ `,` ⟨*assignment-expression*⟩

⟨*assignment-expression*⟩ ::= ⟨*conditional-expression*⟩
| ⟨*unary-expression*⟩ `=` ⟨*assignment-expression*⟩

⟨*unary-operator*⟩ ::= `+`
| `-`
| `!`

⟨*compound-statement*⟩ ::= ⟨*declaration*⟩* ⟨*statement*⟩*

⟨*declaration*⟩ ::= ⟨*type-qualifier*⟩ ⟨*identifier*⟩ `;`

⟨*statement*⟩ ::= ⟨*expression-statement*⟩
| ⟨*compound-statement*⟩
| ⟨*selection-statement*⟩
| ⟨*iteration-statement*⟩
| ⟨*inclusion-statement*⟩ `;`
| ⟨*removal-statement*⟩
| ⟨*io-statement*⟩
| ⟨*jump-statement*⟩

⟨*expression-statement*⟩ ::= ⟨*expression*⟩? `;`

⟨*membership-expression*⟩ ::= ⟨*expression*⟩ `in` ⟨*expression*⟩

⟨*type-check-expression*⟩ ::= `is_set (` ⟨*identifier*⟩ `)`

⟨*selection-statement*⟩ ::= `if (` ⟨*expression*⟩ `)` ⟨*statement*⟩
| `if (` ⟨*expression*⟩ `)` ⟨*statement*⟩ `else` ⟨*statement*⟩
| ⟨*existence-statement*⟩

⟨*existence-statement*⟩ ::= `exists (` ⟨*membership-expression*⟩ `)`
| `exists (` ⟨*identifier*⟩ `in` ⟨*identifier*⟩ `)`

⟨*iteration-statement*⟩ ::= `for (` ⟨*expression*⟩? `;` ⟨*expression*⟩? `;` ⟨*expression*⟩?
`)` ⟨*statement*⟩
| `forall (` ⟨*membership-expression*⟩ `)` ⟨*statement*⟩

⟨*inclusion-statement*⟩ ::= ⟨*inclusion-expression*⟩ `;`

⟨*inclusion-expression*⟩ ::= `add (` ⟨*membership-expression*⟩ `)`
| `add (` ⟨*expression*⟩ `in` ⟨*inclusion-statement*⟩ `)`

⟨*removal-statement*⟩ ::= `remove (` ⟨*membership-expression*⟩ `) ;`

⟨*io-statement*⟩ ::= `write (` ⟨*expression*⟩ `) ;`
| `writeln (` ⟨*expression*⟩ `) ;`
| `read (` ⟨*identifier*⟩ `) ;`

⟨*jump-statement*⟩ ::= `return` ⟨*expression*⟩? `;`

⟨*identifier*⟩ ::= ⟨*letter*⟩⟨*letter*⟩
| ⟨*digit*⟩*

⟨*integer-constant*⟩ ::= ⟨*digit*⟩+

⟨*character-constant*⟩ ::= `'` ⟨*letter*⟩ `'`

⟨*floating-constant*⟩ ::= ⟨*digit*⟩+`.`⟨*digit*⟩+

⟨*empty-constant*⟩ ::= `EMPTY`

⟨*string*⟩ ::= `“` ⟨*character*⟩* `”`

⟨*character*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩ | ⟨*symbol*⟩

⟨*letter*⟩ ::= `_` | `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` |
`P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z` | `a` | `b` | `c` | `d` | `e` |
`f` | `g` | `h` | `i` | `j` | `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` |
`v` | `w` | `x` | `y` | `z`

⟨*digit*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⟨*symbol*⟩ ::= `|` | ` ` | `!` | `#` | `$` | `%` | `&` | `(` | `)` | `*` | `+` | `,` | `-` | `.` | `/` | `:` |
`;` | `>` | `=` | `<` | `?` | `@` | `[` | `\` | `]` | `^` | `_` | `‘` | `{` | `}` | `~`

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.

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 ideial 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 anlisador 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 planejamente 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: EDIÇAO 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. 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, último acesso em 17 de fevereiro de 2021 às 20:59.