Analisador Léxico

Vitor Fernandes Dullens - 16/0148260

Universidade de Brasília

1 Introdução

Este relatório abordará sobre a primeira etapa do processo de criação de um tradutor, que consiste na implementação do analisador léxico para uma linguagem específica, como apresentado no livro base da disciplina [ALSU07]. Neste documento será descrito com detalhes a motivação por trás desta implementação, assim como a descrição da análise e instruções para a compilação e execução do programa. Também será apresentada uma descrição da linguagem à ser analisada.

2 Motivação

Na linguagem C, muitas vezes sentimos falta de operações que facilitam a utilização de conjuntos, como existe em outras linguagens de mais alto nível. Dito isso, afim de facilitar essas operações, uma implementação de uma nova primitiva de dados para conjuntos foi proposta dentro da linguagem C - `set`, assim como operações para a mesma - `add`, `remove`, entre outras.

Abaixo segue um exemplo de código na nova linguagem proposta.

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

    add(1 in add(2 in add(3 in s)));
    /* s = (1, 2, 3) */

    remove(1 in s);
    /* s = (2, 3) */

}
```

Além de operações com conjuntos, também foi adicionado um tipo polimórfico - `elem` que facilita, também, no uso de conjuntos. Mais detalhes sobre a linguagem podem ser encontrados no apêndice A.

3 Descrição da análise léxica

Para a implementação do análisador foi utilizado o programa *Fast Lexical Analyzer Generator* - FLEX [Wes], que consiste em uma ferramenta geradora de programas que reconhecem padrões léxicos em textos.

No arquivo de nome `lexical.l` é possível visualizar as regras léxicas. Para o tratamento das mesmas são declaradas expressões regulares (regex) que às identificam e após essas declarações existe uma sequência de ações que o analisador executa ao encontrar uma regra. Além das regras e ações, no arquivo `lexical.l` também foram definidas funções para leitura de arquivo, assim como três variáveis `int line`, `int column` e `int errors`, que representam, respectivamente, a linha em qual está acontecendo a ação, a coluna e o número de erros encontrados até o momento.

No momento, as ações de quando são encontradas as regras léxicas são apenas de impressão na tela quando um token é encontrado ou uma mensagem de erro caso ocorra. Os tipos possíveis de erros são: token mal formatado e caractere não indentificado, como é possível ver na Seção 4. Ao final da execução, o programa aprensta uma menssagem de quantos erros foram encontrados ao total.

4 Descrição dos arquivos testes

Os testes se encontram na pasta `tests/`, os arquivos `sucess1.c`, `sucess2.c`, `sucess3.c` e `sucess4.c` são testes que contém código correto, já os arquivos `error1.c` e `error2.c` contém códigos incorretos, sendo os seus erros, respectivamente:

1. `ERROR: Unidentified character: '#' - line: 2 - column: 15`
2. `ERROR: Incorrect token format: '321varivel' - line: 2 - column: 4`

5 Instruções para compilação e execução do programa

O programa foi criado e testado em um sistema operacional baseado no UNIX. É necessário a instalação do FLEX [Wes] para a compilação do programa. Ao executar o programa também deverá ser passado o arquivo que será analisado.

Comandos para compilação e execução:

```
$ flex lexical.l
$ gcc lex.yy.c
$ ./a.out tests/<nome-arquivo>.c
```

Referências

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2007.

[Wes] Westes. Flex: Fast lexical analyzer generator.

A Gramática

⟨*program*⟩ ::= ⟨*declaration_list*⟩

⟨*declaration_list*⟩ ::= ⟨*declaration*⟩ ⟨*declaration_list*⟩ | ⟨*declaration*⟩

⟨*declaration*⟩ ::= ⟨*function_declaration*⟩ | ⟨*var_declaration*⟩

⟨*var_declaration*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ ';'

⟨*function_declaration*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ '(' ⟨*params_list*⟩ ')' ⟨*brackets_stmt*⟩

⟨*params_list*⟩ ::= (⟨*type*⟩ ⟨*id*⟩ ',')*

⟨*stmt*⟩ ::= ⟨*for_stmt*⟩ | ⟨*if_else_stmt*⟩ | ⟨*return_stmt*⟩ | ⟨*io_stmt*⟩
| ⟨*brackets_stmt*⟩
| ⟨*exp_stmt*⟩
| ⟨*set_stmt*⟩

⟨*brackets_stmt*⟩ ::= '{' ⟨*declaration*⟩ | ⟨*stmt*⟩* '}'

⟨*io_stmt*⟩ ::= read '(' ⟨*id*⟩ ')' ';'
| write'(' ⟨*string*⟩ | ⟨*exp*⟩ ')' ';'
| writeln'(' ⟨*string*⟩ | ⟨*exp*⟩')' ';'

⟨*for_stmt*⟩ ::= for '(' ⟨*exp*⟩? ';' ⟨*exp*⟩? ';' ⟨*exp*⟩? ')' ⟨*stmt*⟩

⟨*if_else_stmt*⟩ ::= if '(' ⟨*expression*⟩ ')' ⟨*stmt*⟩
| if '(' ⟨*expression*⟩ ')' ⟨*stmt*⟩ else ⟨*stmt*⟩

⟨*return_stmt*⟩ ::= return ';' | return ⟨*exp*⟩ ';'

⟨*set_stmt*⟩ ::= forall'(' ⟨*id*⟩ in ⟨*set_exp*⟩ ')' ⟨*stmt*⟩
| is_set '(' ⟨*id*⟩ | ⟨*set_exp*⟩ ')'

⟨*exp_stmt*⟩ ::= ⟨*exp*⟩ ';' | ';'

⟨*exp*⟩ ::= ⟨*id*⟩ = ⟨*exp*⟩ | ⟨*or_exp*⟩ | ⟨*set_exp*⟩

⟨*set_exp*⟩ ::= add'(' ⟨*set_in_exp*⟩ ')'
| remove'(' ⟨*set_in_exp*⟩ ')'
| exists'(' ⟨*set_in_exp*⟩ ')'

⟨*set_in_exp*⟩ ::= ⟨*or_exp*⟩ in ⟨*set_stmt*⟩

⟨*or_exp*⟩ ::= ⟨*or_exp*⟩ '||' ⟨*and_exp*⟩ | ⟨*and_exp*⟩

⟨*and_exp*⟩ ::= ⟨*and_exp*⟩ '&&' ⟨*relational_exp*⟩ | ⟨*relational_exp*⟩

⟨*relational_exp*⟩ ::= ⟨*relational_exp*⟩ ⟨*relational_op*⟩ ⟨*sum_exp*⟩ | ⟨*sum_exp*⟩

⟨*relational_op*⟩ ::= '<' | '>' | '>=' | '<=' | '==' | '!='

⟨*sum_exp*⟩ ::= ⟨*sum_exp*⟩ '+' ⟨*mul_exp*⟩
| ⟨*sum_exp*⟩ '-' ⟨*mul_exp*⟩
| ⟨*mul_exp*⟩

⟨*mul_exp*⟩ ::= ⟨*mul_exp*⟩ '*' ⟨*primal_exp*⟩
| ⟨*mul_exp*⟩ '/' ⟨*primal_exp*⟩
| ⟨*primal_exp*⟩

⟨*primal_exp*⟩ ::= ⟨*id*⟩ | ⟨*const*⟩ | '(' ⟨*exp*⟩ ')'

⟨*type*⟩ ::= ⟨*basic_type*⟩ | ⟨*elem_type*⟩ | ⟨*set_type*⟩

⟨*const*⟩ ::= ⟨*int_const*⟩ | ⟨*float_const*⟩ | ⟨*empty_const*⟩

⟨*int_const*⟩ ::= ⟨*digit*⟩+

⟨*float_const*⟩ ::= ⟨*digit*⟩+ '.' ⟨*digit*⟩+

⟨*empty_const*⟩ ::= 'EMPTY'

⟨*elem_type*⟩ ::= elem

⟨*set_type*⟩ ::= set

⟨*basic_type*⟩ ::= int | float

⟨*string*⟩ ::= .* | '.*'

⟨*id*⟩ ::= [a-zA-Z_][_a-z0-9A-Z]*

⟨*digit*⟩ ::= [0-9]