Analisador Léxico

Rômulo de Vasconcelos Feijão Filho - 140031260@aluno.unb.br
Profª Cláudia Nalon - nalon@unb.br

Universidade de Brasília, Brasília DF 70910-900, Brasil
{pcr,cic}@unb.br

Abstract. This is the first of four parts of a compiler's implementation, made for the Compilers course administered by professor Cláudia Nalon at University of Brasilia. In this article we will present the motivation, description of the lexical analysis, description of test files and instructions for building and executing the project.

Keywords: Compilers · Lexical Analysis · FLEX.

1 Motivação

Como a primeira fase de um compilador, a tarefa principal do analisador léxico é ler os caracteres da entrada do programa fonte, agrupá-los em lexemas e produzir como saída uma sequência de tokens para cada lexema no programa fonte [1]. Para esse curso, é essencial essa implementação, tanto para alcançar o resultado final desejado do projeto da disciplina, quanto para o aprendizado e fixação dos conceitos ministrados, estes que também serão testados nas provas. A linguagem a ser desenvolvida foi projetada com o intuito de facilitar o uso de conjuntos (*set*) em programas escritos em C. Além do tipo *set*, também será implementado um tipo polimórfico, denominado *elem*.

2 Descrição da Análise Léxica

A Análise Léxica foi feita com o auxílio da ferramenta open source **FLEX** (*Fast Lexical Analyzer Generator*). Foi criado um arquivo **newc.l**, contendo as definições e regras do analisador léxico da linguagem previamente definida e apresentada pela professora. Ao executar o comando **flex newc.l**, é gerado pelo FLEX um arquivo lex.yy.c, contendo a lógica do analisador. Foi criado, com o auxílio de variáveis locais e do FLEX, um mecanismo de detecção de caracteres não permitidos, indicando a linha e a coluna em que este se encontra. Existe também uma função **main** no arquivo **newc.l**, que foi customizada para receber um arquivo e executar o analisador léxico sobre essa entrada. Futuramente, os tokens gerados pelo analisador léxico serão inseridos em uma tabela de símbolos, pois estes serão usados na etapa de análise sintática.

3 Descrição dos Arquivos de Teste

Existem 5 arquivos de teste implementados, todos estes se encontram na pasta **tests**. Os arquivos **test1.txt**, **test2.txt** e **test3.txt** devem passar pelo analisador léxico sem encontrar erros, pois estes contém apenas lexemas identificáveis. Já os arquivos **test4.txt** e **test5.txt** devem apresentar erros, indicando o que causou tal erro e sua localização no arquivo. O arquivo de teste **test4.txt** deve apresentar dois erros após a execução do analisador léxico: "@ at line: 4, column: 7" e "@ at line: 6, column: 9". O arquivo de teste **test5.txt** também deve apresentar dois erros após a execução do analisador léxico: "& at line: 6, column: 8" e "$| atline : 10, column : 8$"

4 Instruções de Compilação e Execução

Para a compilação do programa e execução dos arquivos de teste, foi criado um arquivo **makefile**, a fim de simplificar e acelerar esse processo. Para rodar os arquivos de teste, basta entrar na pasta **src**, utilizando o comando **cd src** e dentro dessa, executar o comando **make runtests**. Na pasta **results** serão criados os arquivos resultantes de cada caso de teste. Caso deseje executar cada comando de compilação separadamente, também dentro da pasta **src** execute, **flex newc.l**, seguido de **gcc lex.yy.c -o newc -ll -g -Wall**, e por fim, **./newc < ../tests/[nome-do-arquivo-entrada] > ../results/[nome-do-arquivo-saida]**.

References

1. Aho, Alfred V., Lam, Monica S., Sethi, Ravi, Ullman, Jeffrey D.: Compiladores: Princípios, Técnicas e Ferramentas. 2nd edn. Pearson, (1999)
2. Paxson, Vern: Lexical Analysis With Flex, for Flex 2.6.2, https://westes.github.io/flex/manual/. Último acesso em: 15 de Fevereiro de 2021

A Grámatica

Uma possível gramática inicial da linguagem a ser implementada foi montada da seguinte maneira:

```
program → declarations_list
declarations_list → declaration declarations_list | declaration
declaration → var_dec | function_declaration
var_dec → type id;
function_declaration → type id ( parameters_list ) { statement_list }
parameters_list → parameter, parameters_list | ε
parameter → type id
```

```
statement_list → statement statement_list | ε
statement → ret_st | var_dec | io_ops | basic_ops | assign_value | expression
assign_value → id = expression | id = set_op
basic_ops → if_op | for_op | forall_op
if_op → IF ( log_op ) { statement_list } |
        IF ( log_op ) { statement_list } ELSE { statement_list }
for_op → FOR ( log_op ) { statement_list }
forall_op → FORALL ( in_set ) { statement_list }
expression → terminal | math_op | log_op | set_op
terminal → id | number | set
set_op → ADD ( in_set ) | REMOVE ( in_set ) | EXISTS ( in_set )
log_op → in_set | is_type | expression log_operand expression
in_set → expression IN expression
diff_is_type → IS_INT | IS_FLOAT | IS_SET
is_type → diff_is_type ( terminal )
math_op → expression operands expression
io_ops → read | write | writeln
read → id = READ()
write → WRITE ( expression )
writeln → WRITELN ( expression )
ret_st → RETURN expression;


id → letter(letter|digit|_|−)*
integer → {digit}+
decimal → {digit}*.{digit}+
number → ({integer}|{decimal})
type → int | float | set | elem
log_operand → < | > | <= | >= | == | != | || | && | !
operands → + | − | * | /
letter → a | ... | z | A | ... | Z
digit → 0 | ... | 9
newline → \n
empty → "EMPTY"
```

Lista de Palavras Reservadas:

- int
- float
- set
- elem
- empty
- if
- else
- for
- return
- read

- write
- writeln
- in
- is_int
- is_float
- is_set
- add
- remove
- exists
- forall