Теория Формальных Языков

Лабораторная работа №5

Язык JavaScript

Мамаев А. А.
ИУ9-52Б

Цель лабораторной работы.

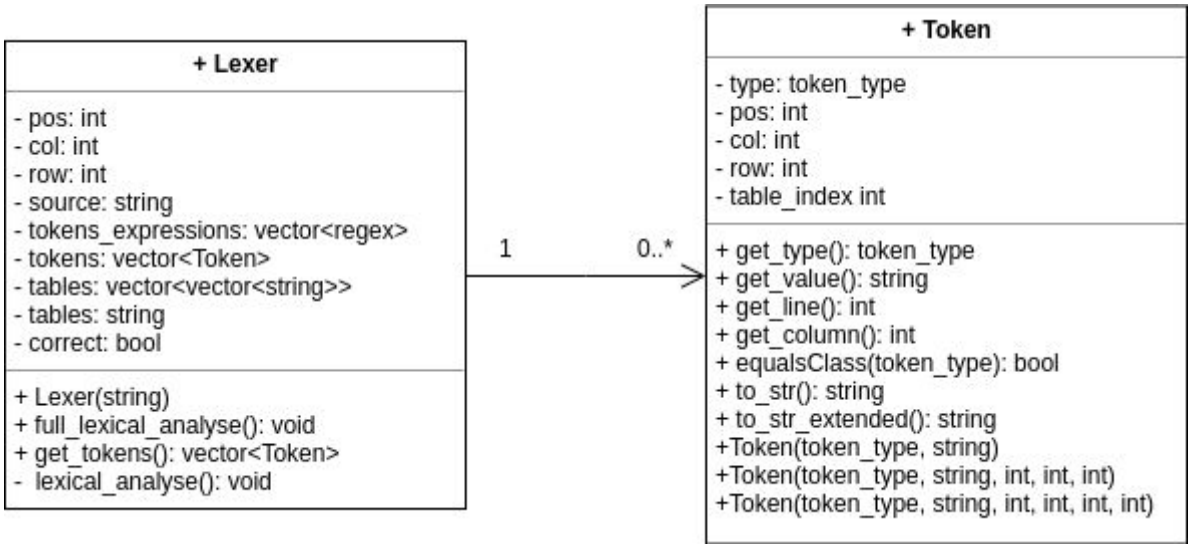
Научиться разрабатывать программные реализации лексического анализатора.

Порядок выполнения лабораторной работы.

1. Выберите язык программирования (см. приложение 4) и в нем определите фрагмент L, для которого будет реализовываться лексический анализатор.
2. Постройте для L грамматику G.
3. Для L определите множество лексем и токенов .
4. Перепишите G в грамматику токенов G′.
5. Постройте лексический анализатор, который должен выполнять следующие функции: – нормализацию исходной программы; – трансляцию программы в промежуточный код; – заполнение таблиц лексем (можно предложить альтернативный способ хранения информации, отсутствующей в промежуточном коде); – поиск ошибок на уровне лексера с указанием их места в исходной программе.

Результат выполнения лабораторной работы.

UML-диаграмма классов лексического анализатора:



Грамматика G:

```
<statement-list>            := <statement> <statement-list-tail> | e
<statement>                 := <expression-statement> |
<assignment-statement> | <while-statement> | <if-else-statement> |
<for-statement> | <function-declaration> | { <statement-list> } | ;
<expression-statement>      := <expression> ;
<expression>                := <primary-expr> | <postfix-expr> |
<prefix-expr> | <unary-expr> | <binary-expr> | <assignment-expr> |
```

```
<ternary-expr> | <object-literal> | <array-literal>
<primary-expr>              := <object-accessor> | <literal> | (
<expression> )
<object-accessor>           := IDENTIFIER <object-accessor-tail>
<object-accessor-tail>      := [ <expression> ] <object-accessor-tail> | .
<object-accessor> | ( <method-argument-list> <object-accessor-tail> | e
<method-argument-list>      := <argument> <argument-list-tail> | )
<method-argument-list-tail> := , <argument> <argument-list> | )
<method-argument>           := IDENTIFIER <argument-tail>
<literal>                   := STRING | NULL_LITERAL | BOOLEAN | NUMERIC |
REGEXP
<postfix-expr>              := <object-accessor> | <object-accessor>
<doubled-op>
<prefix-expr>               := <doubled-op> <object-accessor> |
<object-accessor>
<doubled-op>                := ++ | --
<unary-expr>                := <unary-op> <primary-expr>
<unary-op>                  := - | + | !
<binary-expr>               := <primary-expr> <binary-op> <expression>
<binary-op>                 := === | !== | <= | >= | == | != | << | >> | &&
| || | < | > | + | - | * | / | %
<assignment-expr>           := <assignment> | <assignment> ,
<assignment-expr>
<assignment>                := <object-accessor> <assignment-op>
<expression> | <object-accessor> = <function-declaration>
<assignment-op>             := += | -= | *= | %= | &= | |= | ^= | =
<ternary-expr>              := <primary-expr> ? <primary-expr> :
<primary-expr>
<object-literal>            := { <object-key-value-list>
<object-key-value-list>     := <object-key-value>
<object-key-value-list-tail> | }
<object-key-value-list-tail> := , <object-key-value>
<object-key-value-list-tail> | }
<object-key-value>          := STRING : <expression> | IDENTIFIER :
<expression>
<array-literal>             := [ <array-element-list>
<array-element-list>        := <expression> <array-element-list-tail> | ]
<array-element-list-tail>   := , <expression> <array-element-list-tail> | ]
<assignment-statement>      := var <assignment-expr> ; | let
<assignment-expr> ; | const <assignment-expr> ;
<function-declaration>      := function ( <argument-list> <function-body>
<argument-list>             := <argument> <argument-list-tail> | )
<argument-list-tail>        := , <argument> <argument-list> | )
<argument>                  := IDENTIFIER <argument-tail>
<argument-tail>             := = <expression> | e
<function-body>             := { <func-body-statement-list> |
<func-body-statement>
<func-body-statement-list>  := <func-body-statement>
```

```
<func-body-statement-list-tail> | }
<func-body-statement>       := <statement> | return <expression> ;
<while-statement>           := while ( <expression> ) <loop-body>
<for-statement>             := for ( <expression> ; <expression> ;
<expression> ) <loop-body>
<loop-body>                 := { <loop-body-statement-list> |
<loop-body-statement>
<loop-body-statement-list>  := <loop-body-statement>
<loop-body-statement-list-tail> | }
<loop-body-statement>       := break ; | continue ; | <statement>
<if-else-statement>         := <if-statement> <else-statement-list>
<if-statement>              := if ( <expression> ) <if-body>
<else-statement-list>       := else <if-statement> <else-statement-list> |
else <if-body> | e
<if-body>                   := { <if-body-statement-list> |
<if-body-statement>
<if-body-statement-list>    := <if-body-statement> <if-body-statement-list>
| }
<if-body-statement>         := <statement> | break ;
```

Грамматика токенов G′:

```
<statement-list>            := <statement> <statement-list-tail> | e
<statement-list-tail>       := <statement> <statement-list-tail> | e
<statement>                 := <expression-statement> |
<assignment-statement> | <while-statement> | <if-else-statement> |
<for-statement> | <function-declaration> | LEFT_CURLY <statement-list>
RIGHT_CURLY | SEMICOLON
<expression-statement>      := <expression> SEMICOLON
<expression>                := <primary-expr> | <postfix-expr> |
<prefix-expr> | <unary-expr> | <binary-expr> | <assignment-expr> |
<ternary-expr> | <object-literal> | <array-literal>
<primary-expr>              := <object-accessor> | <literal> | LEFT_ROUND
<expression> RIGHT_ROUND
<object-accessor>           := IDENTIFIER <object-accessor-tail>
<object-accessor-tail>      := LEFT_SQUARE <expression> RIGHT_SQUARE
<object-accessor-tail> | POINT <object-accessor> | LEFT_ROUND
<method-argument-list> <object-accessor-tail> | e
<method-argument-list>      := <argument> <argument-list-tail> |
RIGHT_ROUND
<method-argument-list-tail> := COMMA <argument> <argument-list> |
RIGHT_ROUND
<method-argument>           := IDENTIFIER <argument-tail>
<literal>                   := STRING | NULL_LITERAL | BOOLEAN | NUMERIC |
REGEXP
<postfix-expr>              := <object-accessor> | <object-accessor>
<doubled-op>
```

```
<prefix-expr>               := OP_DOUBLED <object-accessor> |
<object-accessor>
<doubled-op>                := OP_DOUBLED
<unary-expr>                := <unary-op> <primary-expr>
<unary-op>                  := OP_ADDITIVE | OP_EXCLAMATION
<binary-expr>               := <primary-expr> <binary-op> <expression>
<binary-op>                 := OP_ADDITIVE | OP_BINARY
<assignment-expr>           := <assignment> | <assignment> COMMA
<assignment-expr>
<assignment>                := <object-accessor> <assignment-op>
<expression> | <object-accessor> = <function-declaration>
<assignment-op>             := OP_ASSIGN
<ternary-expr>              := <primary-expr> QUESTION <primary-expr> COLON
<primary-expr>
<object-literal>            := LEFT_CURLY <object-key-value-list>
<object-key-value-list>     := <object-key-value>
<object-key-value-list-tail> | RIGHT_CURLY
<object-key-value-list-tail> := COMMA <object-key-value>
<object-key-value-list-tail> | RIGHT_CURLY
<object-key-value>          := STRING : <expression> | IDENTIFIER :
<expression>
<array-literal>             := LEFT_SQUARE <array-element-list>
<array-element-list>        := <expression> <array-element-list-tail> |
RIGHT_SQUARE
<array-element-list-tail>   := COMMA <expression> <array-element-list-tail>
| RIGHT_SQUARE
<assignment-statement>      := VAR <assignment-expr> SEMICOLON | LET
<assignment-expr> SEMICOLON | CONST <assignment-expr> SEMICOLON
<function-declaration>      := function LEFT_ROUND <argument-list>
<function-body>
<argument-list>             := <argument> <argument-list-tail> |
RIGHT_ROUND
<argument-list-tail>        := COMMA <argument> <argument-list> |
RIGHT_ROUND
<argument>                  := IDENTIFIER <argument-tail>
<argument-tail>             := OP_EQUAL <expression> | e
<function-body>             := LEFT_CURLY <func-body-statement-list> |
<func-body-statement>
<func-body-statement-list>  := <func-body-statement>
<func-body-statement-list-tail> | RIGHT_CURLY
<func-body-statement>       := <statement> | return <expression> SEMICOLON
<while-statement>           := WHILE LEFT_ROUND <expression> RIGHT_ROUND
<loop-body>
<for-statement>             := FOR LEFT_ROUND <expression> SEMICOLON
<expression> SEMICOLON <expression> RIGHT_ROUND <loop-body>
<loop-body>                 := LEFT_CURLY <loop-body-statement-list> |
<loop-body-statement>
<loop-body-statement-list>  := <loop-body-statement>
```

```
<loop-body-statement-list-tail> | RIGHT_CURLY
<loop-body-statement>       := BREAK SEMICOLON | CONTINUE SEMICOLON |
<statement>
<if-else-statement>         := <if-statement> <else-statement-list>
<if-statement>              := IF LEFT_ROUND <expression> RIGHT_ROUND
<if-body>
<else-statement-list>       := ELSE <if-statement> <else-statement-list> |
ELSE <if-body> | e
<if-body>                   := LEFT_CURLY <if-body-statement-list> |
<if-body-statement>
<if-body-statement-list>    := <if-body-statement> <if-body-statement-list>
| RIGHT_CURLY
<if-body-statement>         := <statement> | BREAK SEMICOLON
```

Листинг 1. Использование (main.cpp):

```
int main() {

    string input_file = "/home/alexey/TFL/Lab6/test.js";
    string source = read_file(input_file);

    Lexer lex(source);
    lex.full_lexical_analyse();

    return 0;
}
```

Листинг 2. Лексический анализ (Lexer.cpp):

```
void Lexer::lexical_analyse() {
    regex cur_space_regex("^([ \t]+)");
    pos = 0, col = 0, row = 0;

    while (pos < source.size()) {
        string cur_source = source.substr(pos);
        smatch cur_space_match;

        if (regex_search(cur_source, cur_space_match, cur_space_regex)) {
            pos += cur_space_match[0].length();
            continue;
        }

        bool found = false;
        smatch match;

        for (int i = 0; i < tokens_expressions.size(); ++i) {
            if (regex_search(cur_source, match, tokens_expressions[i])) {
                int code = i;
                string matched_substr = match[0];
                if (STRING <= code && code <= OP_ADDITIVE) {
```

```
                    tokens.emplace_back(static_cast<token_type>(code),
matched_substr, pos, col, row,
                                        tables[i - STRING].size());

                    tables[i - STRING].push_back(matched_substr);
                } else if (NEWLINE < code) {
                    tokens.emplace_back(static_cast<token_type>(code),
matched_substr, pos, col, row);
                }
                pos += matched_substr.length();
                if (code == NEWLINE) {
                    col = 0;
                    row++;
                } else if (code == COMMENT && matched_substr[1] == '*') {
                    int count_from_space = 0, height = 0;
                    for (char c : matched_substr) {
                        if (c == '\n') {
                            count_from_space = 0;
                            height++;
                        } else {
                            count_from_space++;
                        }
                    }
                    row += height;
                    col = count_from_space;
                } else {
                    col += matched_substr.length();
                }

                found = true;
                break;
            }
        }
        if (!found) {
            string res = cur_source.substr(0, 1);
            if (correct) {
                correct = false;
                incorrect = Token(UNKNOWN, res, pos, col, row,
tables[UNKNOWN].size());
            }

            tokens.emplace_back(incorrect);
            tables[UNKNOWN].push_back(res);
            pos++;
            col++;
        }
    }
}
```

Типы токенов (реализованы как перечисление enum):

```
enum token_type {
    // DEFINITE TOKENS
    BREAK,
    CONTINUE,
    RETURN,
    FUNCTION,
    FOR,
    WHILE,
```

```
    IF,
    ELSE,
    VAR,
    LET,
    CONST,

    // STORABLE TOKENS
    STRING,
    BOOLEAN,
    NUMERIC,
    REGEXP,
    IDENTIFIER,
    OP_DOUBLED,
    OP_BINARY,
    OP_ASSIGN,
    OP_ADDITIVE,

    // DEFINITE TOKENS
    OP_EQUAL,
    OP_EXCLAMATION,     // восклицательный знак !
    NULL_LITERAL,
    SEMICOLON,          // точка с запятой ;
    POINT,
    COMMA,
    QUESTION,
    COLON,              // двоеточие :
    LEFT_ROUND,
    RIGHT_ROUND,
    LEFT_SQUARE,
    RIGHT_SQUARE,
    LEFT_CURLY,
    RIGHT_CURLY,

    UNKNOWN
};
```

Соответствующие данным типам регулярные выражения:

```
regex(R"(^((/\*(.|\n)*?\*/)|(//[^\n]*)))"),
regex(R"(^\n)"),


regex(R"(^break)"),
regex(R"(^continue)"),
regex(R"(^return)"),
regex(R"(^function)"),
regex(R"(^for)"),
regex(R"(^while)"),
regex(R"(^if)"),
regex(R"(^else)"),
regex(R"(^var)"),
regex(R"(^let)"),
regex(R"(^const)"),


regex(R"(^(("([^"\\]|\\.)*")|('([^'\\]|\\.)*')))"),
regex(R"(^(true|false))"),
regex(R"(^((0[xb][a-fA-F0-9]+)|([0-9]+((\.[0-9]+)([eE][+\\-]?[0-9]+)?)?)))"),
regex(R"(^(/.*/[gimsuy]*))"),
```

```
regex(R"(^([a-zA-Z$][\w]*))"),
regex(R"(^((\+{2})|(--)))"),
regex(R"(^((===)|(!==)|(<=)|(>=)|(==)|(!=)|(<<)|(>>)|(&&)|([|]{2})|[<>*/%]))"),
regex(R"(^((\+=)|(-=)|(\*=)|(%=)|(/=)))"),
regex(R"(^([+\-])),


regex(R"(^([=])),
regex(R"(^([!])),
regex(R"(^null)"),
regex(R"(^([;])),
regex(R"(^([.])),
regex(R"(^([,])),
regex(R"(^([?])),
regex(R"(^([:])),
regex(R"(^([(])),
regex(R"(^([)])),
regex(R"(^([\[])),
regex(R"(^([\]])),
regex(R"(^([{])),
regex(R"(^([}]))")
```