Теория Формальных Языков

Лабораторная работа №6

Вариант №14. Язык JavaScript

Мамаев А. А.
ИУ9-52Б

Цель лабораторной работы.

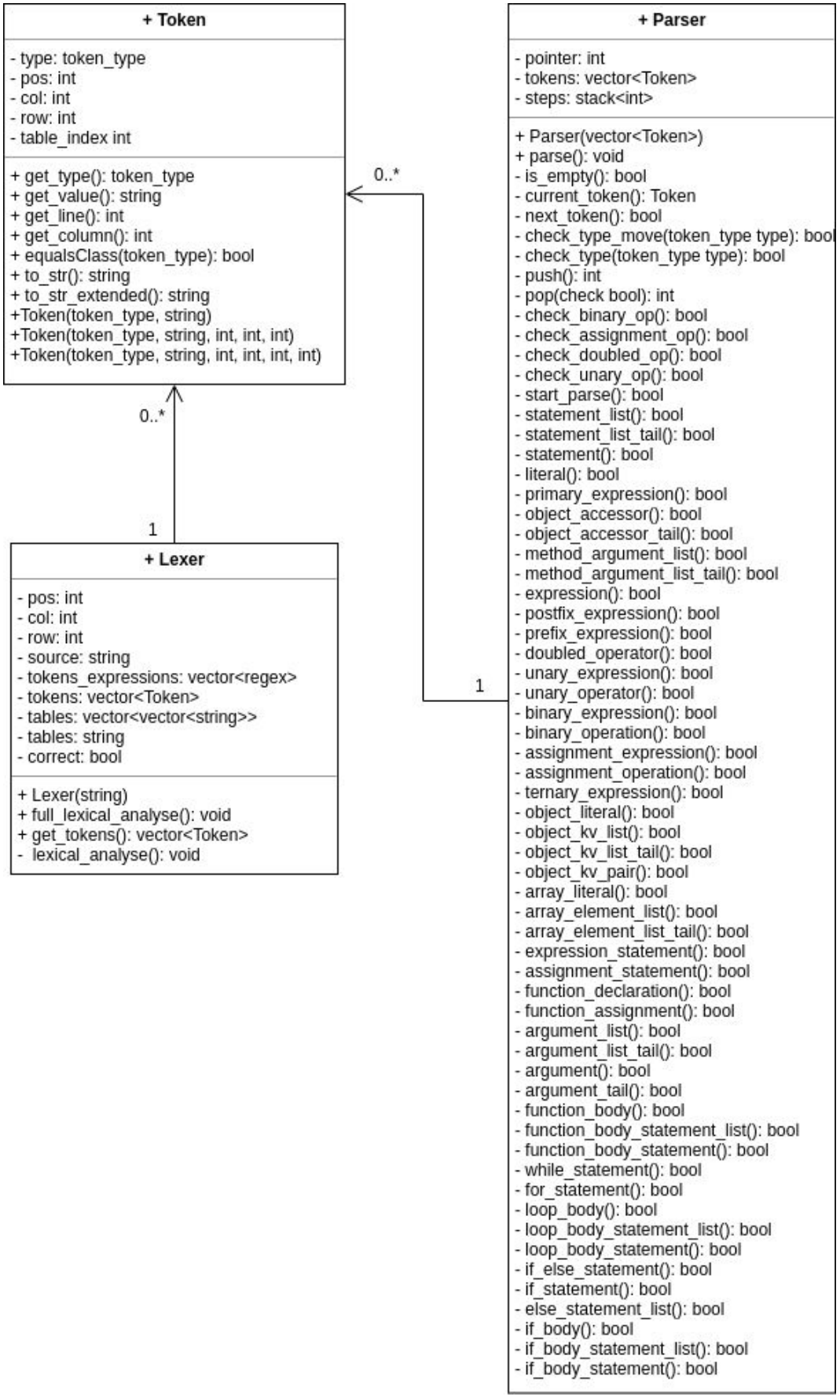
Научиться:
– задавать язык с помощью контекстно-свободной грамматики
– устранять левую рекурсию,
– делать синтаксический разбор сверху вниз
– реализовывать синтаксический анализатор

Порядок выполнения лабораторной работы.

1. Построить синтаксический анализатор на основе алгоритма сверху вниз.

Результат выполнения лабораторной работы.

UML-диаграмма классов синтаксического анализатора:



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();

    Parser parser(lex.getTokens());
    parser.parse();

    return 0;
}
```

Листинг 2. Лексический анализ (Parser.cpp):

```
void Parser::parse() {
    correct = start_parse();

    cout << (correct ? "CORRECT" : "NOT CORRECT") << endl;

    if (!correct && !is_empty())
        cout << "Syntax error on line " << current_token().get_line();
}

bool Parser::start_parse() {
    pointer = 0;
    return statement_list();
}

// PARSER


bool Parser::statement_list() {
    if (is_empty())
        return true;
    return statement() && statement_list_tail();
}

bool Parser::statement_list_tail() {
    if (is_empty())
        return true;
    return statement() && statement_list_tail();
}

bool Parser::statement() {

    bool expressions = expression_statement() ||
                       assignment_statement() ||
                       while_statement() ||
                       if_else_statement() ||
                       for_statement() ||
                       function_declaration();
```

```
    if (expressions)
        return expressions;

    if (check_type(SEMICOLON)) {
        next_token();
        return true;
    }

    push();

    bool check_block = check_type_move(LEFT_CURLY) && statement_list() &&
check_type_move(RIGHT_CURLY);

    pop(check_block);
    return check_block;
}

bool Parser::expression_statement() {
    push();
    bool check_expression = expression() && check_type_move(SEMICOLON);

    pop(check_expression);
    return check_expression;
}

bool Parser::literal() {
    bool check_literal = check_type(STRING) ||
                         check_type(NULL_LITERAL) ||
                         check_type(BOOLEAN) ||
                         check_type(NUMERIC) ||
                         check_type(REGEXP);
    if (check_literal)
        next_token();
    return check_literal;
}

bool Parser::primary_expression() {
    bool expressions = object_accessor() || literal();
    if (expressions)
        return expressions;

    push();
    bool check_expression = check_type_move(LEFT_ROUND) && expression() &&
check_type_move(RIGHT_ROUND);

    pop(check_expression);

    return check_expression;
}

bool Parser::object_accessor() {
    push();
    bool check_ident = check_type_move(IDENTIFIER) && object_accessor_tail();

    pop(check_ident);
    return check_ident;
}

bool Parser::object_accessor_tail() {
    push();
```

```
    if (check_type(LEFT_SQUARE)) {
        next_token();
        bool accessor_expression = expression() && check_type_move(RIGHT_SQUARE)
&& object_accessor_tail();

        pop(accessor_expression);
        return accessor_expression;
    } else if (check_type(POINT)) {
        next_token();
        bool accessor_expression = object_accessor();

        pop(accessor_expression);
        return accessor_expression;
    } else if (check_type(LEFT_ROUND)) {
        next_token();
        bool accessor_expression = method_argument_list() &&
check_type_move(RIGHT_ROUND) && object_accessor_tail();

        pop(accessor_expression);
        return accessor_expression;
    }
    pop(true);
    return true;
}
```

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"(^([}])))")
```

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
```

2. Провести сравнительный анализ синтаксиса конструкций if , While и способа определения переменных в PYTHON от C#

1) Способ определения переменных:

● Python — это объектно-ориентированный язык со строгой динамической неявной типизацией. Явно указывать тип значения переменной нет необходимости, поэтому объявление новой переменной происходит следующим образом:

```
>>> a = 5
>>> a
5
>>> b = "test"
>>> b
"test"
>>> c = float(5)   # без явного приведения типа с будет int
>>> c
5.0
>>> d = list(1, 2, 3, 4, 5.0)
>>> d
[1, 2, 3, 4, 5.0] # первые 4 элемента — int, 5 — float
```

Начиная с ранних версий Python 3 существует возможность указывать типы, а с Python 3.6 и объявляемых переменных, что тем не менее ничего не гарантирует.

```
>>> def sqr(a: float) → float:
>>> ...
>>> e: int = "temp"
>>> e
"temp"
```

● C# - это объектно-ориентированный язык со строгой статической обычно явной (почему обычно — будет показано далее) типизацией.

```
int f = 42;
```

var g = 5.0; (компилятор автоматически выведет тип переменной)

Тем не менее, так как помимо C# платформа .NET поддерживает и множество других языков (например, F#), то в C# все же есть возможность создать переменную с динамической типизацией, что делать крайне не рекомендуется.

```
Dynamic h = 7;
h.hello(); // компилятор не выдаст никакой ошибки, но в рантайме программа упадет
```

2) If:

● Python не использует «;» и «{}» - классические разделительные операторы в C-подобных языках, к коим относится и C#. Выделение блоков кода строится на основе пробелов или табуляций (первое предпочтительнее). Также в условиях не требуются круглые скобки.

Оператор if в Python помимо классических if и else имеет также и ветвь elif (сокращение от else if):

```
if a == 5:
   print("a = 5")
elif a == 6:
   print("a = 6")
else if a == 7:
   print("a = 5")
else:
   print("test")
   print("a = 8")
```

● C# - это классический C-подобный язык и оператор If выглядит в нем соответствующе:

```
if (a == 5)
   console.WriteLine("a = 5")
else if (a == 6)
   console.WriteLine("a = 6")
else
   console.WriteLine("a = 7")
```

Для того, чтобы разместить в одном блоке несколько операций, требуется использовать фигурные скобки.

```
if (a == 5) {
   cconsole.WriteLine("a = 5")
   console.WriteLine("a equals to five")
}
```

3) While:

● Оператор While (также как и For) в языке Python интересен тем, что он имеет else ветвь, которая выполняется, если в цикле не был вызван break. В остальном Pythonовский While вполне обычен. Помимо break есть и continue.

```
>>> i = 0
>>> while i < 5:
>>>    i += 1
>>>    if i == 5:
>>>      break
>>> else:
>>>   print("5 not found")
```

Важно отметить, что цикла Do-While в Python нет и его приходится

«имитировать» посредством while True: с проверкой с breakом в конце.

● Оператор While в языке C# типичен для C-подобных языков и не представляет из себя ничего интересного. Помимо него имеется и такой же классические Do-While.

```
int i = 0;
while (i < 5) {
   Console.WriteLine(i);
   i++;
}
int i = 5;
do {
   Console.WriteLine(i);
   i--;
}
while (i > 0);
```