Национальный исследовательский университет ИТМО
Мегафакультет компьютерных технологий и управления
Факультет программной инженерии и компьютерной техники

Отчет по практическому заданию №1

по курсу «Языки программирования»

Вариант: 2

Выполнил студент группы P4119 _____ Д.Л. Симоновский
(подпись)

Руководитель _____ Ю. Д. Кореньков
(подпись)

22 октября 2025 г.

Санкт-Петербург
2025

Оглавление

1. Цель работы 2

2. План работы 2

3. Ход работы 2

3.1. Разработка синтаксического анализатора 2

3.2. Разработка тестовой программы 9

3.3. Тестирование разработанной грамматики 13

4. Вывод 19

1. Цель работы

Использовать средство синтаксического анализа по выбору, реализовать модуль для разбора текста в соответствии с языком по варианту. Реализовать построение по исходному файлу с текстом синтаксического дерева с узлами, соответствующими элементам синтаксической модели языка. Вывести полученное дерево в файл в формате, поддерживающем просмотр графического представления.

2. План работы

1. Изучить выбранное средство синтаксического анализа.
2. Изучить синтаксис разбираемого по варианту языка и записать спецификацию для средства синтаксического анализа.
3. Реализовать модуль, использующий средство синтаксического анализа для разбора языка по варианту.
4. Реализовать тестовую программу для демонстрации работоспособности созданного модуля.

3. Ход работы

3.1. Разработка синтаксического анализатора

Для разработки синтаксического анализатора была выбрана библиотека tree-sitter, на ней уже есть готовые анализаторы для СИ, C++ и других языков, что поможет при проектировании своего анализатора.

Выполним реализацию общей части синтаксической модели для всех вариантов:

```
identifier: "[a-zA-Z_][a-zA-Z_0-9]*"; // идентификатор
str: "\"[^\"\\]*(?:\\.[^\"\\]*)*\"";  // строка, окруженная двойными кавычками
char: "'[^']'";                       // одиночный символ в одинарных кавычках
hex: "0[xX][0-9A-Fa-f]+";             // шестнадцатеричный литерал
bits: "0[bB][01]+";                   // битовый литерал
dec: "[0-9]+";                        // десятичный литерал
bool: 'true'|'false';                 // булевский литерал
list<item>: (item (',' item)*)?;      // список элементов, разделённых запятыми
```

В tree-sitter эти элементы будут выглядеть следующим образом:

Листинг 3.1. Базовые элементы грамматики

```
identifier: _ => /[a-zA-Z_][a-zA-Z_0-9]*/,
str: _ => /\"[^\"\\]*(?:\\.[^\"\\]*)*\"/,
char: _ => /\'[^\']\'/,
hex: _ => /0[xX][0-9A-Fa-f]+/,
bits: _ => /0[bB][01]+/,
dec: _ => /[0-9]+/,

_true: _ => token('true'),
_false: _ => token('false'),
bool: $ => choice($._true, $._false),
```

Как можно заметить, в tree-sitter грамматика пишется на js, однако после этого компилируется в быстрый СИ-код. В приведенном фрагменте отсутствует описание списка элементов т.к. он будет реализовываться для каждого конкретного случая отдельно.

Далее будет приведена реализация блоков для варианта 2. Разработка выполнялась снизу вверх, сначала expr, затем statement, а потом уже верхнеуровневое описание sourceItem.

Ниже приведен блок синтаксической модели для expr:

```
expr: {                         // присваивание через ':='
   |binary: expr binOp expr;    // где binOp - символ бинарного оператора
   |unary: unOp expr;           // где unOp - символ унарного оператора
   |braces: '(' expr ')';
   |call: expr '(' list<expr> ')';
   |indexer: expr '[' list<expr> ']';
   |place: identifier;
   |literal: bool|str|char|hex|bits|dec;
};
```

Аналогичным образом выглядит этот же блок в tree-sitter:

Листинг 3.2. Блок expr в tree-sitter

```
expression: $ => choice(
  field('unary', $.unary_expression),
  field('binary', $.binary_expression),
  field('braces', $.braces_expression),
  field('call', $.call_expression),
  field('indexer', $.indexer),
  field('place', $.identifier),
  field('literal', $.literal),
),
```

Каждый из полей ссылается на другое правило. Самые простые – последние два. Place напрямую ссылается на идентификатор, объявленный раньше, literal же является простым выбором между объявленными ранее элементами:

Листинг 3.3. Поле literal в tree-sitter

```
literal: $ => choice($.bool, $.str, $.char, $.hex, $.bits, $.dec),
```

Далее идут очень схожие конструкции: call и indexer:

Листинг 3.4. Поля indexer и call

```
indexer: $ => prec(PREC.SUBSCRIPT, seq(
  field('expr', $.expression),
  seq('[', optional(field('listExpr', $.list_expr)), ']')
)),

call_expression: $ => prec(PREC.CALL, seq(
  field('expr', $.expression),
  seq('(', optional(field('listExpr', $.list_expr)), ')')
)),

list_expr: $ => seq(
  field('expr', $.expression),
  repeat(seq(',', field('expr', $.expression)))
),
```

PREC задает приоритет операций вызова, это enum с значениями приоритета, взятый из репозитория для парсинга языка СИ, он выглядит следующим образом:

Листинг 3.5. Enum приоритетов операций

```
const PREC = {
  PAREN_DECLARATOR: -10,
  ASSIGNMENT: -2,
  CONDITIONAL: -1,
  DEFAULT: 0,
  LOGICAL_OR: 1,
  LOGICAL_AND: 2,
  INCLUSIVE_OR: 3,
  EXCLUSIVE_OR: 4,
  BITWISE_AND: 5,
  EQUAL: 6,
  RELATIONAL: 7,
  OFFSETOF: 8,
  SHIFT: 9,
  ADD: 10,
  MULTIPLY: 11,
  CAST: 12,
  SIZEOF: 13,
  UNARY: 14,
  CALL: 15,
  FIELD: 16,
  SUBSCRIPT: 17,
};
```

Как можно заметить, тут также заданы приоритеты и для многих других бинарных операций. Их использование приведено ниже:

Листинг 3.6. Обработка бинарных операций в tree-sitter

```
binary_expression: $ => {
      const table = [
```

```
        [':=', PREC.ASSIGNMENT],
        ['+', PREC.ADD],
        ['-', PREC.ADD],
        ['*', PREC.MULTIPLY],
        ['/', PREC.MULTIPLY],
        ['%', PREC.MULTIPLY],
        ['||', PREC.LOGICAL_OR],
        ['&&', PREC.LOGICAL_AND],
        ['|', PREC.INCLUSIVE_OR],
        ['^', PREC.EXCLUSIVE_OR],
        ['&', PREC.BITWISE_AND],
        ['=', PREC.EQUAL],
        ['!=', PREC.EQUAL],
        ['>', PREC.RELATIONAL],
        ['>=', PREC.RELATIONAL],
        ['<=', PREC.RELATIONAL],
        ['<', PREC.RELATIONAL],
        ['<<', PREC.SHIFT],
        ['>>', PREC.SHIFT],
      ];

      return choice(...table.map(([operator, precedence]) => {
        return prec.right(precedence, seq(
          field('expr', $.expression),
          // @ts-ignore
          field('binOp', alias(operator, $.bin_op)),
          field('expr', $.expression),
        ));
      }));
    },
```

Данный блок кода также был взят из кода для анализатора для языка СИ и адаптирован под существующую грамматику. Данный код создает правило «любой из» с заданными приоритетами.

Оставшиеся конструкции приведены ниже:

Листинг 3.7. Грамматика для unary и braces

```
unary_expression: $ => prec.left(PREC.UNARY, seq(
  field('unOp', alias(choice('!', '~'), $.un_op)),
  field('expr', $.expression),
)),

braces_expression: $ => seq(
  '(',
  field('expr', $.expression),
  ')'
),
```

Таким образом были реализованы все блоки Expressions, далее будет рассмотрен следующий уровень, а именно Statement, основной «строительный блок» функции.

Ниже приведена синтаксическая модель, данная во втором варианте, которую необходимо реализовать:

```
statement: {
   |if: 'if' expr 'then' statement ('else' statement)?;
   |block: 'begin' statement* 'end' ';';
   |while: 'while' expr 'do' statement;
   |do: 'repeat' statement ('while'|'until') expr ';';
   |break: 'break' ';';
   |expression: expr ';';
};
```

В грамматике tree-sitter он будет иметь следующий вид:

Листинг 3.8. Блок statement в грамматике tree-sitter.

```
statement: $ => choice(
  field('if', $.if_statement),
  field('block', $.block_content),
  field('while', $.while_content),
  field('do', $.do_content),
  field('break', $.break_content),
  field('expression', $.expression_content),
),
```

Здесь также удобно пройтись снизу вверх, разобрав каждый блок. Ниже приведены блоки break и expression, как самые простые:

Листинг 3.9. Конструкции break и expression

```
break_content: $ => seq('break', ';'),
expression_content: $ => field('expr', seq($.expression, ';')),
```

Конструкции циклов while и do, а также block не сложные, они приведены ниже:

Листинг 3.10. Конструкции циклов и block

```
do_content: $ => seq(
  'repeat',
  field('statement', $.statement),
  choice('while', 'until'),
  field('expr', $.expression),
  ';'
),

while_content: $ => seq(
  'while',
  field('expr', $.expression),
  'do',
  field('statement', $.statement),
),

block_content: $ => seq(
  'begin',
  repeat(field('statement', $.statement)),
  'end',
  ';'
),
```

Конструкция if не так проста, как кажется из-за блока else, который является необязательным. Здесь необходимо правильно решить проблему «висящего else». В данной работе это было решено по примеру с СИ грамматики:

Листинг 3.11. Конструкция if-else

```
if_statement: $ => prec.right(seq(
  'if',
  field('expr', $.expression),
  'then',
  field('statement', $.statement),
  optional(seq('else', field('statement', $.statement))),
)),
```

Таким образом был полностью обработан блок statement, для завершения грамматики осталось лишь разработать объявление функций:

```
source: sourceItem*;

typeRef: {
   |builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
   |custom: identifier;
   |array: 'array' '[' (',')* ']' 'of' typeRef;
};

funcSignature: identifier '(' list<argDef> ')' (':' typeRef)? {
   argDef: identifier (':' typeRef)?;
};

sourceItem: {
   |funcDef: 'method' funcSignature (body|';') {
      body: ('var' (list<identifier> (':' typeRef)? ';')*)? statement.block;
};
```

Здесь нет никаких особенностей, все пишется ровно так, как и написано, лишь с особенностями перевода на tree-sitter, код приведен ниже:

Листинг 3.12. Конструкции объявления функций и параметров

```
source: $ => repeat(
  field('sourceItem', $.sourceItem),
),
sourceItem: $ => field('funcDef', $.funcDef),

funcDef: $ => seq(
  'method',
  field('funcSignature', $.funcSignature),
  choice(
    field('body', $.body),
    ';',
  )
),

body: $ => seq(
  optional(seq(
    'var',
    repeat(seq(
      optional(field('list_identifier', $.list_identifier)),
      optional(seq(
        ':',
```

```
        field('typeRef', $.typeRef)
      )),
      ';'
    ))
  )),
  field('statment_block', $.block_content)
),

list_identifier: $ => seq(
  field('identifier', $.identifier),
  repeat(seq(',', field('identifier', $.identifier)))
),

funcSignature: $ => seq(
  field('identifier', $.identifier),
  '(',
  optional(field('list_argDef', $.list_argDef)),
  ')',
  optional(seq(
    ':',
    field('typeRef', $.typeRef),
  ))
),

list_argDef: $ => seq(
  field('argDef', $.argDef),
  repeat(seq(',', field('argDef', $.argDef)))
),

argDef: $ => seq(
  field('identifier', $.identifier),
  optional(seq(
    ':',
    field('typeRef', $.typeRef)
  ))
),

typeRef: $ => choice(
  alias(choice(
    'bool',
    'byte',
    'int',
    'uint',
    'long',
    'ulong',
    'char',
    'string',
  ), 'builtin'),
  field('custom', $.identifier),
  field('array', $.array)
),

array: $ => seq(
  'array',
  '[',
  repeat(','),
  ']',
  'of',
  $.typeRef,
),
```

Приведенная грамматика генерирует практически точное дерево для задания, за несколькими отличиями:

1. В названиях полей запрещены точки и скобки, из-за этого несколько названий требуется переименовать.
2. В tree-sitter нельзя сделать полностью пустой узел, который бы отобразился в дереве, из-за этого возникает проблема с отображением пустого списка.

Обе эти проблемы необходимо решать через тестовую программу, которая будет принимать грамматику и выводить дерево разбора, вместе с ошибками, возникшими в ходе разбора переданного файла.

3.2. Разработка тестовой программы

Tree-sitter был выбран по причине того, что результатом его разбора является дерево, что ускоряет процесс построения итогового результата. Тестовую программу было решено делать на python из-за его удобства при написании скриптов.

Первый этап – получение аргументов командной строки, для этого используется следующая функция:

Листинг 3.13. Функция для получения данных из командной строки

```
def parse_cli():
    p = argparse.ArgumentParser(description="Сборка и запуск tree-sitter парсера")
    p.add_argument(
        "--lib",
        dest="lib_path",
        help="Путь к уже скомпилированной tree-sitter библиотеке (.so/.dll/.dylib). "
             "В этом случае grammar_dir и lang_name не нужны.",
    )
    p.add_argument("grammar_dir", nargs="?", help="Путь к папке с грамматикой (library)")
    p.add_argument("lang_name", nargs="?", help="Имя парсера, напр. 'foo' для tree_sitter_foo")
    p.add_argument("file_path", help="Путь к файлу для парсинга")
    p.add_argument("out_file_path", help="Путь к выходному файлу")
    a = p.parse_args()
    if a.lib_path:
        grammar_dir = None
        stem = Path(a.lib_path).stem  # e.g. libfoo -> libfoo, foo -> foo
        lang_name = stem[3:] if stem.startswith("lib") else stem
        lib_path = a.lib_path
    else:
        if not a.grammar_dir or not a.lang_name:
            p.error("Нужно указать либо --lib <путь_к_библиотеке>, либо grammar_dir и
lang_name.")
        grammar_dir = a.grammar_dir
        lang_name = a.lang_name
        lib_path = None

    return grammar_dir, lang_name, a.file_path, a.out_file_path, lib_path
```

Функция, приведенная выше может принимать грамматику в двух формах:

1. В виде пути к папке с грамматикой и её названием;
2. В виде непосредственно пути к файлу грамматики.

Такой подход позволяет упростить отладку (не требуется перекомпиляция т.к. происходит в автоматическом режиме), но и соответствовать заданию (тестовый скрипт должен принимать на вход файл грамматики).

В случае, если библиотеку не передали, требуется её скомпилировать, делается это следующей функцией:

Листинг 3.14. Функция для компиляции лексического анализатора

```
def build_parser(grammar_dir: str, lang_name: str) -> str:
    """Собрать shared-библиотеку в <grammar_dir>/build/<lang_name>.(dll|so|dylib)."""
    out_dir = os.path.join(grammar_dir, "build")
    os.makedirs(out_dir, exist_ok=True)
    out_path = os.path.join(out_dir, f"{lang_name}.dll")

    subprocess.run(
        ["tree-sitter", "generate", "--abi", str(tree_sitter.LANGUAGE_VERSION)],
        cwd=grammar_dir,
        check=True,
    )
    subprocess.run(
        ["tree-sitter", "build", "-o", out_path], cwd=grammar_dir, check=True
    )
    return out_path
```

Следующим этапом является открытие скомпилированной библиотеки и получение корня дерева для её обхода:

Листинг 3.15. Функция для получения корня дерева

```
def load_and_parse(lib_path: str, lang_name: str, file_path: str):
    """Загрузить язык из DLL, распарсить файл, вернуть корневой узел."""
    cdll = CDLL(os.path.abspath(lib_path))
    func_name = f"tree_sitter_{lang_name}"
    if not hasattr(cdll, func_name) and hasattr(cdll, func_name + "_language"):
        func_name = func_name + "_language"
    func = getattr(cdll, func_name)
    func.restype = c_void_p
    ptr = func()

    PyCapsule_New = PYFUNCTYPE(py_object, c_void_p, c_char_p, c_void_p)(
        ("PyCapsule_New", pythonapi)
    )
    capsule = PyCapsule_New(ptr, b"tree_sitter.Language", None)
    lang = Language(capsule)

    parser = Parser(lang)
    with open(file_path, "rb") as f:
        data = f.read()
    tree = parser.parse(data, encoding="utf8")
    return tree.root_node
```

Последняя функция разбирает дерево и выводит его в удобочитаемой форме.

```
def write_tree_to_file(node, out_path: str | Path, *, ascii: bool = False) -> None:
    """
    Пишет вывод дерева в файл out_path.
    Ветвление оформлено как в `tree`. Логика печати полностью соответствует исходной.
    """
    VBAR, SPACE, TEE, ELBOW = ("|   ", "    ", "|-- ", "`-- ") if ascii else ("│   ", "    ",
"├── ", "└── ")

    replace_names = {
        "list_argDef": "list<argDef>",
        "list_identifier": "list<identifier>",
        "statment_block": "statment.block",
        "listExpr": "list<expr>",
    }

    def _prefix(anc_has_next: list[bool], is_last: bool) -> str:
        if not anc_has_next:
            return ""

        base = "".join(VBAR if has_next else SPACE for has_next in anc_has_next)
        return base + (ELBOW if is_last else TEE)

    def _child_anc(anc_has_next: list[bool], is_last: bool) -> list[bool]:
        return [*anc_has_next, not is_last]

    def _walk(n, indent: int, anc_has_next: list[bool], is_last: bool, out):
        fname = field_name_of(n)
        is_token = n.type in (
            "identifier",
            "bin_op",
            "str",
            "char",
            "hex",
            "bits",
            "dec",
            "un_op",
            "builtin",
        )
        if fname in replace_names:
            fname = replace_names[fname]

        if is_token:
            out.write(f"{_prefix(anc_has_next, is_last)}{fname}\n")

        if n.is_missing:
            print(f"Error: missing element \"{n.type}\" in end point {n.end_point}",
file=sys.stderr)
        if n.is_error:
            print(f"Error: incorrect in end point {n.end_point}", file=sys.stderr)

        if len(n.children) == 0:
            text = n.text.decode("utf-8")
            if is_token:
                child_pref = _prefix(_child_anc(anc_has_next, is_last), True)
                out.write(f'{child_pref}"{text}"\n')
            else:
                if not anc_has_next:
                    out.write(f"{_prefix(anc_has_next, is_last)}{fname}\n")
                else:
                    out.write(f'{_prefix(anc_has_next, is_last)}"{text}"\n')
        else:
            out.write(f"{_prefix(anc_has_next, is_last)}{fname}\n")
```

```
        child_anc = _child_anc(anc_has_next, is_last)

        inject_cfg = {
            "funcSignature": ("list_argDef", "("),
            "call_expression": ("listExpr", "("),
            "indexer": ("listExpr", "["),
        }
        need_field, left_token = inject_cfg.get(n.type, (None, None))
        present_fields = {n.field_name_for_child(i) for i in range(n.child_count)} - {None}
        should_inject = bool(need_field) and (need_field not in present_fields)
        injected = False

        for i, ch in enumerate(n.children):
            _walk(ch, indent + 1, child_anc, i == len(n.children) - 1, out)

            if (should_inject and not injected and ch.type == left_token):
                display = replace_names.get(need_field, need_field)
                out.write(f"{_prefix(child_anc, False)}{display} (empty)\n")
                injected = True

    out_path = Path(out_path)
    out_path.parent.mkdir(parents=True, exist_ok=True)
    with out_path.open("w", encoding="utf-8") as f:
        _walk(node, 0, [], True, f)
```

Самый важный тут момент, это функция *_walk*, которая непосредственно «рисует» дерево в файл. Грубо говоря здесь приведен обычный обход дерева, который лишь исправляет «косяки» лексического анализатора.

Первым делом он обрабатывает имя очередного ребенка и, при необходимости, его обновляет (таким образом получается добавить различные стрелочки и точки в названия). Потом обрабатываются токены, те элементы, которые изначально не выводятся, чтоб вместо:

$$typeRef -> custom -> abc$$

был добавлен промежуточный

$$typeRef -> custom -> identifier -> abc.$$

Последующая обработка – проверяет, не является ли данный элемент ошибкой или пропуском и при необходимости выводит информацию об этом. После этого проверяет, а не лист ли это, чтоб вывести его значение.

Последняя обработка занимается вставкой пустого списка, который опускается при лексическом разборе, если его нет.

В итоге, функция верхнего уровня для разбора выглядит следующим образом:

Листинг 3.16. Функция верхнего уровня для разбора входного файла

```
def main():
    grammar_dir, lang_name, file_path, out_file_path, lib_path = parse_cli()

    if lib_path:
        root = load_and_parse(lib_path, lang_name, file_path)
    else:
        built_lib = build_parser(grammar_dir, lang_name)
        root = load_and_parse(built_lib, lang_name, file_path)

    write_tree_to_file(root, out_file_path)
```

После этого можно переходить непосредственно к тестированию.

3.3. Тестирование разработанной грамматики

Для тестирования грамматики будем подавать различные входные файлы и смотреть на результат. Если он соответствует ожиданиям, значит все работает корректно.

Подадим следующий файл:

Листинг 3.17. Файл, подающийся в тестовую программу

```
method hello_world()
begin
    b + 3;
    !g;
    (3);
    a();
    a[a,b,c];
    asd;
    true; false;
    "asd";
    'a';
    0x123;
    0b1010;
    123;

    if a then a;
    if a then a; else a;
    begin end;
    begin begin end; begin end; end;
    while a do begin end;
    repeat begin end; while a;
    repeat begin end; until 1;
    break;

    if a then
        if b then
            a = 2;
        else
            b = 2;

    (ok := !false) * (x := 10) - 1;
end;
```

Приведенный файл демонстрирует все возможные наборы элементов Exr, Structure, а также сложное скобочное выражение и пример для «висячего else».

Запустим код и получим файл разбора. Буду приводить его по частям. Сначала заголовок, с объявлением функции:

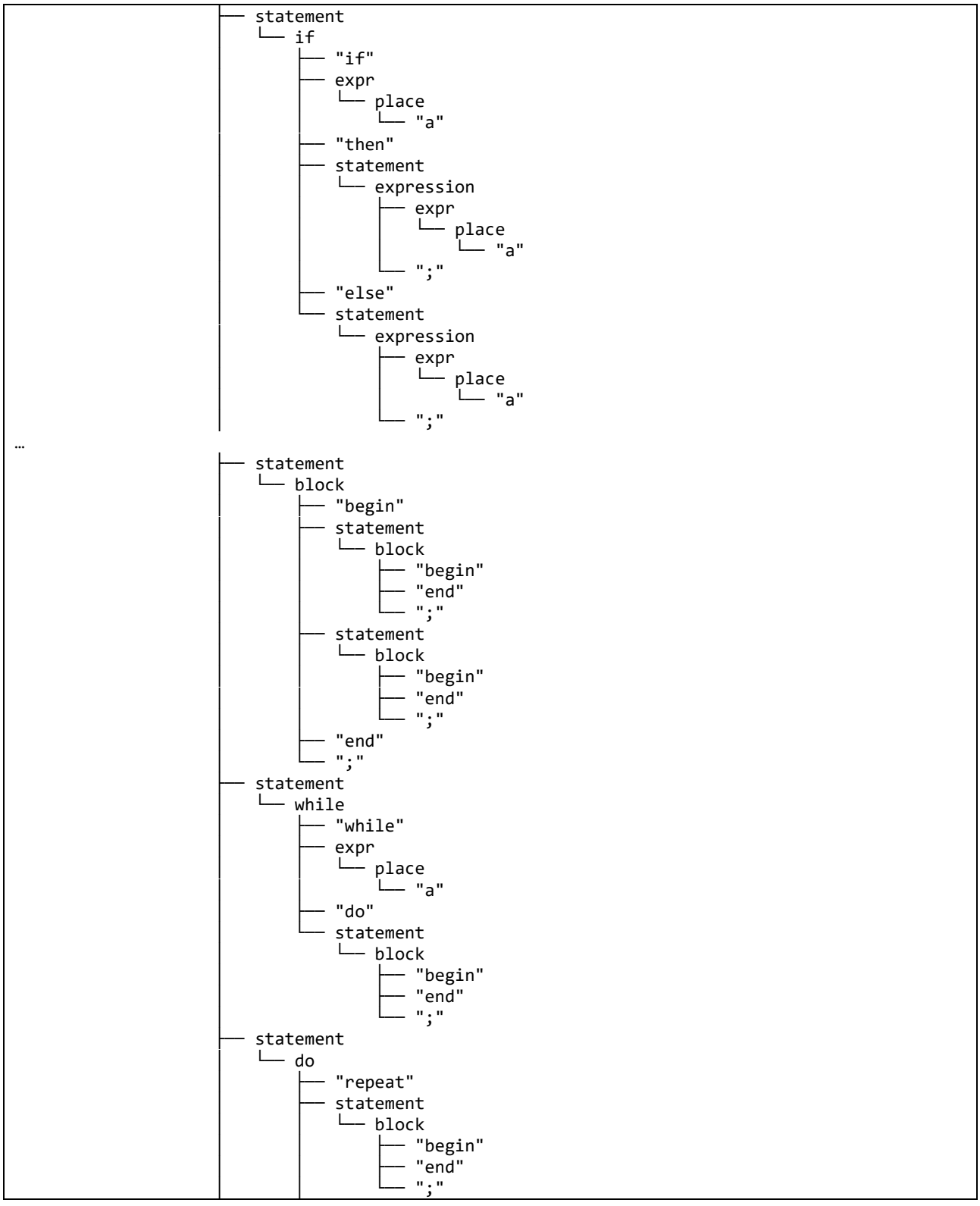
```
source
    └── sourceItem
        └── funcDef
            ├── "method"
            ├── funcSignature
            │   ├── identifier
            │   │   └── "hello_world"
            │   ├── "("
            │   ├── list<argDef> (empty)
            │   └── ")"
            └── body
                └── statment.block
```

Видно, что дерево построено корректно, для случая, когда перед началом выполнения функции нет объявлений переменных, а также сама функция ничего не принимает.

```
                    ├── "begin"
                    ├── statement
…
                    │   └── expression
                    │       ├── expr
                    │       │   └── call
                    │       │       ├── expr
                    │       │       │   └── place
                    │       │       │       └── "a"
                    │       │       ├── "("
                    │       │       ├── list<expr> (empty)
                    │       │       └── ")"
                    │       └── ";"
                    ├── statement
                    │   └── expression
                    │       ├── expr
                    │       │   └── indexer
                    │       │       ├── expr
                    │       │       │   └── place
                    │       │       │       └── "a"
                    │       │       ├── "["
                    │       │       ├── list<expr>
                    │       │       │   ├── expr
                    │       │       │   │   └── place
                    │       │       │   │       └── "a"
                    │       │       │   ├── ","
                    │       │       │   ├── expr
                    │       │       │   │   └── place
                    │       │       │   │       └── "b"
                    │       │       │   ├── ","
                    │       │       │   └── expr
                    │       │       │       └── place
                    │       │       │           └── "c"
                    │       │       └── "]"
                    │       └── ";"
```

Здесь я привел лишь малую часть вывода программы, подробнее можно ознакомиться в репозитории в папке Examples (all_structures_and_expr). По результатам видно, что все выражения expr без исключений выполняются корректно.

Далее идут различные statement:



```
        ├── "while"
        ├── expr
        │   └── place
        │       └── "a"
        └── ";"
```

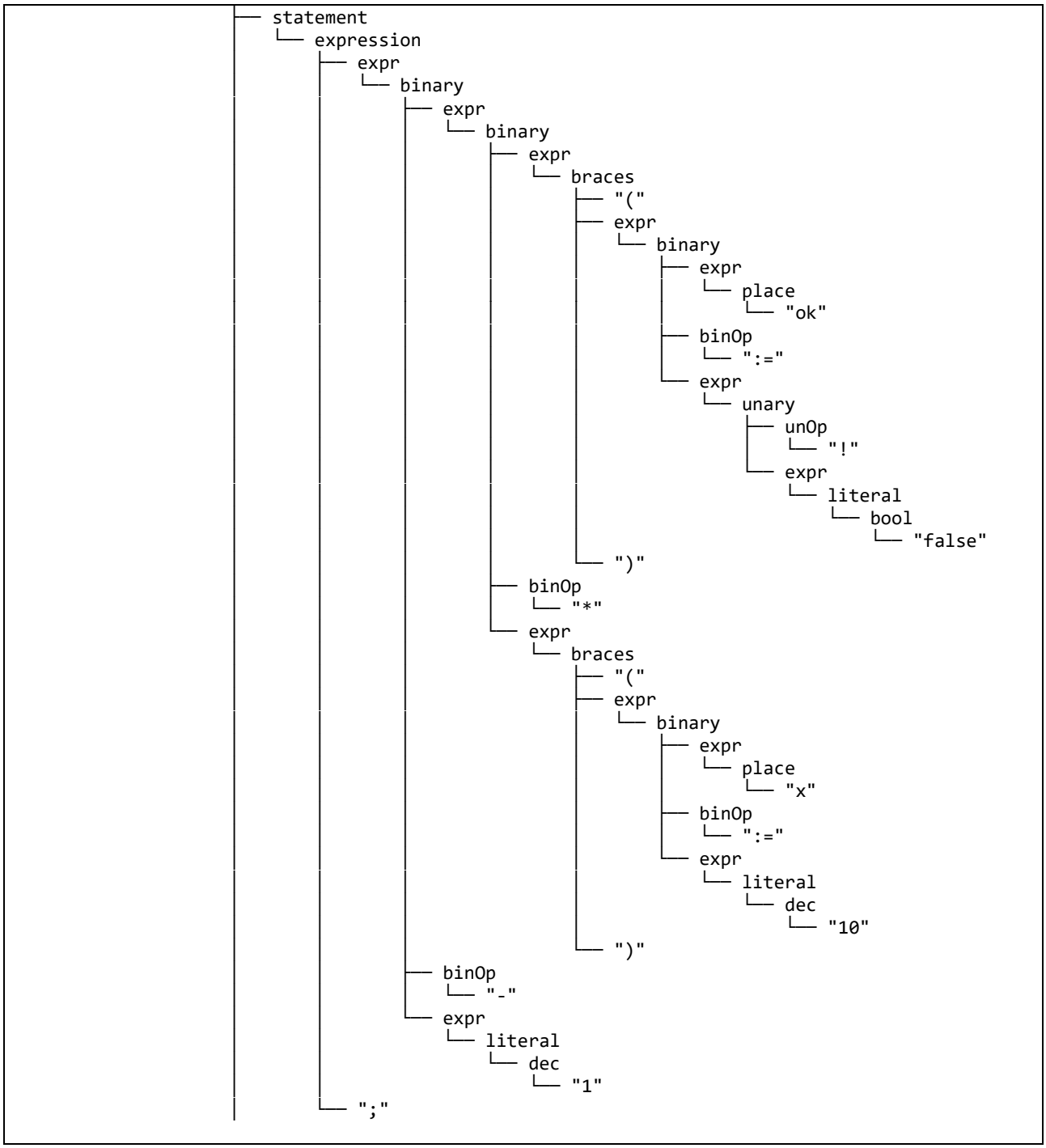
Выше также пропущены незначительные блоки, однако важно отметить, что все работает корректно и безошибочно выводится то дерево, которое требовалось.

Теперь перейдем к более интересным случаям, а именно к «висящему» else:

```
├── statement
│   └── if
│       ├── "if"
│       ├── expr
│       │   └── place
│       │       └── "a"
│       ├── "then"
│       └── statement
│           └── if
│               ├── "if"
│               ├── expr
│               │   └── place
│               │       └── "b"
│               ├── "then"
│               ├── statement
│               │   └── expression
│               │       ├── expr
│               │       │   └── binary
│               │       │       ├── expr
│               │       │       │   └── place
│               │       │       │       └── "a"
│               │       │       ├── binOp
│               │       │       │   └── "="
│               │       │       └── expr
│               │       │           └── literal
│               │       │               └── dec
│               │       │                   └── "2"
│               │       └── ";"
│               ├── "else"
│               └── statement
│                   └── expression
│                       ├── expr
│                       │   └── binary
│                       │       ├── expr
│                       │       │   └── place
│                       │       │       └── "b"
│                       │       ├── binOp
│                       │       │   └── "="
│                       │       └── expr
│                       │           └── literal
│                       │               └── dec
│                       │                   └── "2"
│                       └── ";"
```

Здесь нужно обратить внимание, что else идет к «ближайшему» else, что соответствует документации СИ, поскольку достоверно неизвестно, какой именно результат ожидается для данного языка, будем считать, что задача решена корректно.

И последний интересный блок: скобочное выражение.



По дереву видно, что порядок действий примерно такой:

1. Вычислить !false;
2. Присвоить !false в ok;
3. Присвоить x значение 10;
4. Перемножить результаты;
5. Вычесть 1.

Этот порядок полностью соответствует правильному, что свидетельствует о корректности разбора различных унарных и бинарных выражений.

Далее проведем тестирование на пустом файле, ожидается, что будет только пустой source, что мы и получили.

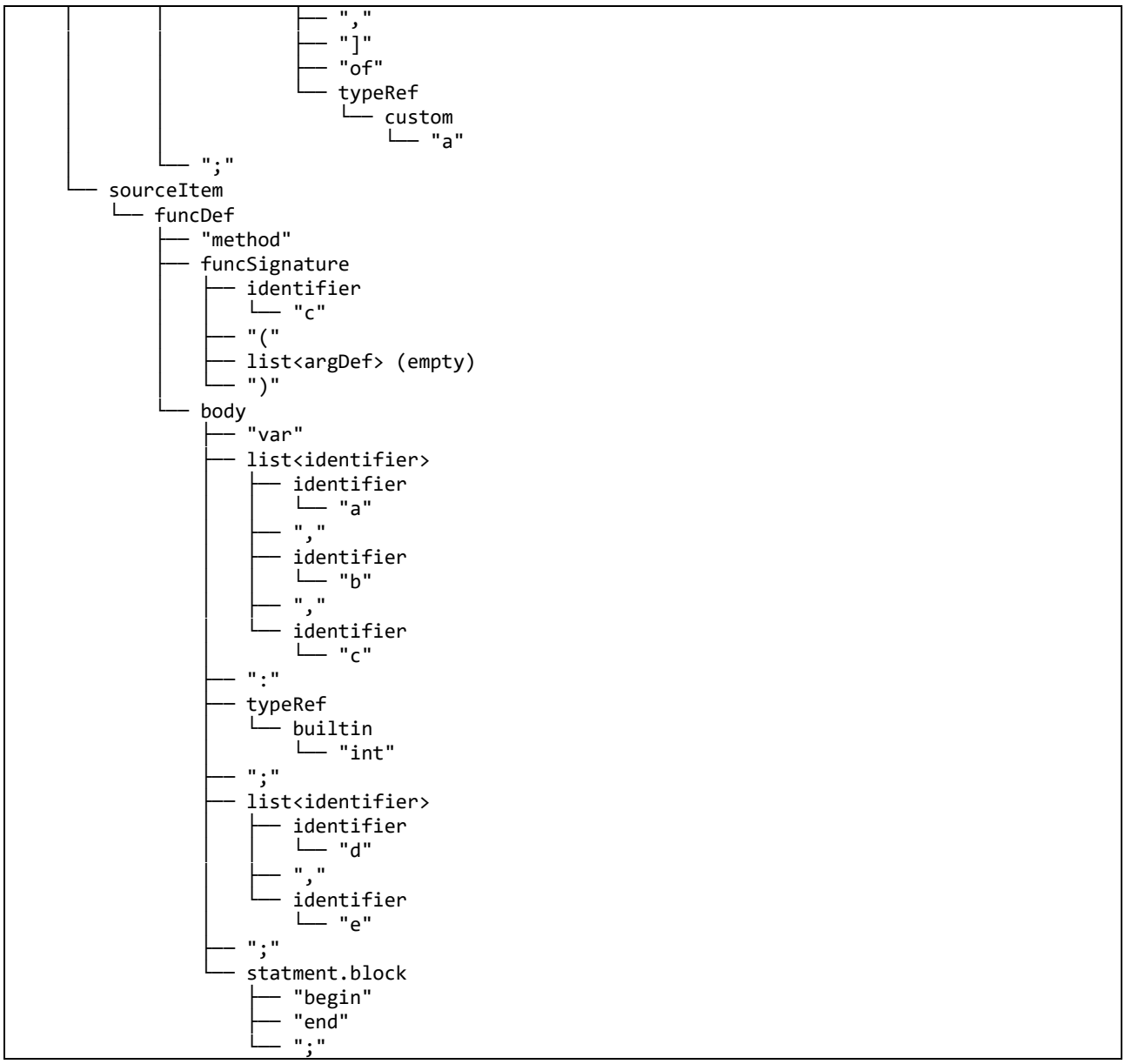
Третий тест будет проходить над следующим файлом:

Листинг 3.18. Тестирование корректности работы обработчика функций

```
method hello_world();
method hello_world_2(a, b: int, c: a): array[,,] of a;
method c()
var a,b,c:int; d,e; begin end;
```

Запустим данный тест. Результат приведен ниже:

```
source
    ├── sourceItem
    │   └── funcDef
    │       ├── "method"
    │       ├── funcSignature
    │       │   ├── identifier
    │       │   │   └── "hello_world"
    │       │   ├── "("
    │       │   ├── list<argDef> (empty)
    │       │   └── ")"
    │       └── ";"
    ├── sourceItem
    │   └── funcDef
    │       ├── "method"
    │       ├── funcSignature
    │       │   ├── identifier
    │       │   │   └── "hello_world_2"
    │       │   ├── "("
    │       │   ├── list<argDef>
    │       │   │   ├── argDef
    │       │   │   │   └── identifier
    │       │   │   │       └── "a"
    │       │   │   ├── ","
    │       │   │   ├── argDef
    │       │   │   │   ├── identifier
    │       │   │   │   │   └── "b"
    │       │   │   │   ├── ":"
    │       │   │   │   └── typeRef
    │       │   │   │       └── builtin
    │       │   │   │           └── "int"
    │       │   │   ├── ","
    │       │   │   └── argDef
    │       │   │       ├── identifier
    │       │   │       │   └── "c"
    │       │   │       ├── ":"
    │       │   │       └── typeRef
    │       │   │           └── custom
    │       │   │               └── "a"
    │       │   ├── ")"
    │       │   ├── ":"
    │       │   └── typeRef
    │       │       └── array
    │       │           ├── "array"
    │       │           ├── "["
    │       │           ├── ","
```



Здесь видно, что запись нескольких функций работает корректно, как и объявление переменных, их параметров и типа возвращаемых значений.

Таким образом было проведено полное тестирование разработанной грамматики.

4. Вывод

В ходе выполнения лабораторной работы был спроектирован и реализован синтаксический анализатор для заданного языка программирования с использованием библиотеки tree-sitter. В процессе работы была изучена структура и принципы построения грамматик на данном инструменте, реализованы все необходимые элементы языка: выражения, операторы,

конструкции ветвления и циклов, а также механизм объявления функций и переменных. Полученная грамматика была протестирована на ряде примеров, демонстрирующих корректную работу парсера для всех предусмотренных структур и выражений, включая вложенные и сложные синтаксические случаи.

Созданная тестовая программа на Python позволила автоматически компилировать и запускать парсер, строить синтаксическое дерево и выводить его в удобочитаемом виде. Проведённые испытания подтвердили корректность работы как самого анализатора, так и алгоритма визуализации дерева разбора.

Таким образом, цель лабораторной работы достигнута. Создан работоспособный модуль синтаксического анализа, формирующий корректное синтаксическое дерево по исходному коду заданного языка и обеспечивающий возможность дальнейшего анализа структуры программы.

5. Исходные файлы

Исходные файлы проекта: репозиторий [Электронный ресурс]. — GitHub. — Режим доступа: https://github.com/DafterT/parsing_grammar (дата обращения: 20.10.2025).