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](#_Toc210574064)

[2. План работы 2](#_Toc210574065)

[3. Инструментальные средства 2](#_Toc210574066)

[4. Ход работы 2](#_Toc210574067)

[4.1. Описание алгоритма 2](#_Toc210574068)

[4.2. Реализация модуля на SystemVerilog 3](#_Toc210574069)

[4.3. Симуляционное тестирование 8](#_Toc210574070)

[4.4. Характеристики разработанного устройства 10](#_Toc210574071)

[4.5. Тестирование проекта на ПЛИС 11](#_Toc210574072)

[5. Вывод 12](#_Toc210574073)

# 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.

# 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.

# Ход работы

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 .. 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 .. 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 .. 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 .. 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 .. 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 .. 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 .. 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 .. 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 .. 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 .. 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 c отображением пустого списка.