Simple C-Style Language
Документация проекта

January 9, 2026

Contents

1 Введение

1.1 О проекте

Simple C-Style Language — это минималистичный образовательный язык программирования в стиле C с поддержкой:

- 32-битных беззнаковых целых чисел как единственного типа данных
- Функций с параметрами и возвращаемыми значениями
- Циклов `for` и `while`
- Условных операторов (`if/else`)
- Арифметических и логических операций

1.2 Требования

- Python 3.7 или выше
- Дополнительные зависимости не требуются

1.3 Использование

Запуск программы:

```
python main.py <source_file>
```

Программа выполнится и выведет возвращаемое значение функции `main`.
Пример:

```
python main.py examples/factorial.sc
```

1.4 Компиляция

Компиляция исходного файла в FASM ассемблер, затем в бинарный файл:

```
python compile.py <source_file> [output_file] [--run]
```

Опции:

- `output_file` — опциональный путь к выходному файлу `.asm` (по умолчанию: `<source_file>.asm`)
- `-run` — после компиляции запустить бинарный файл с помощью `interpreter_x64.exe` из `int_pack`

Примеры:

```
#

python compile.py examples/basic/sum_range.sc

#
python compile.py examples/basic/sum_range.sc --run
```

Это генерирует:

- `.asm` файл — исходный код FASM ассемблера
- `.bin` файл — бинарный исполняемый файл (скомпилированный с `int_pack/FASM.EXE`)
- `.mif` файл — файл инициализации памяти для Quartus

2 Спецификация языка

2.1 Обзор

Это простой язык программирования в стиле C с минимальным набором функций. Язык разработан для образовательных целей и фокусируется на основных концепциях программирования с ограниченной системой типов.

2.2 Основные возможности

- **Единственный тип данных**: только 32-битные беззнаковые целые числа (`uint32`)
- **Функции**: пользовательские функции с параметрами и возвращаемыми значениями
- **Управление потоком**: циклы `for` и `while`
- **Выражения**: арифметические, логические и побитовые операции
- **Операторы**: объявления переменных, присваивания, вызовы функций и управление потоком
- **Включение файлов**: директива `#include` для модульной организации кода
- **Поддержка железа**: доступ к регистрам, GPIO, UART, операции таймера
- **Прерывания**: обработчики прерываний (ISR)
- **Побитовые операции**: встроенные функции для побитовых операций

2.3 Типы данных

2.3.1 Беззнаковое 32-битное целое число (uint32)

Единственный тип данных в языке. Все переменные, параметры функций и возвращаемые значения неявно являются `uint32`.

- Диапазон: от 0 до 4,294,967,295 ($2^{32} - 1$)
- Поведение при переполнении: зацикливается (по модулю 2^{32})

2.4 Лексические элементы

2.4.1 Идентификаторы

- Должны начинаться с буквы или подчёркивания
- Могут содержать буквы, цифры и подчёркивания
- Регистрозависимые
- Примеры: `x`, `myVar`, `_count`, `counter123`

2.4.2 Литералы

- **Десятичные литералы**: последовательности цифр (0-9)
  - Примеры: `0`, `42`, `1000`, `4294967295`
- **Шестнадцатеричные литералы**: с префиксом `0x` или `0X`, за которым следуют шестнадцатеричные цифры (0-9, A-F, a-f)
  - Примеры: `0x0`, `0xFF`, `0x10`, `0X1A2B`, `0xABCDEF`
  - Поддерживаются как заглавные, так и строчные буквы A-F
  - Допускается смешанный регистр: `0xAa`, `0XbB`

2.4.3 Ключевые слова

```
uint32    (                                        ,
             uint32)
function  (                                      )
for       (          for)
while     (          while)
if        (                                   )
else      (          else)
return    (                                   )
register  (                                              )
volatile  (                           volatile)
interrupt (
                          )
```

2.4.4 Операторы

Арифметические:

- `+` (сложение)
- `-` (вычитание)
- `*` (умножение)
- `/` (целочисленное деление)
- `%` (модуло)

Отношения:

- `==` (равенство)
- `!=` (неравенство)
- `<` (меньше)
- `<=` (меньше или равно)
- `>` (больше)

- `>=` (больше или равно)

Логические:

- `&&` (логическое И)
- `||` (логическое ИЛИ)
- `!` (логическое НЕ)

Побитовые:

- `&` (побитовое И)
- `|` (побитовое ИЛИ)
- `^` (побитовое исключающее ИЛИ)
- `~` (побитовое НЕ)
- `«` (сдвиг влево)
- `»` (сдвиг вправо)

Присваивание:

- `=` (присваивание)
- `++` (инкремент)
- `-` (декремент)

2.5 Синтаксис

2.5.1 Структура программы

Программа состоит из серии определений функций. Точка входа — функция `main`, которая не принимает параметры.

```
function main() {
    //
}
```

2.5.2 Объявление переменной

Переменные должны быть объявлены с ключевым словом `uint32` перед использованием.

```
uint32 x;
uint32 y = 42;
uint32 z = x + y;
```

2.5.3 Присваивание

Переменные присваиваются с помощью оператора `=`.

```
x = 10;
x = x + 1;
x = y * 2;
```

2.5.4 Инкремент и декремент

Переменные могут быть увеличены или уменьшены с помощью операторов `++` и `--`. Поддерживаются как префиксная, так и постфиксная формы.

```
uint32 x = 5;
++x;     //                                    : x
                          6
x++;     //                                      : x
                          7
--x;     //                                    : x
                          6
x--;     //                                      : x
                          5
```

Операторы инкремента и декремента также могут использоваться в приращениях цикла `for`:

```
for (uint32 i = 0; i < 10; i++) {
    //
}

for (uint32 i = 10; i > 0; i--) {
    //
}
```

2.5.5 Выражения

Выражения комбинируют литералы, переменные и операторы. Приоритет операторов следует стандартным правилам C:

1. Скобки
2. Унарные операторы (`!`, `-`)
3. Мультипликативные (`*`, `/`, `%`)
4. Аддитивные (`+`, `-`)
5. Отношения (`<`, `<=`, `>`, `>=`)
6. Равенство (`==`, `!=`)
7. Логическое И (`&&`)
8. Логическое ИЛИ (`||`)

2.5.6 Функции

Функции объявляются с ключевым словом `function`, за которым следует имя функции, параметры и тело.

```
function functionName(param1, param2, ...) {
    //
    return value;
}
```

- Имена функций следуют правилам идентификаторов
- Параметры неявно являются `uint32` (не включайте ключевое слово `uint32` в список параметров)
- Функции могут возвращать значение с помощью `return`
- Если оператор `return` не выполняется, функция возвращает 0
- Функции могут вызываться до их определения (поддержка предварительных объявлений)

2.5.7 Условные операторы

```
if (condition) {
    //
}

if (condition) {
    //
} else {
    //
}
```

2.5.8 Цикл while

```
while (condition) {
    //
}
```

2.5.9 Цикл for

Цикл `for` следует синтаксису в стиле C:

```
for (initialization; condition; increment) {
    //
}
```

- `initialization`: выполняется один раз перед циклом (обычно присваивание переменной)

- `condition`: вычисляется перед каждой итерацией
- `increment`: выполняется после каждой итерации (обычно обновление переменной)

Пример:

```
for (uint32 i = 0; i < 10; i = i + 1) {
    //
}
```

2.6 Грамматика (BNF-like)

```
program         := function_def*
function_def    := 'function' IDENTIFIER '(' param_list? ')' '{' statement* '}'
param_list      := IDENTIFIER (',' IDENTIFIER)*

statement       := var_decl ';'
                 | assignment ';'
                 | function_call ';'
                 | return_stmt ';'
                 | if_stmt
                 | while_stmt
                 | for_stmt
                 | block

var_decl        := 'uint32' IDENTIFIER ('=' expression)?
assignment      := IDENTIFIER '=' expression
function_call   := IDENTIFIER '(' expr_list? ')'
return_stmt     := 'return' expression?

if_stmt         := 'if' '(' expression ')' statement ('else' statement)?
while_stmt      := 'while' '(' expression ')' statement
for_stmt        := 'for' '(' (var_decl | assignment)? ';' expression? ';' assignment? ')' statement

block           := '{' statement* '}'
expr_list       := expression (',' expression)*

expression      := logical_or
logical_or      := logical_and ('||' logical_and)*
logical_and     := equality ('&&' equality)*
equality        := relational (('==' | '!=') relational)*
relational      := additive (('<' | '<=' | '>' | '>=') additive)*
additive        := multiplicative (('+' | '-') multiplicative)*
multiplicative  := unary (('*' | '/' | '%') unary)*
unary           := ('!' | '-') unary | primary
primary         := IDENTIFIER
                 | LITERAL
                 | '(' expression ')'
                 | function_call
```

2.7 Возможности железа

Язык включает встроенную поддержку периферийных устройств MCU.

2.7.1 Доступ к регистрам

CPU регистры (r0-r31) могут быть доступны напрямую:

```
register uint32 r0 = 10;
register uint32 r1 = 20;
register uint32 r2 = r0 + r1;   // r2 = 30
```

- Регистровые переменные должны быть названы `r0` через `r31`
- Регистр r31 (указатель инструкций) только для чтения
- Регистры неявно volatile

2.7.2 Операции GPIO

Управление GPIO пинами для цифрового ввода/вывода:

```
// 0
gpio_set(0, GPIO_OUTPUT, GPIO_NONE);

// GPIO
gpio_write(0, GPIO_HIGH);

// GPIO
uint32 value = gpio_read(0);
```

Функции:

- `gpio_set(pin, direction, mode)` — настройка GPIO пина
- `gpio_read(pin)` — чтение значения GPIO пина
- `gpio_write(pin, value)` — запись значения GPIO пина

2.7.3 Операции UART

Последовательная связь через UART:

```
//
uart_set_baud(115200);

//
uint32 status = uart_get_status();

// UART
uint32 data = uart_read();

// UART
uart_write(data);
```

Функции:

- `uart_set_baud(baud_rate)` — установка скорости UART
- `uart_get_status()` — получение статуса UART
- `uart_read()` — чтение байта из UART
- `uart_write(data)` — запись байта в UART

2.7.4 Операции таймера

Управление аппаратным таймером:

```
//
timer_set_mode(TIMER_PERIODIC);
timer_set_period(1000000);  // 1

timer_start();
```

```
5
6  //
7  if (timer_expired()) {
8      timer_reset();
9  }
```

Функции:

- `timer_set_mode(mode)` — установка режима таймера (ONESHOT, PERIODIC, CONTINUOUS)
- `timer_set_period(microseconds)` — установка периода таймера
- `timer_start()` — запуск таймера
- `timer_stop()` — остановка таймера
- `timer_reset()` — сброс счётчика таймера
- `timer_get_value()` — получение текущего значения таймера
- `timer_expired()` — проверка истечения таймера

2.7.5 Обработчики прерываний

Обработка аппаратных прерываний:

```
1  volatile uint32 counter = 0;
2
3  interrupt function timer_isr() {
4      counter++;
5      timer_reset();
6  }
7
8  function main() {
9      timer_set_mode(TIMER_PERIODIC);
10     timer_set_period(1000000);
11     timer_start();
12     enable_interrupts();
13
14     while (counter < 10) {
15         //
16     }
17
18     disable_interrupts();
19     return counter;
20 }
```

2.7.6 Побитовые операции

Встроенные функции для побитовых операций:

```
uint32 value = 0;

value = set_bit(value, 5);       //                                  5
value = clear_bit(value, 5);     //                               5
value = toggle_bit(value, 3);    //                                  3
uint32 bit = get_bit(value, 5);  //
                                 5

//

uint32 mask = 0xFF;               // 8
uint32 flags = 0x0F;              //               4
uint32 result = value & mask;     //
```

Функции:

- `set_bit(value, bit)` — установить бит
- `clear_bit(value, bit)` — сбросить бит
- `toggle_bit(value, bit)` — переключить бит
- `get_bit(value, bit)` — получить значение бита (0 или 1)

2.7.7 Функции задержки

Программные задержки:

```
delay_ms(100);       //                  100
delay_us(1000);      //                  1000
delay_cycles(1000); //                   N              CPU
```

2.8 Примеры

2.8.1 Пример 1: Простая программа

```
function main() {
    uint32 x = 10;
    uint32 y = 20;
    uint32 sum = x + y;
    return sum;
}
```

2.8.2 Пример 2: Функция факториала

```
function factorial(n) {
    if (n == 0 || n == 1) {
        return 1;
```

```
    }
    uint32 result = 1;
    uint32 i = 2;
    while (i <= n) {
        result = result * i;
        i = i + 1;
    }
    return result;
}

function main() {
    uint32 n = 5;
    uint32 fact = factorial(n);
    return fact;  //                          120
}
```

2.8.3 Пример 3: Пример с циклом for

```
function sum_range(start, end) {
    uint32 sum = 0;
    uint32 i;
    for (i = start; i <= end; i = i + 1) {
        sum = sum + i;
    }
    return sum;
}

function main() {
    uint32 result = sum_range(1, 10);
    return result;  //                          55
}
```

3 Архитектура системы

3.1 Обзор

Этот документ описывает архитектуру и дизайн интерпретатора Simple C-Style Language.

3.2 Пайплайн компиляции

Язык следует традиционному пайплайну компилятора/интерпретатора:

```
                                (.sc)

[                               ]  #
   #include

                               (
                                )

[              ]           #


[              ]           #


   (AST)

[                                ]  #

```

3.3 Детали компонентов

3.3.1 1. Препроцессор (`preprocessor.py`)

Назначение: Обрабатывает директивы `#include` до лексирования.

Основные возможности:

- Рекурсивное включение файлов
- Обнаружение циклических зависимостей
- Разрешение путей (относительных и абсолютных)
- Вставка комментариев для отладки

Классы:

- `Preprocessor` — основной класс препроцессинга
- `PreprocessingError` — исключение для ошибок препроцессинга

3.3.2 2. Лексер (`lexer.py`)

Назначение: Преобразует исходный код в поток токенов.

Основные возможности:

- Распознавание токенов для всех элементов языка
- Отслеживание строк и столбцов для отчётов об ошибках
- Обработка комментариев (однострочных и многострочных)
- Обработка пробелов

Классы:

- `TokenType` — перечисление всех типов токенов
- `Token` — токен с типом, значением, строкой и столбцом
- `Lexer` — основной класс лексера

3.3.3 3. Парсер (`parser.py`)

Назначение: Строит абстрактное синтаксическое дерево (AST) из токенов.

Основные возможности:

- Рекурсивный нисходящий парсинг
- Обработка приоритета операторов
- Парсинг выражений с правильной ассоциативностью
- Парсинг операторов (объявления, присваивания, управление потоком)

Типы узлов AST:

- `Program` — корневой узел, содержащий все функции
- `FunctionDef` — определение функции
- `VarDecl` — объявление переменной
- `Assignment` — присваивание переменной
- `IfStmt` — оператор if/else
- `WhileStmt` — цикл while
- `ForStmt` — цикл for
- `ReturnStmt` — оператор return
- `Block` — блок кода
- `BinaryOp` — бинарная операция
- `UnaryOp` — унарная операция
- `FunctionCall` — вызов функции
- `Identifier` — ссылка на переменную
- `Literal` — целочисленный литерал

3.3.4 4. Интерпретатор (`interpreter.py`)

Назначение: Выполняет AST и управляет состоянием времени выполнения.
Основные возможности:

- Управление окружением (область видимости переменных)
- Обработка вызовов функций
- Выполнение управления потоком
- Встроенные аппаратные функции
- Симуляция регистров
- Обработка прерываний

Классы:

- `Environment` — управление областью видимости переменных
- `Interpreter` — основной класс интерпретатора
- `RuntimeError` — исключение времени выполнения

Состояние времени выполнения:

- `global_env` — окружение глобальных переменных
- `functions` — определения функций
- `registers` — симуляция CPU регистров (32 регистра)
- `register_map` — отображение имён регистров на индексы
- `interrupt_handlers` — подпрограммы обработки прерываний
- Состояние железа (GPIO, UART, Timer)

3.4 Поток данных

3.4.1 Объявление переменной

```
VarDecl AST

execute_var_decl()

Environment.declare()


```

3.4.2 Вызов функции

```
FunctionCall AST

evaluate_expression()

execute_function()
```

3.5 Модель памяти

3.5.1 Хранение переменных

- Переменные хранятся в объектах `Environment`
- Каждая область видимости имеет своё окружение
- Окружения образуют цепочку (отношение родитель-потомок)
- Глобальные переменные находятся в `global_env`

3.5.2 Хранение регистров

- CPU регистры (r0-r31) хранятся в массиве `interpreter.registers`
- Доступ к регистровым переменным осуществляется через `register_map`
- Доступ к регистрам обходит обычный поиск переменных

4 Механизм включения файлов

4.1 Обзор

В нашем языке файлы `.h` (и любые другие файлы) работают через механизм **препроцессора** (директивы `#include`). Препроцессор обрабатывает исходный код **до** лексического анализа и парсинга, вставляя содержимое включённых файлов напрямую в исходный код.

4.2 Процесс обработки

4.2.1 1. Порядок выполнения

```
                              (.sc)

                             (#include                        )




                (                          )

                (AST)

                                (                       )
```

4.2.2 2. Синтаксис #include

Поддерживаются два формата:

```
#include "filename.h"     //
   (                    )
#include <filename.h>     //
   (                                  )
```

Важно: В текущей реализации оба формата работают одинаково — ищут файл относительно текущего файла или базовой директории.

4.2.3 3. Алгоритм поиска файлов

Препроцессор ищет файлы в следующем порядке:

1. **Абсолютный путь** — если указан полный путь, используется он
2. **Относительно текущего файла** — сначала ищет в директории, где находится файл с `#include`
3. **Относительно базовой директории** — затем ищет в директории основного файла
4. **Текущая рабочая директория** — в конце проверяет текущую директорию

4.2.4 4. Рекурсивная обработка

Препроцессор обрабатывает вложенные `#include` рекурсивно:

main.sc:

```
#include "utils.sc"     //                    utils.sc
#include "math_ops.sc" //                     math_ops.sc

function main() {
    return add(1, 2);  //
       utils.sc
}
```

utils.sc:

```
function add(a, b) {
    return a + b;
}
```

Результат препроцессинга:

```
// Included from: utils.sc
function add(a, b) {
    return a + b;
}
// End include: utils.sc

// Included from: math_ops.sc
// ...                          math_ops.sc ...
// End include: math_ops.sc

function main() {
    return add(1, 2);
}
```

4.2.5 5. Защита от циклических включений

Препроцессор отслеживает уже включённые файлы и предотвращает циклические включения:

circular_a.sc:

```
#include "circular_b.sc"  //                    :

function func_a() {
    return 1;
}
```

circular_b.sc:

```
#include "circular_a.sc"  //                    :

function func_b() {
    return 2;
}
```

Результат: `PreprocessingError: Circular include detected`

4.3 Важные особенности

4.3.1 1. Нет поддержки #define

Важно: В текущей реализации препроцессор **НЕ** обрабатывает директивы `#define`. Файлы с `#define` (например, `hardware.h`) будут включены, но макросы не будут раскрыты.

Решение: Используйте комментарии для документации констант или определяйте их как переменные/функции.

4.3.2 2. Имя файла не важно

Расширение `.h` или `.sc` не имеет значения — препроцессор просто вставляет содержимое файла. Можно использовать:

- `#include "utils.sc"`
- `#include "hardware.h"`
- `#include "constants.txt"` (если это валидный код)

5 Генерация кода

5.1 Обзор

Генератор кода (`codegen.py`) переводит AST (абстрактное синтаксическое дерево) в код FASM ассемблера, нацеленный на ISA, описанную в `isa/README.md`.

5.2 Архитектура

5.2.1 Распределение регистров

Генератор кода использует простую стратегию фиксированного распределения регистров:

- **r0-r10**: Временные регистры (для вычисления выражений)
- **r11-r25**: Локальные переменные
- **r26-r30**: Параметры функций
- **r31**: Указатель инструкций (только для чтения в пользовательском коде, управляется железом)

5.2.2 Соглашение о вызовах функций

- Параметры передаются в r26-r30 (до 5 параметров)
- Возвращаемое значение в r0
- Регистр связи: r30 (для адреса возврата, в правильной реализации)

5.2.3 Пайплайн генерации кода

```
AST

CodeGenerator.generate()

RegisterAllocator (                                          )



          FASM
```

5.3 Статус реализации

5.3.1 Реализовано

- **Выражения**:
  - Литералы
  - Идентификаторы/переменные
  - Бинарные операции (add, sub, and, or, xor, shl, shr)

- Унарные операции (not, logical not, unary minus)
- Операции сравнения (==, !=, <, <=, >, >=)
- Логические операции (&&, ||, !)

- **Операторы**:
  - Объявления переменных
  - Присваивания
  - Операторы return
  - Блоки

- **Аппаратные функции**:
  - GPIO (gpio_set, gpio_read, gpio_write)
  - UART (uart_set_baud, uart_read, uart_write)

- **Функции**:
  - Определения функций
  - Вызовы функций (упрощённые)

5.4 Пример сгенерированного кода

5.4.1 Простое сложение

Исходный код (`add.sc`):

```
function main() {
    uint32 a = 5;
    uint32 b = 3;
    uint32 c = a + b;
    return c;
}
```

Сгенерированный ассемблер (`add.asm`):

```
include "  int_pack/ISA.inc"

func_main:
  ; Function: main
  mov r11, 5
  mov r12, 3
  add r0, r11, r12
  mov r13, r0
  mov r0, r13
  hlt
```

6 Руководство по разработке

6.1 Настройка разработки

1. Клонировать репозиторий
2. Убедиться, что установлен Python 3.7+
3. Дополнительные зависимости не требуются

6.2 Запуск тестов

```
#
python run_tests.py

#
python -m unittest test_lexer
python -m unittest test_parser
python -m unittest test_interpreter
python -m unittest test_preprocessor
```

6.3 Стиль кода

- Следовать руководящим принципам стиля PEP 8
- Использовать подсказки типов, где уместно
- Добавлять строки документации ко всем классам и публичным методам
- Держать функции сфокусированными и небольшими
- Добавлять комментарии для сложной логики

6.4 Добавление новых функций

6.4.1 Добавление нового оператора

1. **Лексер** (`lexer.py`):
   - Добавить тип токена в перечисление `TokenType`
   - Добавить распознавание в метод `tokenize()`
   - Обеспечить правильный приоритет (многознаковые операторы сначала)
2. **Парсер** (`parser.py`):
   - Добавить метод парсинга, если нужен для приоритета
   - Интегрировать в иерархию парсинга выражений
   - Обновить таблицу приоритета операторов
3. **Интерпретатор** (`interpreter.py`):

- Добавить вычисление в `evaluate_binary_op()` или `evaluate_unary_op()`
- Обработать граничные случаи (переполнение, деление на ноль и т.д.)

4. **Тесты**:
   - Добавить тестовые случаи в соответствующий файл тестов
   - Тестировать обычные случаи, граничные случаи и случаи ошибок
5. **Документация**:
   - Обновить `LANGUAGE_SPEC.md`
   - Обновить `README.md`, если это основная функция

7 Структура проекта

7.1 Корневая директория

```
aiproj/
                                                    (            )
                main.py              #
                lexer.py             #                          (                                          )
                parser.py            #               (                 AST)
                interpreter.py       #                              (AST                          )
                preprocessor.py      #                                (                    #include)
                codegen.py           #                                   (AST                         )
                compile.py           #

                README.md            #                                                                (            )
                doc/                 #
                    LANGUAGE_SPEC.md     #
                    ARCHITECTURE.md       #
                    INCLUDE_MECHANISM.md  #                                    #include
                    CONTRIBUTING.md       #
                    CODE_GENERATION.md   #
                    PROJECT_STRUCTURE.md  #

                examples/
                      basic/           #
                            hello_world/ #              Hello World
                      hardware/        #                                          /MCU
                      operators/       #
                      includes/        #
                      advanced/        #
                      README.md        #
                user_examples/       #
                    simple_return/   #
                    complex_example/ #
                    test_example/    #

                self_tests/          #
                    test_lexer.py        #
                    test_parser.py       #
                    test_interpreter.py  #
                    test_preprocessor.py #
                    run_tests.py        #

          int_pack/                #                       FASM
              FASM.EXE             #                    FASM
              interpreter_x64.exe  #
              ISA.inc              #                       ISA
              macros.inc           #                FASM
              ...
```

7.2 Ограничения

- Нет массивов или указателей
- Нет строк или символов
- Нет чисел с плавающей точкой
- Только один тип данных (uint32)
- Деление на ноль вызывает ошибку времени выполнения
- Переполнение целых чисел зацикливается (по модулю 2^{32})
- Нет поддержки макросов `#define` (поддерживается только `#include`)

8 Тестирование

Проект включает комплексные модульные тесты для всех компонентов. Запуск всех тестов:

```
python self_tests/run_tests.py
```

Или запуск отдельных файлов тестов:

```
python -m unittest self_tests.test_lexer
python -m unittest self_tests.test_parser
python -m unittest self_tests.test_interpreter
python -m unittest self_tests.test_preprocessor
```

8.1 Покрытие тестами

- **Тесты лексера** (`test_lexer.py`): распознавание токенов, обработка комментариев, пробелы, операторы, случаи ошибок, отслеживание строк/столбцов
- **Тесты парсера** (`test_parser.py`): определения функций, объявления переменных, все типы операторов, парсинг выражений, вложенные структуры, случаи ошибок
- **Тесты интерпретатора** (`test_interpreter.py`): арифметические операции, область видимости переменных, вызовы функций, управление потоком, инкремент/декремен граничные случаи (переполнение, деление на ноль, неопределённые переменные/функции)
- **Тесты препроцессора** (`test_preprocessor.py`): обработка директив включения, вложенные включения, обнаружение циклических включений, разрешение путей

Все тесты используют встроенный фреймворк `unittest` Python — внешние зависимости не требуются.

9 Инструкции по компиляции в PDF

9.1 Онлайн-компиляция LaTeX в PDF

Этот документ может быть скомпилирован в PDF с помощью различных онлайн-сервисов. Ниже приведены популярные варианты:

9.2 Вариант 1: Overleaf (рекомендуется)

Overleaf — самый популярный онлайн-редактор LaTeX.

1. Перейдите на `https://www.overleaf.com/`
2. Создайте бесплатный аккаунт (если у вас его ещё нет)
3. Нажмите "New Project" "Upload Project"
4. Загрузите файл `documentation.tex`
5. Overleaf автоматически скомпилирует документ
6. Нажмите "Download PDF" для получения готового PDF-файла

Преимущества:

- Не требует установки
- Автоматическая компиляция при изменениях
- Поддержка совместной работы
- Большой выбор шаблонов

9.3 Вариант 2: LaTeX Base

Альтернативный онлайн-редактор LaTeX.

1. Перейдите на `https://latexbase.com/`
2. Вставьте содержимое файла `documentation.tex` в редактор
3. Нажмите "Compile PDF"
4. Скачайте готовый PDF-файл

Преимущества:

- Простой интерфейс
- Не требует регистрации
- Быстрая компиляция

9.4 Вариант 3: ShareLaTeX (сейчас часть Overleaf)

ShareLaTeX был объединён с Overleaf, используйте Overleaf (см. Вариант 1).

9.5 Вариант 4: Papeeria

Ещё один онлайн-редактор LaTeX.

1. Перейдите на `https://papeeria.com/`
2. Создайте проект
3. Загрузите или вставьте содержимое `documentation.tex`
4. Нажмите "Compile"
5. Скачайте PDF

9.6 Вариант 5: CoCalc

Облачная платформа для вычислений с поддержкой LaTeX.

1. Перейдите на `https://cocalc.com/`
2. Создайте проект
3. Загрузите файл `documentation.tex`
4. Откройте файл и используйте встроенный компилятор LaTeX

9.7 Вариант 6: Compile Latex Online

Простейший инструмент для быстрой компиляции.

1. Перейдите на `https://www.compilelatex.com/`
2. Загрузите файл `documentation.tex`
3. Выберите компилятор (pdflatex рекомендуется)
4. Нажмите "Compile"
5. Скачайте PDF

9.8 Локальная компиляция (для справки)

Если вы предпочитаете компилировать локально:

```
#                    TeX Live (Linux/Windows)        MacTeX (macOS)

#
pdflatex documentation.tex
pdflatex documentation.tex  #


#           bibtex (
                           )
pdflatex documentation.tex
bibtex documentation
pdflatex documentation.tex
pdflatex documentation.tex
```

9.9 Примечания по компиляции

- Этот документ использует пакет `babel` с опциями `russian` и `english`. Убедитесь, что выбранный сервис поддерживает русский язык.
- Документ требует пакета `T2A` для правильного отображения кириллицы.
- Если возникают проблемы с компиляцией, попробуйте сервисы, которые явно поддерживают русский язык (например, Overleaf).
- При первом запуске компиляция может занять больше времени из-за загрузки необходимых пакетов.

9.10 Рекомендация

Для этого документа рекомендуется использовать **Overleaf**, так как он:

- Надёжно поддерживает русский язык
- Имеет все необходимые пакеты
- Предоставляет хорошую документацию
- Позволяет легко делиться и редактировать документ

Документ создан автоматически на основе документации проекта Simple C-Style Language