|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |
| Институт искусственного интеллекта |
| Кафедра программного обеспечения систем радиоэлектронной аппаратуры |

КУРСОВАЯ РАБОТА

по «Проектированию трансляторов»

на тему: «Разработка транслятора из Rust в C»

|  |  |  |  |
| --- | --- | --- | --- |
| Обучающийся | \_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_Кочнов Игорь Олегович\_\_\_\_\_\_\_\_ | |
|  | *Подпись* | *Фамилия Имя Отчество* | |
| Шифр | 22К0009 |  |  |
| Группа | КМБО-02-22 |  |  |
|  |  |  |  |
| Руководитель  работы | \_\_\_\_\_\_\_\_\_\_\_\_\_ | *\_\_\_\_\_\_\_*Волков Артём Владиславович*\_\_\_\_\_\_\_\_\_\_\_\_* | |
|  | *Подпись* | *Фамилия Имя Отчество* | |

Москва 2025

Содержание

[1. Общие положения 3](#_Toc196565353)

[1.1. Основания для разработки 3](#_Toc196565354)

[1.2. Задание на курсовой проект 3](#_Toc196565355)

[1.3. Назначение программы 3](#_Toc196565356)

[2. Основные требования к программе 3](#_Toc196565357)

[2.1. Функциональные и нефункциональные требования 3](#_Toc196565358)

[2.1.1. Функциональные требования 3](#_Toc196565359)

[2.1.2. Нефункциональные требования 3](#_Toc196565360)

[2.2. Среда разработки 4](#_Toc196565361)

[3. Описание архитектуры 4](#_Toc196565362)

[3.1. Описание компонентов системы 5](#_Toc196565363)

[3.1.1. Лексический анализатор 5](#_Toc196565364)

[3.1.2. Синтаксический анализатор 7](#_Toc196565365)

[3.1.3. Семантический анализатор 10](#_Toc196565366)

[3.2. Структурная диаграмма классов 10](#_Toc196565367)

[4. Алгоритм работы программы 10](#_Toc196565368)

[4.1. Общее описание 10](#_Toc196565369)

[4.2. Процесс трансляция языка Rust в язык C 11](#_Toc196565370)

[4.2.1. Диаграмма последовательности 12](#_Toc196565371)

[4.2.2. Описание 12](#_Toc196565372)

[5. Заключение 13](#_Toc196565373)

1. Общие положения
   1. Основания для разработки

Учебный план по направлению подготовки «Прикладная математика и информатика» для дисциплины «Проектирование трансляторов».

* 1. Задание на курсовой проект

Разработка программы, реализующая трансляцию языка Rust в язык C.

* 1. Назначение программы

Программа предназначена для того, чтобы транслировать код Rust в структуру на языке C, а затем с ее помощью напечатать то, что бы напечатал код на Rust, будь он скомпилирован и запущен.

1. Основные требования к программе
   1. Функциональные и нефункциональные требования
      1. Функциональные требования
         1. Вывод в стандартный поток вывода результата выполнение кода на Rust, предварительно транслировав его в код на C
         2. Ввод пути, указывающего на входной файл
      2. Нефункциональные требования
         1. Модифицируемость

Изменения определенного модуля программы, не должны затрагивать соседние модули. Время на внесение изменений в отдельный блок не должно превышать одного рабочего дня.

* 1. Среда разработки

Программу решено было разрабатывать на C в связке с Bison+Flex. Разработка программы происходит под операционную систему Linux.

1. Описание архитектуры

Лексический анализатор был реализован с помощью Flex. Синтаксический анализатор был реализован с помощью Bison. Результат синтаксического анализа записывается в структуру AST (см. далее), которая затем последовательно анализируется в программе на языке C.

Процесс трансляции отображен на рисунке 1.

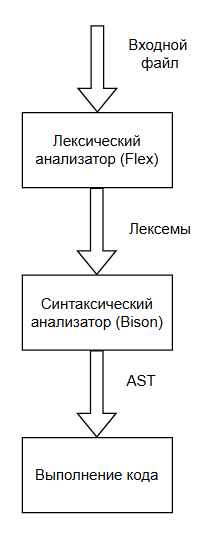


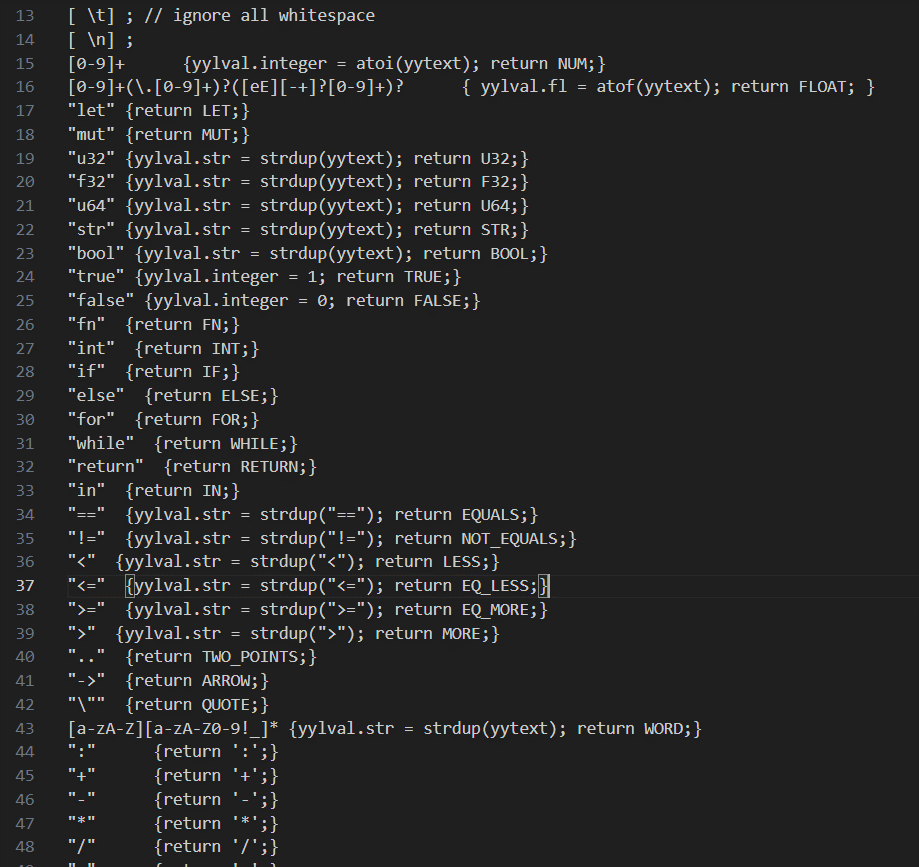
Рисунок 1 – процесс трансляции

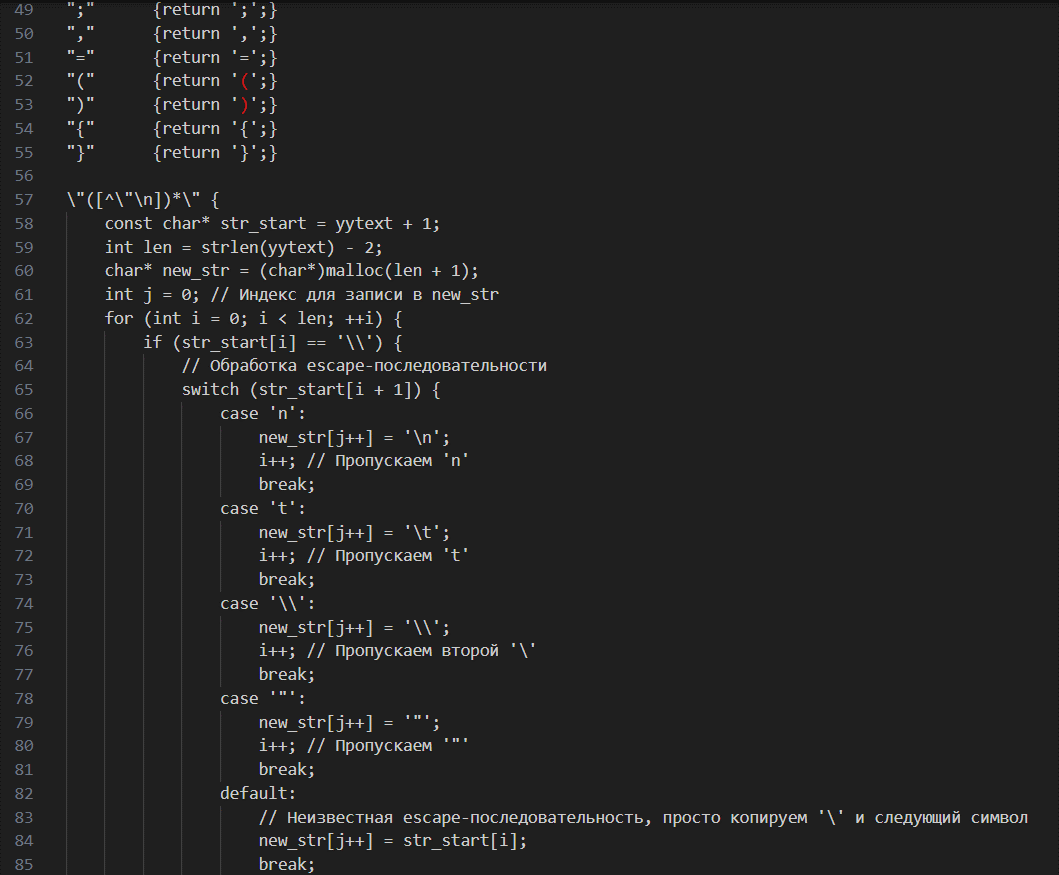
* 1. Описание компонентов системы

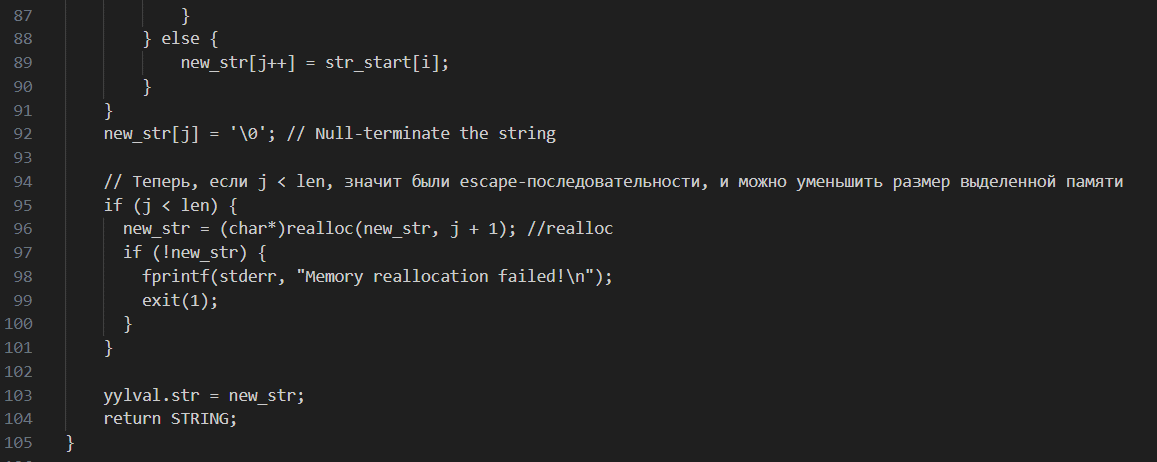
Транслятор представлен в виде программы на C, вызывающую внутри себя функцию yyparce(), которая запускает разбор с использование Bison+Flex, анализирующий полученную структуру AST и последовательно, разбирая ее на отдельные команды, выполняя код.

* + 1. Лексический анализатор

Лексический анализатор реализован в файле lex.l. Список лексем и правила преобразования для flex изображены на рисунке 2-4.







Рисунки 2-4 – правила для Flex

* + 1. Синтаксический анализатор

Синтаксический анализатор реализован в файле sent.y. Грамматика отображена ниже.

<input> ::= %empty

| <input> <command>

<command> ::= <declaration>

| <initialization>

| <functionCall> ';'

| <function>

| <if\_expression>

| <while\_expression>

| <for\_expression>

<declaration> ::= LET MUT WORD ':' <type> <init> ';'

<init> ::= <assignment>

| %empty

<initialization> ::= WORD <assignment> ';'

<assignment> ::= '=' <expr>

<type> ::= U32

| U64

| F32

| STR

| BOOL

<functionCall> ::= WORD '(' <value> ')'

<value> ::= %empty

| <expr>

| <expr> ',' <value>

<parametrs> ::= <par>

| <par> ',' <parametrs>

<par> ::= WORD ':' <type>

| %empty

<val> ::= STRING

| NUM

| FLOAT

| WORD

| TRUE

| FALSE

| <functionCall>

<function> ::= FN WORD '(' <parametrs> ')' <returnArrow> '{' <functionImplementation> '}'

<returnArrow> ::= ARROW <type>

| %empty

<functionImplementation> ::= <funcImpl>

| <funcImpl> <functionImplementation>

<returnVal> ::= <expr>

| %empty

<funcImpl> ::= <functionCall> ';'

| <initialization>

| <declaration>

| <if\_expression>

| <while\_expression>

| <for\_expression>

| RETURN <returnVal> ';'

<brackets\_functionImplementation> ::= '{' <functionImplementation> '}'

<if\_expression> ::= IF <expression> <brackets\_functionImplementation> <else\_expression>

<else\_expression> ::= ELSE <brackets\_functionImplementation>

| %empty

<expression> ::= <val> <cmp> <val>

<cmp> ::= EQUALS

| NOT\_EQUALS

| LESS

| MORE

| EQ\_LESS

| EQ\_MORE

<while\_expression> ::= WHILE <expression> <brackets\_functionImplementation>

<for\_expression> ::= FOR WORD IN NUM TWO\_POINTS NUM <brackets\_functionImplementation>

<expr> ::= <second> '+' <expr>

| <second> '-' <expr>

| <second>

<second> ::= <val> '\*' <second>

| <val> '/' <second>

| <val>

* + 1. Семантический анализатор

Семантический анализатор (вывод результата кода на экран) реализован в файле main.c с использование файлов astHelper.c и ast.c

* 1. Структурная диаграмма классов

На рисунке 5 представлена структурная диаграмма классов для программы на Rust. Корневой структурой является ast. Она же хранит все остальные

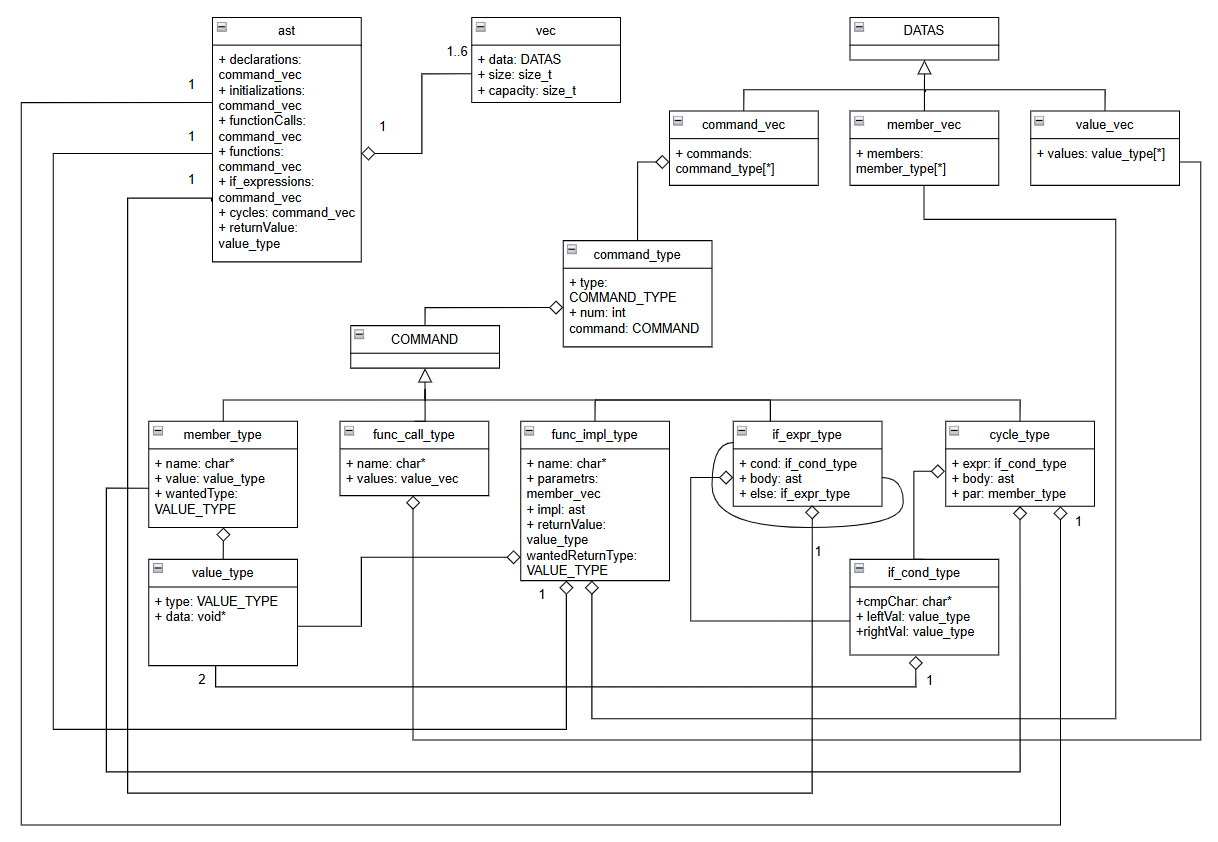


Рисунок 5 – структурная диаграмма классов

1. Алгоритм работы программы
   1. Общее описание

Программа принимает путь, указывающий на исходный файл программы на языке Rust. Далее происходит вызов yyparce(), в случае успешного завершения которого заполняется глобальная структура ast. Затем, вызывается функция doAllAst(struct ast\* ast), которая предварительно проверяет, нет ли вызова функции main (если есть, то это ошибка), и добавляет его самостоятельно для дальнейшего вызова. Далее, вызывается функция doAst(struct ast\* ast). В ее начале провяется корректность ast – вызывается функция checkAst(struct ast\* ast) – идет проверка на то, что для всех переменных объявленный тип совпадает с типом данных, которым ее хотят проинициализовать (не берется в расчет инициализация функциями и другими переменными, обработка такой ошибки происходит после подсчета функции или значения переменной), что не происходит двойного определения переменной и что если есть инициализация переменной (например, a = 5;), то ранее она была определена. После чего все ast сортируется по командам в том порядке, в котором они идут по файлу (функция sortAst(struct ast\* ast)). Затем, идет проходка по получившемуся вектору, и в зависимости от типа команды, выполняется одно из действий:

* MEMBER\_COM\_TYPE (объявление переменной) или FUNC\_IMPL\_TYPE (объявление функции): никаких действий не требуется, продолжаем проходку
* IF\_COM\_TYPE (условный оператор if) – функция doIf(…): подсчитываются левые и правые операнды (подсчет операнда осуществляется в зависимости от типа переменной, если переменная, то через функцию getDataFromObject(…), внутри которого проиходит поиск соответсвующей переменной и получение значения из нее, если вызов функции, то через метод getDataFromFunction(…), внутри которой происходит выполнение функции и получение результата (значения переменной, передаваемой в return), если все остальное, то берется значение из void\* data. Так же реализован случай, когда значение операнда нужно посчитать, например, f(1)+2, или a\*3. Если операции произовдятся просто над числами, то подсчет происходит на этапе синтаксического разбора, если как в примерах, где для подсчета нужно знать какую-то информацию об AST, то происходит следующее: переменная записывается как OBJECT\_TYPE, соответственно, при ее подсчете вызывается метод getDataFromObject(…). Внутри метода проиходит проверка на наличие арифметических знаков, если они есть, то выражение переводится в обратную польскую нотацию, затем уже происходит ее стандарнтый обход с вычислением каждого операнда), сравниваемые в условии оператора (функция checkExpression(…)). Если результат сравнения положительный, то выполняем код в операторе (все внутренние части кода так же хранятся в виде структур AST, поэтому для их выполнения нужно в зависимости от типа определенным образом объединить текущее и следующее AST, затем вызвать функцию doAst(…), и туда передать результат слияния) – происходит слияние с передачей всех переменных из корневого AST. Если результат отрицательный, то проверяем, есть ли else: если есть, ты выполняем его таким же образом, как if, а если нет, то завершаем выполнение текущей команды
* FUNC\_CALL\_TYPE (вызов функции) – функция doFunc(…): идет проверка, не вызываем ли мы println!(…), если это так, то подсчитываем передаваемые туда значения переменных (если таковые имеются), объединяем строки в одну и выводим ее в стандартный поток вывода. Если это не так, то происходит поиск определения соответствующей функции. Параметры функции заменяются параметрами, передаваемыми в вызов функции, происходит слияние AST без передачи переменных из корневого AST, затем происходит выполнение получившегося AST
* CYCLE\_COM\_TYPE (циклы while и for) – функция doCycle(…): вначале идет проверка на непустоту параметра cycle\_type.par – если оно непусто, то там хранится переменная в качестве member\_type, которую надо добавить перед циклом (это происходит в случае, когда цикл for (также преобразуется условие итерации по циклу к аналогичному для цикла while – таким образом, на этом этапе разница между этими двумя циклами пропадает), когда while, то переменная уже есть, либо ее нет, но это ошибка и мы поймем это, когда попробуем получить ее значение). Далее идет проверка выполнения условия для захода в цикл. Если выполняется, то происходит выполнение тела цикла (это происходит также, как выполнение тела оператора if), затем совершается необходимое преобразование над переменной. Если условие не выполняется, то заканчиваем выполнение
* INIT\_COM\_TYPE (инициализация переменной) – doInit(…): происходит проход по всем переменным, находится нужную (или сообщается об ошибке, если не найдено), затем, использую механизм, описанный выше, считается новое значение переменной, старое затирается и на его место записывается посчитанное
* RETURN\_TYPE (возврат из функции): для реализации механизма return было решено завести две глобальные переменные – одна булевая, сообщающая, находится ли программа на этапе выполнения функции. Если это не так, то сообщается об ошибке. Вторая хранит в себе NULL, пока ей не будет присвоено возвращаемое значение. Во время прохода по всем командам в AST идет проверка на непустоту второй глобальной переменной – если туда что-то записалось, то следует закончить выполнение текущего AST. Затем, при выполнении функции getDataFromObject, текущее значение глобальной переменной записывается в результат переменной и обнуляется для корректности дальнейшего выполнения программы
  1. Процесс трансляция языка Rust в язык C

Содержимое примера файла отображено ниже

fn factorial(n: u32) -> u32 {

if n == 0 {

return 1;

} else {

return n \* factorial(n - 1);

}

}

fn fibonacci(n: u32) -> u32 {

if n <= 1 {

return 1;

} else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

fn main() {

let mut number: u32 = 42;

println!("The number is: {}", number);

let mut factorial\_result: u32 = factorial(5);

println!("Factorial of 5 is: {}", factorial\_result);

let mut pi: f32 = 3.14159;

println!("Pi is approximately: {}", pi);

let mut radius: f32 = 5.0;

let mut area: f32 = pi \* radius \* radius;

println!("Area of a circle with radius {} is: {}", radius, area);

let mut is\_true: bool = true;

let mut is\_false: bool = false;

println!("Is true? {}", is\_true);

println!("Is false? {}", is\_false);

for i in 0..4 {

let mut b : u32 = i+1;

println!("The {}th Fibonacci number is: {}", b, fibonacci(b));

}

let mut a : u32 = 0;

while a < 10 {

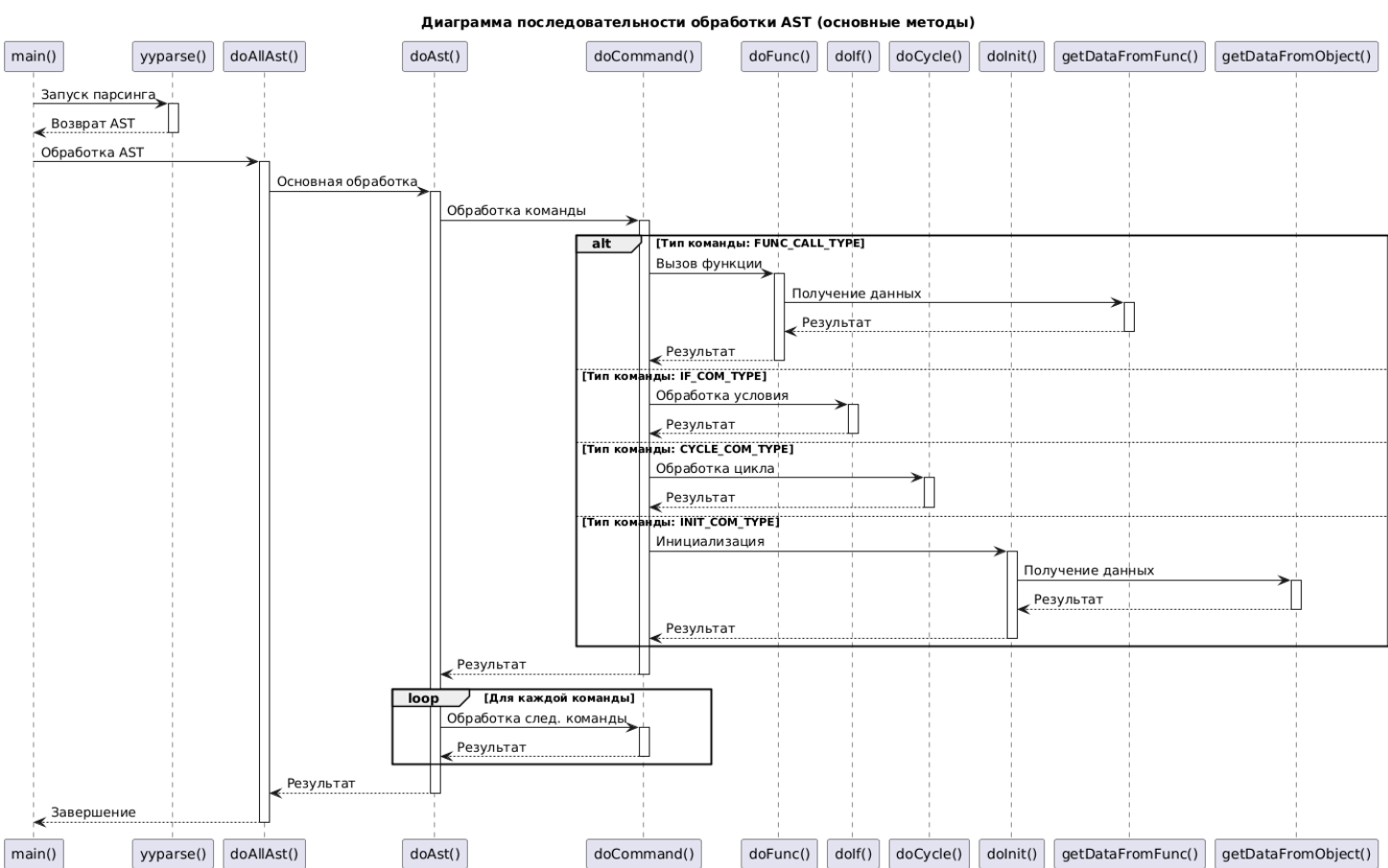
println!("{a} ");

a = a + 2;

}

}

* + 1. Диаграмма последовательности



* + 1. Описание

Для более ясного представления процесса, было решено выделить на диаграмме последовательности только ту часть кода, в которой непосредственно отображена работа архитектуры программы.

Все начинается в main(). Она берет файл из второго аргумента командной строки и отдает его на парсинг с помощью yyparce(). Этот метод вызывает парсинг с помощью Bison+Flex. По итогу, заполняется глобальная переменная ast. Затем вызывается функция doAllAst(ast). Внутри проводятся преобразования (например, добавляется вызов функции main()), после чего вызывается метод doAst, в котором уже непосредственно идет обработка стуктуры. Для этого формируется вектор sortedAst, отсортированный по номеру строки, в которой встретилась команда. Далее идет проход по этому вектору, для каждой команды выполняется метод doCommand(), которая, в зависимости от типа команды, выполняет либо doFunc(), либо doCycle(), либо doIf(), либо doInit(). Эти функции либо каким-то образом меняют переменные, записанные в ast, либо, в случае doFunc(), если у функции имя “println!”, печатают то, что ей передается в качестве аргумента (либо строка, либо значение переменной, либо конкатенация и того и другого)

1. Заключение

В рамках курсового проекта разработан транслятор, позволяющий транслировать код Rust в структуру на языке C, а затем с ее помощью напечатать то, что бы вывел в стандартный поток код на Rust, будь он скомпилирован и запущен. В процессе выполнения задания приобретены практические навыки курса «проектирование трансляторов».