|  |  |
| --- | --- |
| *voenmeh* | МИНОБРНАУКИ РОССИИ  федеральное государственное бюджетное образовательное учреждение  высшего образования  **«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»**  **(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)** |
| БГТУ.СМК-Ф-4.2-К5-01 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Факультет |  | И |  | Информационные и управляющие системы |
|  |  | шифр |  | наименование |
| Кафедра |  | И5 |  | Информационные системы и программная инженерия |
|  |  | шифр |  | наименование |
| Дисциплина |  | Программирование на языке высокого уровня | | |

КУРСОВАЯ РАБОТА

на тему

|  |
| --- |
| Создание компилятора подгруппы команд языка Rust на |
| ASMi386. |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Выполнил студент группы | | | |  | И582 |
| Дубровский В.И. | | | | | |
| Фамилия И.О. | | | | | |
| **РУКОВОДИТЕЛЬ** | | | | | |
| Иванов К.С. | |  |  | | |
| Фамилия И.О. Подпись | | | | | |
| Оценка |  | | | |  |
| «\_\_\_\_\_» |  | | | | 2019г. |

САНКТ-ПЕТЕРБУРГ

2020

**Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»  
(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)**

|  |  |  |  |
| --- | --- | --- | --- |
| Факультет | И |  | Информационные и управляющие системы |
|  | шифр |  | наименование |
| Кафедра | И5 |  | Информационные системы и программная инженерия |
|  | шифр |  | наименование |
| Дисциплина | Системное ПО | | |

**ЗАДАНИЕ**

**НА КУРСОВУЮ РАБОТУ**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студенту | | Дубровскому Владиславу Игоревичу | | | | | | |
|  | | (фамилия, имя, отчество) | | | | | | |
| Научный руководитель | | |  | | | | | |
|  | | | (ученая степень, звание, Ф.И.О) | | | | | |
| Тема: | Создание компилятора для подмножества языка Rust на ASMi386 | | | | | | | |
|  | | | | | | | | |
| Основный вопросы, подлежащие разработке: | | | |  | | | | |
| 1. Лексический анализ, алфавит, символы | | | | | | | | |
| 1. Синтаксический анализ, грамматика, дерево разбора | | | | | | | | |
| 1. Семантический анализ, проверка типов | | | | | | | | |
| 1. Генерация кода, создание программ на языке ассемблера | | | | | | | | |
| Задание выдал: Иванов Константин Сергеевич | | | | |  |  |  |  | |
|  | | | | |  | дата |  | подпись | |
| Задание принял: Иванов Константин Сергеевич | | | | |  |  |  |  | |
|  | | | | |  | дата |  | подпись | |

Санкт-Петербург  
2020

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc45186141)

[**1 Описания входного и выходного языков** 5](#_Toc45186142)

[**1.1 Описание входного языка** 5](#_Toc45186143)

[**1.2 Алфавит и граматика** 5](#_Toc45186144)

[**1.2 Описание выходного языка** 7](#_Toc45186145)

[**2 Архитектура компилятора и основных его частей** 9](#_Toc45186146)

[**2.1Архитектура компилятора** 9](#_Toc45186147)

[**2.2Архитектура «Лексического анализатора»** 9](#_Toc45186148)

[**2.3 Архитектура «Синтаксического анализатора»** 12](#_Toc45186149)

[**3 Тестовые примеры.** 14](#_Toc45186150)

[ЗАКЛЮЧЕНИЕ 15](#_Toc45186151)

[СПИСОК ИСТОЧНИКОВ 16](#_Toc45186152)

[ПРИЛОЖЕНИЕ 17](#_Toc45186153)

# ВВЕДЕНИЕ

Целью данной курсовой работы стала разработка компилятора для языка высокого уровня – Rust. Компилятор должен поддерживать следующие подмножества:

1. Математические выражения
2. Логические выражения
3. Оператор(ы) цикла
4. Оператор(ы) ветвления
5. Функции и/или процедуры с корректной реализацией механизмов вызова, передачи и возврата параметров.
6. Функции ввода вывода или возможность использовать стандартные библиотеки.

# **1 Описания входного и выходного языков**

## **1.1 Описание входного языка**

Перед тем как описывать формальную грамматику языка Rust обратимся к его спецификации. Изначально Rust принимает на вход UTF-8 текст. Но так как в поставленная задача не обязует выполнять этот пункт, на вход данной подгруппы команд языка Rust будет приниматься ASCII-текст.

В дальнейшем мы так же будем ссылаться на спецификацию языка Rust, но уже без явного указания на это, опуская части его правил, ненужных для выполнения задачи.

## **1.2 Алфавит и граматика**

Перейдём же к описанию формальной грамматики языка. Терминальные алфавит предлагаю разбить на подмножества для наглядности, а для краткости записи воспользуемся синтаксисом регулярных выражений:

* ALPHA: [a-zA-Z]
* PUNCTUATION: [+-\*/%^!&|><=\_.#$?,:;’\]
* DELIMITERS: [\{\}\[\]\(\)]
* WHITESPACE: [\t\n\r ]
* HEX\_DIGIT: [0-9a-fA-F]
* NON\_ZERO\_DEC\_DIGIT: [1-9]
* DEC\_DIGIT: [0-9]
* OCT\_DIGIT: [0-7]
* BIN\_DIGIT: [0-1]
* Нетерминальный алфавит:
* IDENTIFIER\_OR\_KEYWORD:([a-zA-Z][a-zA-Z0-9\_]\*)|(\_[a-zA-Z0-9\_]+)
* LINE\_COMMENT: //([^\n]\*)(|$)
* BLOCK\_COMMENT: "(/\\\*.\*\\\*/)
* CHAR\_LITERAL: ' ( ~[' \ \n \r \t] | QUOTE\_ESCAPE | ASCII\_ESCAPE) '
* QUOTE\_ESCAPE: \' | \"
* ASCII\_ESCAPE: \x OCT\_DIGIT HEX\_DIGIT | \n | \r | \t | \\ | \0
* STRING\_LITERAL:"(^["\IsolatedCR]|QUOTE\_ESCAPE|ASCII\_ESCAPE|STRING\_CONTINUE )\* "
* STRING\_CONTINUE: \ followed by \n
* IsolatedCR: \r^\n
* INTEGER\_LITERAL: ( DEC\_LITERAL | BIN\_LITERAL | OCT\_LITERAL | HEX\_LITERAL ) INTEGER\_SUFFIX?
* DEC\_LITERAL: DEC\_DIGIT (DEC\_DIGIT|\_)\*
* BIN\_LITERAL: 0b (BIN\_DIGIT|\_)\* BIN\_DIGIT (BIN\_DIGIT|\_)\*
* OCT\_LITERAL: 0o (OCT\_DIGIT|\_)\* OCT\_DIGIT (OCT\_DIGIT|\_)\*
* HEX\_LITERAL: 0x (HEX\_DIGIT|\_)\* HEX\_DIGIT (HEX\_DIGIT|\_)\*
* INTEGER\_SUFFIX: u8 | u16 | u32 | u64 | u128 | usize | i8 | i16 | i32 | i64 | i128 | isize
* FLOAT\_LITERAL: DEC\_LITERAL . (not immediately followed by ., \_ or an identifier) | DEC\_LITERAL FLOAT\_EXPONENT | DEC\_LITERAL . DEC\_LITERAL FLOAT\_EXPONENT?| DEC\_LITERAL (. DEC\_LITERAL)? FLOAT\_EXPONENT? FLOAT\_SUFFIX
* FLOAT\_EXPONENT:(e|E)(+|-)?(DEC\_DIGIT|\_)\*DEC\_DIGIT (DEC\_DIGIT|\_)\*
* FLOAT\_SUFFIX: f32 | f64
* PUNCTUATION\_EXTENDED:(||)|(&&)|(>>)|(<<)|(+=)|(-=)|(\*=)|(/=)|(%=)|(^=)|(&=)|(|=)|(<<=)|(>>=)|(==)|(>=)|(<=)|(::)
* BOOLEAN\_LITERAL: true | false
* KEYWORDS:(if)|(else)|(while)|(for)|(match)|(break)|(continue)|(fn)|(mut)|(let)|(ref)|(return)|(const)||(in)

Rust, как заявляют создатели языка - это прежде всего язык выражений (expression language). Это означает, что большинство форм языка представляющих ценность или влияющих на результат итоговой программы, являются выражениями (expression) с унифицированным синтаксисом. Каждый вид выражений обычно может быть вложен в любой другой тип выражения, а правила оценки выражений включают в себя указание как значения, создаваемого выражением, так и порядка, в котором его подвыражения оцениваются.

Напротив же, операторы в Rust служат главным образом для того, чтобы содержать в себе выражения и при этом явно определять их последовательность.

# **1.2 Описание выходного языка**

В качестве выходного(конечного) языка в данной работе будет использоваться ассемблер. Т.к разработка проекта велась под управлением ОС Windows, будет использоваться синтаксис ассемблера называемый MASM.

При этом язык ассемблера имеет некоторые особенности. Одной из основных особенностей можно назвать ряд заранее определённые идентификаторы. Такие идентификаторы выделяются специально для обозначение регистров процессора, к которым необходимо обращаться в процессе работы с языком.

В связи с этим такие идентификаторы не требуют предварительного описание, однако при этом они не могут быть использованы разработчиком для иных целей. Так же важно отметить, что в каждом ассемблере свой набор определённых идентификаторов.

В качестве операндов на языке ассемблера порой допускается использование определённых ограниченных сочетаний обозначений и регистров, идентификаторов и констант, которые объединены некоторыми знаками операций. Чаще всего они используются для обозначения типов адресации, допустимых в машинных командах целевой вычислительной системы.

Распределение памяти в компиляторах с языка ассемблера предельно упрощено. Они работают только со статической памятью. Если же необходимо использовать работу с динамической памятью, то в таком случае подключают соответствующие библиотеки или функции ОС, а распределение памяти является задачей разработчика исходной программы.

Так же ранее при описании определённых идентификаторов мною упоминались регистры.

Регистр - это сверхбыстрая память, которая расположена в процессоре. Регистры общего назначения служат для хранения промежуточных данных. В основном их размер 32 байта.

1. eax - применяется для хранения промежуточных данных, то есть он используется в качестве универсального аккумулятора значений.
2. ebx - базовый регистр, который используется для хранения адреса на некоторый блок в памяти.
3. ecx - счетчик, который применяется в основном для циклов.
4. edx - регистр данных, используется для хранения промежуточных вычислений.
5. esp - указатель стека. Содержит адрес вершины стека.
6. esi - индекс источника, в цепочечных операциях содержит указатель на текущий элемент-источник.
7. edi - индекс приёмника, в цепочечных операциях содержит указатель на текущий элемент-приёмник.

# **2 Архитектура компилятора и основных его частей**

## **2.1Архитектура компилятора**

Прежде чем говорить об архитектуре частей компилятора стоит сначала обсудить, что из себя представляет компилятор, и из каких частей он состоит.

Компилятор – это программа, суть работы которой заключается в переводе исходной программы на входном языке в аналогичную объектную программу на выходном языке, ассемблере или языке машинных команд.

В основные функции компилятора входят:

1) Проверка исходной последовательности символов на принадлежность к входному языку;

2) Генерация выходного файла, написанного на ассемблере или языке машинных команд.

Сам процесс компиляции состоит из двух основных этапов: анализа и синтеза. На этапе анализа распознаётся текст исходной программы и заполняется таблица идентификаторов. В результате данного этапа мы получаем некоторое внутреннее представление программы, с которым по итогу компилятору удобно работать.

На этапе синтеза компилятор, используя внутреннее представление программы и информацию, которая содержится в таблице идентификаторов, создаёт выходной текст программы. В результате данного этапа получается сформированный объектный код.

Эти этапы состоят из более мелких стадий, называемых фазами. Состав фаз и их взаимодействие зависит от реализации, выбранной разработчиком.

## **2.2Архитектура «Лексического анализатора»**

Лексический анализ – это процесс разбиения исходной последовательности символов на лексемы – “токены”. В нашем компиляторе лексический анализ многоуровневый.

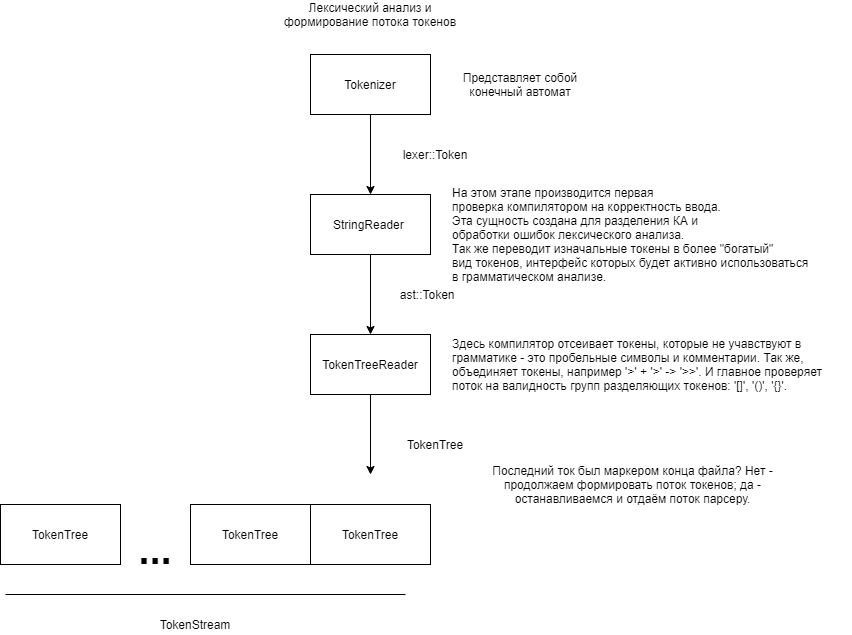


Рисунок 1 – Схема этапов лексического анализа

Первый этап – это выделение с помощью конечного автомата первичных токенов. За это отвечает класс Tokenizer. Он наследуется от класса-обёртки над std::string\_view - Cursor. Cursor реализует методы для просмотра и поглощения символов. Он так же хранит в себе индекс на текущий символ, что позволяет узнавать, сколько символов было поглощено, сколько осталось поглотить, наступил ли конец исходной строки.

Tokenizer получает из вне ссылку на строку и с помощью метода advanceToken(), реализующего конечный автомат, возвращает следующий токен.

Структура токена же такова:

struct Token

{

TokenKind kind = TokenKind::Unknown;

std::size\_t consumed = 0;

TokenData data;

};

TokenKind – это тип токена; consumed – сколько символов было поглощено с момента последнего вызова advanceToken(); TokenData – это std::variant(union из STL), который содержит в себе дополнительную информацию для определённых токенов, например вид литерала.

За второй этап отвечает класс StringReader. Он хранит в себе Tokenizer и указатель на вспомогательную сущность ParseSession, хранящую в себе информацию о текущем файле, его source map и предоставляющую интерфейс для формирования ошибок с восстановлением исходного кода.

StringReader принимает и переводит с помощью метода nextToken() токен из Tokenizer’а в более богатые токены для AST, проверяет корректность некоторых токенов, подготавливая ошибки и предупреждения в случаи неверных входных данных, и формирует таблицу внутренних символов компилятора.

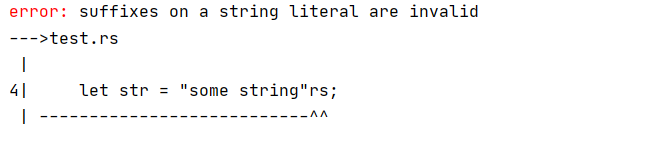


Рисунок 2 - Пример указания ошибки StringReader'а

Таблица внутренних символов исходит из того, что Tokenizer не выделяет ключевые слова компилятора, как отдельный вид токенов, а поглощает их, как идентификаторы. StringReader же встречая идентификатор, обращается к внутренней таблице символов, которая либо создаёт, либо возвращает уже имеющийся символ в ней. for,while,if и прочие зарезервированные компилятором единицы, инициализируют таблицу, обеспечивая постоянный индекс, вне зависимости от рантайма.

Третий и последний этап – это TreeReader. Эта сущность хранит в себе StringReader и указатель на ParseSession для вывода ошибок. Его задача сформировать деревья разделителей и токенов и проверить, что у каждого разделителя есть пара. На рисунке 3 представлен вывод ошибки, вызванной тем, что у закрывающей фигурной скобки нет открытой пары.

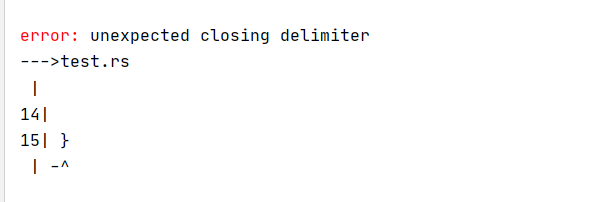


Рисунок 3- Пример ошибки TreeReader'a

Если это возможно, то TreeReader склеивает токены в воедино, например из двух ‘=’ делает ‘==’. И так же, отсекает Whitespaces и комментарии. Результатом работы TreeReader’а является поток токенов – TokenStream, используемый парсером в синтаксическом анализе.

## **2.3 Архитектура «Синтаксического анализатора»**

Синтаксический анализ – это процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой.

В ходе работы синтаксического анализатора поток токенов преобразуется в абстрактное синтаксическое дерево. Наш парсер является нисходящим. Нисходящий парсер (англ. top-down parser) — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов. Строим дерево с помощью метода рекурсивного спуска.

Задача синтаксического анализа состоит в том, чтобы определить имеет ли цепочка лексем конструкцию, заданную синтаксисом языка. Этот этап является основным для поиска ошибок в написанной программе.

Спецификация Rust вводит следующие основные элементы языка: Statements, Expressions, Items, Patterns, Types. От куда в нашем решении созданы следующие структуры.

Построение дерева начинается с разбора Statements и в зависимости от последующих токенов разбор будет сделан выбор между Statements-Expression, Statements-Item, Statements-Semi или Statements-Empty.

# **3 Тестовые примеры.**

Приведём пример AST следующей программы:

**const *SOME\_CONST*** : **i32** = 23;  
  
**fn** print(sus: **i32**)  
{  
 sus+5  
}  
  
**fn** main()  
{  
 **let** a = 32;  
 **let** s = **if** a + ***SOME\_CONST*** > 60 {  
 a  
 }  
 **else** {  
 ***SOME\_CONST*** };  
  
 print(s);  
}

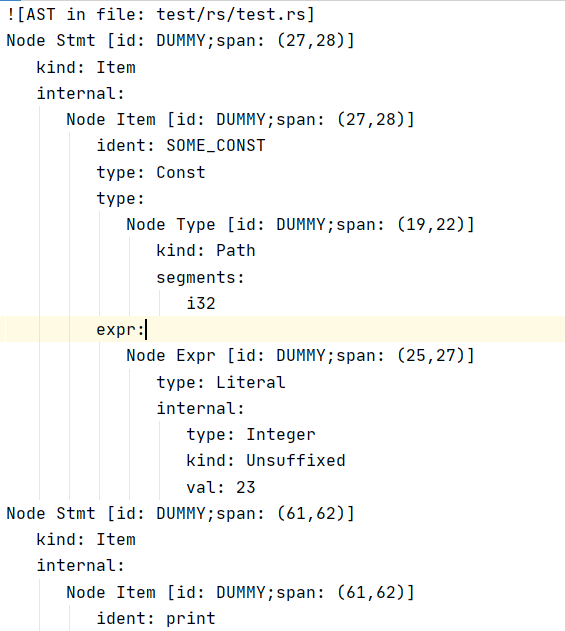


Рисунок 4 - Пример AST

# ЗАКЛЮЧЕНИЕ

В процессе выполнения данной курсовой работы были получены знания и навыки связанные с поэтапной разработкой компилятора. Был сформирован конечный автомат для распознавания и формирования лексем текста. Разработан лексический анализатор текста программы на входном языке. Реализован синтаксический анализатор методом рекурсивного спуска.

Программа была разработана на языке программирования C++ в среде разработки CLion, использовался компилятор Microsoft Visual C++ Compiler 16.29215.179 (x86). Реализованы фазы компилятора для языка высокого уровня – Rust на язык низкого уровня – ассемблер: лексический анализатор, синтаксический анализатор. Программа была протестирована на различных примерах.

# СПИСОК ИСТОЧНИКОВ

1. Ахо А., Лам М., Сети Р., Ульман Д. Компиляторы принципы, технологии и инструментарий Второе издание . - М.:ООО “И. Д. Вильямс”, 2008. - 1184 с.
2. https://doc.rust-lang.org/reference/introduction.html

# ПРИЛОЖЕНИЕ

<https://github.com/alohaeee/Corrosion>