**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук

Образовательная программа «Программная инженерия»

|  |  |  |  |
| --- | --- | --- | --- |
| СОГЛАСОВАНО  Научный руководитель, профессор департамента «Программной инженерии», доктор технических наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.И. Легалов  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г. | |  | УТВЕРЖДАЮ  Академический руководитель образовательной программы «Программная инженерия»  профессор департамента программной инженерии, канд. техн. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н. А. Павлочев  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г. |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл*** |  | | **КРОСС-КОМПИЛЯТОРС ЯЗЫКА ПРОГРАММИРОВАНИЯ SMALL C**  **В АССЕМБЛЕР ПРОЦЕССОРА RISC-V, ДЛЯ ЭМУЛЯТОРА RARS**  **Пояснительная записка**  **ЛИСТ УТВЕРЖДЕНИЯ**  **RU.17701729.04.01-01 81 01-1-ЛУ** | | | | | |
|  | | Исполнитель  студент группы БПИ229  \_\_\_\_\_\_\_\_\_ / Р. С. Громов/  «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2024 г. | | | |
|  | | | |

**Москва 2024**

УТВЕРЖДЕН

RU.17701729.10.03-01 81 01-1-ЛУ

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл*** |  | | **КРОСС-КОМПИЛЯТОРС ЯЗЫКА ПРОГРАММИРОВАНИЯ SMALL C**  **В АССЕМБЛЕР ПРОЦЕССОРА RISC-V, ДЛЯ ЭМУЛЯТОРА RARS**  **Пояснительная записка**  **RU.17701729.04.01-01 81 01-1**  **Листов 27** | |
|  |  |
|  |
|  | |

**Москва 2024**

**АННОТАЦИЯ**

В данном программном документе приведена пояснительная записка к программе «Кросс-компилятор с языка программирования Small C в ассемблер процессора RISC-V для эмулятора RARS», предназначенной для трансляции высокоуровнего кода в ассемблерные инструкции.

В разделе «Введение» указано наименование программы, краткое наименование программы и документы, на основании которых ведется разработка.

В разделе «Назначение и область применения» указано функциональное и эксплуатационное назначение программы и краткая характеристика области применения программы.

В разделе «Технические характеристики» содержатся следующие подразделы:

* постановка задачи на разработку программы;
* описание алгоритма и функционирования программы с обоснованием выбора схемы алгоритма решения задачи и возможные взаимодействия программы с другими программами;
* описание и обоснование выбора метода организации входных и выходных данных;
* описание и обоснование выбора состава технических и программных средств.

В разделе «Ожидаемые технико-экономические показатели» указана предполагаемая потребность и экономические преимущества разработки по сравнению с отечественными и зарубежными образцами или аналогами.

Настоящий документ разработан в соответствии с требованиями:

1. ГОСТ 19.101-77 Виды программ и программных документов [1];
2. ГОСТ 19.102-77 Стадии разработки [2];
3. ГОСТ 19.103-77 Обозначения программ и программных документов [3];
4. ГОСТ 19.104-78 Основные надписи [4];
5. ГОСТ 19.105-78 Общие требования к программным документам [5];
6. 6) ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом [6];
7. ГОСТ 19.404-79 Пояснительная записка. Требования к содержанию и оформлению [7].

Изменения к Пояснительной записке оформляются согласно ГОСТ 19.603-78 [8], ГОСТ 19.604-78 [9].

Перед прочтением данного документа рекомендуется ознакомиться с терминологией, приведенной в Приложении 1 настоящей пояснительной записки.

**СОДЕРЖАНИЕ**

[**1.** **ВВЕДЕНИЕ** 5](#_Toc162888760)

[**1.1** **Наименование программы** 5](#_Toc162888761)

[**1.2** **Документы, на основании которых ведётся разработка** 5](#_Toc162888762)

[**2.** **НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ** 6](#_Toc162888763)

[**2.1** **Назначение программы** 6](#_Toc162888764)

[**2.1.1.** **Функциональное назначение** 6](#_Toc162888765)

[**2.1.1** **Эксплуатационное назначение** 6](#_Toc162888766)

[**2.2** **Краткая характеристика области применения** 6](#_Toc162888767)

[**3.** **ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ** 7](#_Toc162888768)

[**3.1** **Постановка задачи на разработку программы** 7](#_Toc162888769)

[**3.2** **Описание алгоритма и функционирования программы** 7](#_Toc162888770)

[**3.3** **Описание функционирования обработчика файлов** 7](#_Toc162888771)

[**3.4** **Описание функционирования препроцессора** 8](#_Toc162888772)

[**3.5** **Описание функционирования лексического анализатора** 9](#_Toc162888773)

[**3.6** **Описание функционирования синтаксического анализатора** 11](#_Toc162888774)

[**3.7** **Описание функционирования генератора ассемблерного кода** 14](#_Toc162888775)

[**4.** **ОЖИДАЕМЫЕ ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ** 18](#_Toc162888776)

[**4.1** **Предполагаемая потребность** 18](#_Toc162888777)

[**4.2** **Экономические преимущества разработки по сравнению с отечественными и зарубежными образцами или аналогами** 18](#_Toc162888778)

[**5.** **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ** 19](#_Toc162888779)

[**ПРИЛОЖЕНИЕ 1** 21](#_Toc162888780)

[**ПРИЛОЖЕНИЕ 2** 23](#_Toc162888781)

1. **ВВЕДЕНИЕ**
   1. **Наименование программы**

Наименование темы разработки: «Кросс-компилятор с языка программирования Small C в ассемблер процессора RISC-V для эмулятора RARS».

Наименование темы разработки на английском языке: «Cross-compiler from the Small C Programming Language into the Assembler of the RISC-V Processor, for the RARS Emulator».

Условное обозначение темы разработки: «RISC-V Cross-compiler».

* 1. **Документы, на основании которых ведётся разработка**

Программа выполнена в рамках задания на курсовую работу в соответствии с учебным планом подготовки бакалавров (НИУ ВШЭ, факультет компьютерных наук) по направлению 09.03.04 «Программная инженерия» и утверждённой академическим руководителем темы курсового проекта. Приказ декана факультета компьютерных наук И. В. Аржанцева "Об утверждении тем, руководителей курсовых работ студентов образовательной программы «Программная инженерия» факультета компьютерных наук"

1. **НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ**
   1. **Назначение программы**
      1. **Функциональное назначение**

Функциональное назначение «RISC-V Cross-compiler» представимо в виде списка основных функциональных групп приложения:

* Выбор исходных файлов программы, написанной на языке Small C, а также их валидация для её дальнейшей компиляции.
* Препроцессинг: получение отформатированной копии программы путём удаления лишних символов и комментариев и передача её лексическому анализатору.
* Компиляция: получение ассемблерного листинга исходной программы в ходе выполнения лексического, синтаксического, семантического анализа и генерации промежуточного кода.
* Получение информации о синтаксических и компиляционных ошибках и предупреждений
* Генерация промежуточного кода в ассемблерные инструкции архитектуры RISC-V.
* Запуск ассемблерного кода на эмуляторе RARS.
  + 1. **Эксплуатационное назначение**

Приложение «RISC-V Cross-compiler» представляет из себя полноценный кроссплатформенный компилятор для трансляции высокоуровнего кода в ассемблерные инструкции.

* 1. **Краткая характеристика области применения**

Это приложение предназначено для студентов программной инженерии, которые хотят улучшить свои знания в области архитектуры вычислительных систем и усовершенствовать свои навыки работы с языком ассемблера процессора RISC-V для успешного выполнения семинарских и практических заданий.

1. **ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ**

## **3.1 Постановка задачи на разработку программы**

Цель проекта – реализовать полноценное консольное кросс-платформенное приложение, способное транслировать высокоуровневый код языка программирования Small-C в инструкции ассемблера архитектуры процессоров RISC\_V.

## **3.2 Описание алгоритма и функционирования программы**

Функционал кросс-компилятора состоит из пяти основных разделов:

1. Обработчик файлов;
2. Препроцессор;
3. Лексический анализатор;
4. Синтаксический анализатор;
5. Генератор ассемблерного кода.

## **3.3 Описание функционирования обработчика файлов**

Обработчик файлов нужен для того, чтобы провалидировать введённое имя файла и сообщить пользователю об ошибках, которые могут возникнуть при его чтении.

Возможные ошибки при введении имени файла:

1. У файла отсутствует расширение.
2. Расширение файла не совпадает с общепринятым расширением для программ, написанных на языке Small-C.
3. Файл отсутствует в директории с исполняемым файлом программы. Указан неверный абсолютный или относительный путь к файлу.

При обнаружении соответствующих ошибок, обработчик файлов направляет в консоль предупреждение о некорректном вводе и просит пользователя повторно ввести полное имя файла, написанного на языке Small-C.

Далее обработчик файлов формирует структуру файла, которая содержит следующую информацию:

1. Имя файла;
2. Расширение;
3. Размер файла в байтах.

Если файл весит большее 4 Кб, обработчик выведет в консоль соответствующее предупреждение о возможной медленной работе компилятора. Если все предыдущие этапы прошли успешно, программа выводит в консоль данную служебную информацию и отправляет файл препроцессору.

## **3.4 Описание функционирования препроцессора**

Задача препроцессора состоит в форматировании необработанного файла для того, чтобы привести его к красивому виду для пользователя и в дальнейшем к удобному формату для лексического анализатора.

При форматировании файла для пользователя, препроцессор выполняет следующие функции:

1. Сокращение количества пробельных символов между словами в тексте до одного знака пробела;
2. Выравнивание строк кода в соответствии с их расположением внутри области видимости, ограниченной фигурными скобками.

При форматировании файла для лексического анализатора, препроцессор использует уже отредактированный для пользователя файл и выполняет следующие функции:

1. Удаление однострочных комментариев;
2. Удаление многострочных комментариев;
3. Удаление всех знаков табуляции;
4. Удаление всех символов переноса на новую строку.

При выполнении второго этапа форматирования файл становится неудобным для чтения пользователем, но зато удобным для работы лексического анализатора. В результате получается одна длинная строка, содержащая весь текст программы.

## **3.5 Описание функционирования лексического анализатора**

После двух предыдущих этапов программа получает на вход обработанную строку обычного текста. Задача лексического анализатора заключается в обработке этой последовательности символов на распознанные группы в рамках текущего высокоуровневого языка для того, чтобы получить на выходе идентифицированные последовательности символов, которые называются токенами.

В языке Small-C можно выделить следующие токены, шаблоны и лексемы, которые описаны в виде следующей таблицы:

**Таблица № 1. Токены, шаблоны и лексемы языка Small-C**

|  |  |  |
| --- | --- | --- |
| **Токен** | **Шаблон (регулярное выражение)** | **Лексемы** |
| IDENTIFIER | [a-zA-Z\_][a-aA-Z\_0-9]\* | main, var12, \_test |
| BINARY\_INT\_LITERAL | 0b[0-1]+ | 0b1011 |
| OCTAL\_INT\_LITERAL | 0o[0-7]+ | 0o0712466 |
| HEX\_INT\_LITERAL | 0[xX][0-9a-fA-F]+ | 0x12Fa, 0XFFA |
| DOUBLE\_LITERAL | [0-9]\*[0-9]\* | 0.123, 12.75, .124 |
| STRING\_LITERAL | "[[:ascii:]]+" | “Lorem ipsum” |
| CHAR\_LITERAL | ‘[[:ascii:]]’ | ‘a’ |
| CHAR | char | char |
| INT | int | int |
| DOUBLE | double | double |
| IF | if | if |
| ELSE | else | else |
| WHILE | while | while |
| DO | do | do |
| FOR | for | for |
| SWITCH | switch | switch |
| CASE | case | case |
| CONTINUE | continue | continue |
| BREAK | break | break |
| Продолжение Таблицы № 1 |  |  |
| RETURN | return | return |
| CONST | const | const |
| VOID | void | void |
| ARITHMETIC\_OPERATOR | \+|\-|\\*|\/|\% | +, -, \*, /, % |
| LOGIC\_OPERATOR | &&|\|\||! | &&, ||, ! |
| COMPARISON\_OPERATOR | \<|\>|\<\=|\>\=|\=\=|\! | <. >, <=, >=, ==, != |
| BITWISE\_OPERATOR | \||\&|\~|\<\<|\>\> | |, &, ~, <<, >> |
| ASSIGN\_OPERATOR | = | = |
| ARITHM\_ASSIGN\_OPERATOR | \+\=|\-\=|\\*\=|\/\=|\%\= | +=, -=, \*=, /=, %= |
| BITWISE\_ASSIGN\_OPERATOR | \|\=|\&\=|\~\=|\<\<\=|\>\>\= | |=, &=, ~=, <<=. >>= |
| ROUND\_BRACKETS | \(|\) | (, ) |
| CURLY\_BRACKETS | \{|\} | {, } |
| SQUARE\_BRACKETS | \[|\] | [, ] |

Во время чтения последовательности символов лексический анализатор сравнивает каждую подпоследовательность символов с шаблоном (регулярным выражением).

Если произошло совпадение с шаблоном, то формируется специальная структура *Token*, которая содержит в себе следующие атрибуты:

* + - 1. Тип токена (представлен в первом столбце таблицы токенов);
      2. Текст лексемы;
      3. Длина лексемы;
      4. Номер строки, в которой была найдена лексема;
      5. Номер символа с начала строки;
      6. Абсолютная позиция начала лексемы.

Тип токена, текст лексемы и её длина нужны программе для того, чтобы синтаксический анализатор работал не с обычными символами, а с идентифицированными лексическими единицами языка.

Номер строки, номер символа с начала строки и абсолютная позиция начала лексемы нужны программе для того, чтобы в случае возникновения ошибки во время лексического или синтаксического анализа, можно было вывести в консоль соответствующее сообщение об ошибке с указанием точной позиции в тексте исходной программы.

Основная функция, которая возвращает последовательность токенов – *struct Vector\* get\_all\_tokens(struct Lexer\* lexer)*. Внутри неё формируется вектор с типом токенов и работает цикл, который распознаёт токен и записывает его в вектор. Если так получилось, что тип токена – END, то цикл завершает свою работу и функция возвращает сформированный вектор токенов.

Также в программе играет большую роль функция *struct Token\* get\_token(char\* content, size\_t position)*, которая по первому символу входного потока определяет предполагаемый тип токена согласно шаблонам, описанным выше, и перенаправляет определение токена в более специализированные функции по идентификации токенов. Например: *determine\_octal\_token(…)*. Если этим функциям не удалось найти никакой конкретный токен во входном потоке, лексический анализатор возвращает INCORRECT и посылает ошибку в консоль.

## **3.6 Описание функционирования синтаксического анализатора**

Задача синтаксического анализатора состоит в проверки синтаксических структур, находящихся в последовательности токенов. Иными словами, парсер проверяет входной сигнал на соответствие правильному синтаксису языка. На этом этапе выполнения программы компилятор проверяет, соответствует ли исходный код программы грамматическим правилам высокоуровнего языка программирования, на котором была написана исходная программа. Грамматические и синтаксические правила описываются с помощью специального метаязыка. В данной работе используется графический метаязык, который называется диаграммами Вирта.

**Таблица № 2. Описание компонентов диаграмм Вирта:**

|  |  |
| --- | --- |
| **Графический компонент** | **Описание компонента** |
|  | Терминальный символ, принадлежащий алфавиту языка |
|  | Постоянная группа терминальных символов, которая определяет название лексемы |
|  | Нетерминальный символ, который определяет название правила |
|  | Соединительные линии, которые обеспечивают связь между терминальными и нетерминальными символами в правилах |
|  | Входная дуга с именем правила, которая определяет его начало |

Основная функция парсера – функция *struct Vector\* start\_parsing(struct Parser\* parser)*, которая дробит поток токенов на подпоследовательности и передаёт каждую подпоследовательность в функцию распределитель.

Функция распределитель по первому токену в подпоследовательности определяет предполагаемый вид синтаксической конструкции и в соответствии с ним вызывает нужные распознаватели.

Функции – распознаватели, на вход получают подпоследовательности токенов, а на выходе выдают булевое значение, обозначающее, является ли эта подпоследовательность синтаксической конструкцией определённого вида.

Для языка программирования Small-C можно выделить следующие основные синтаксические конструкции языка, которые можно описать с помощью диаграмм Вирта и идентифицировать с помощью распознавателей.

**Таблица № 3. Синтаксические конструкции и функции-распознаватели**

|  |  |
| --- | --- |
| **Синтаксическая конструкция** | **Функция – распознаватель в парсере** |
| Объявление одной переменной | is\_single\_declaration\_expression(…) |
| Объявление множества переменных | is\_complex\_declaration\_expression(…) |
| Определение одной переменной | is\_single\_definition \_expression(…) |
| Определение множества переменных | is\_complex \_definition \_expression(…) |
| Арифметическое выражение | is\_arithmetic\_expression(…) |
| Логическое выражение | is\_logic \_expression(…) |
| Сравнительное выражение | is\_relational \_expression(…) |
| Поразрядное выражение | is\_arithmetic\_expression(…) |
| Префиксное выражение | is\_prefix \_expression(…) |
| Постфиксное выражение | is\_postfix \_expression(…) |
| Условный оператор if-else | is\_if\_else\_statement(…) |
| Цикл while | is\_while\_statement(…) |
| Цикл do-while | is\_do\_while \_statement(…) |
| Цикл for | is\_for \_statement(…) |
| Конструкция switch-case | is\_switch\_case\_statement(…) |
| Объявление функции | is\_function\_declaration(…) |

Если функция распознаватель не смогла найти нужную синтаксическую конструкцию, она выводит в консоль сообщение об ошибке. Когда все подпоследовательности будут успешно распознаны, основная функция парсера вернёт последовательность специальных синтаксических структур, которые в дальнейшем будут использоваться генератором ассемблерного кода.

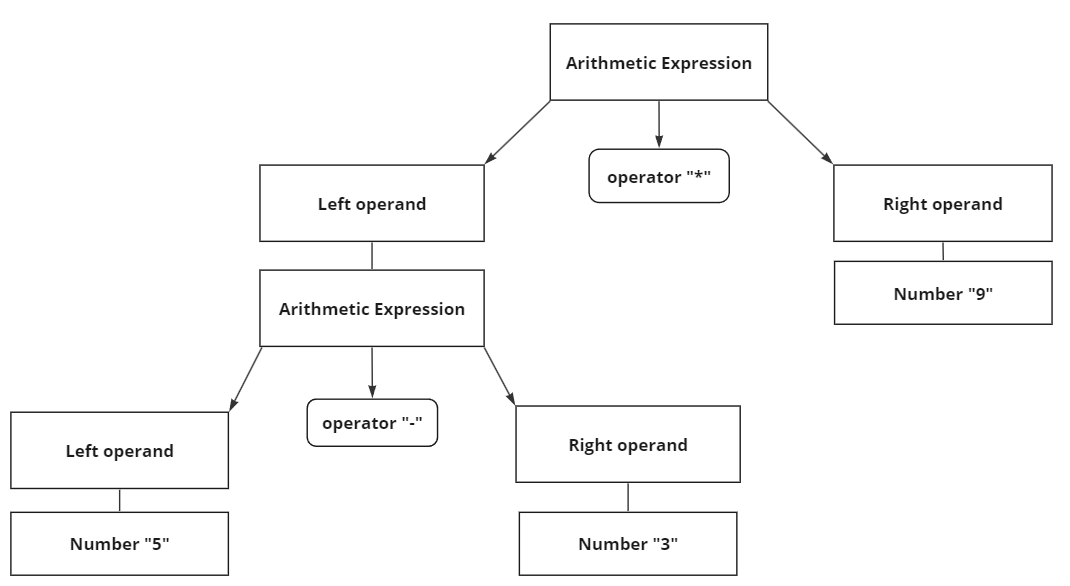
## **3.7 Описание функционирования генератора ассемблерного кода**

Генерация ассемблерного кода является последним этапом компиляции программы. Соответствующий модуль в компиляторе представлен множеством функций, которые на вход принимают специальные синтаксические структуры, являющиеся промежуточным представлением и которые обрабатывают эти структуры, генерирую ассемблерный код под архитектуру процессоров RISC\_V.

**3.7.1 Генерация кода для выражений**

Основные выражения, используемые в языке программирования Small-C: арифметические, побитовые, логические и сравнительные. Их отличие от остальных синтаксических структур заключается в наличии глубокой вложенности компонентов. Иными словами, мы можем представить каждое выражение в виде абстрактного синтаксического дерева.

*Пример: арифметическое выражение (5 - 3) \* 9 имеет следующее промежуточное представление в компиляторе:*



Для того, чтобы сгенерировать ассемблерный код для этого представления, компилятор делает центрированный обход этого дерева, сначала углубляясь в левую ветку, потом применяя оператор, находящийся в текущем узле и далее углубляясь в правую ветку.

При заходе в узел дерева, в котором оба операнда являются числами, он генерирует следующие ассемблерные инструкции:

1. Сохраняет значение левого операнда в регистр t0;
2. Сохраняет значение правого операнда в регистр t1;
3. Применяет соответствующую команду для оператора;
4. Сохраняет Полученное значение на стэк;
5. Очищает временные регистры.

При возникновении ситуации, когда левый операнд представлен в виде выражения, а правый операнд в виде числа, компилятор генерирует следующие инструкции:

1. Удаление вершины стэка и запись в регистр t0. Вершина стэка – результат выполнение операции в левом операнде;
2. Сохранение правого числового операнда в регистр t1;
3. Выполнение соответствующей операции и сохранение результата в стэк;
4. Очистка временных регистров.

При возникновении ситуации, когда правый операнд представлен в виде выражения, компилятор:

1. Выполняет предыдущие действия для генерации ассемблерного кода для правого операнда;
2. Выводит из стэка два значения. Первое значение – результат операции в левом узле дерева или обычное число, а второе – результат операции в правом узле дерева;
3. Применение соответствующей операции для этих двух результатов и сохранение его на стэк;
4. Очистка временных регистров.

В результате всего обхода дерева, мы получим единственное число на стэке, которое будет являться результатом выполнения арифметической операции на языке ассемблера. Код для всех других выражений генерируется аналогичным способом.

**3.7.2 Генерация кода для объявления и определения переменных**

Объявления и определения переменных могут быть представлены в двух видах: В одиночном виде и в множественном виде.

*Пример одиночного объявления и определения переменной:*

***int a;***

***int b = 3 + 1;***

*Пример множественного объявления и определения переменной:*

***char c, v, t;***

***char sym = ‘a’, sym2 = ‘b’, sym\_n = ‘z’;***

Любая переменная, объявленная или определённая внутри функции или какой-нибудь области видимости хранится на стэке. Для того, чтобы не было неоднозначности в хранении данных, для каждой переменной используется специальная метка, которая именуется следующим способом: название переменной + смещение относительно начала стэка. При выходе из функции или области видимости, все переменные, которые находились в ней, удаляются из стэка. Последующие метки переменных именуются с учётом того, что текущий указатель на стэк сместился к началу из-за удаления не используемых переменных.

**3.7.3 Генерация кода для составных операторов**

Составные операторы, такие как if-else, while, do-while, for состоят из описанных выше конструкций: объявлений, определений, логических и условных выражений, и прочих. При генерации кода для таких операторов используется промежуточное представление, содержащее в себе структуры других синтаксических конструкций. Ассемблерный код генерируется по частям.

Пример: Промежуточная структура цикла while состоит из синтаксической структуры логического или условного выражения, а также тела цикла. В котором может быть множество ссылок на любые другие синтаксические структуры языка Small-C. В результате ассемблерный код для составных операторов является совокупностью ассемблерных инструкций каждой части этого оператора. Аналогичная работа генератора используется и при объявлении функций.

**3.7.4 Генерация кода для главной функции main**

Прежде чем генерировать все остальные синтаксические структуры языка, компилятор в начале должен проверить существование функции main, с которой и будет начинаться работать программа. Компилятор для неё сгенерирует глобальную метку *\_main:* Если функция main будет отсутствовать в файле, то будет выведена в консоль ошибка компиляции.

1. **ОЖИДАЕМЫЕ ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ**
   1. **Предполагаемая потребность**

Целевой аудиторией «RISC-V Cross-compiler» являются студенты «Программной инженерии». Их потребность заключается в повышении уровня своих знаний по дисциплине «Архитектура вычислительных систем», а также желании легче справляться с выполнением практических заданий по языку ассемблера процессора RISC-V.

* 1. **Экономические преимущества разработки по сравнению с отечественными и зарубежными образцами или аналогами**

Основное преимущество данного приложения заключается в его бесплатном распространении и открытом исходном коде, которым можно пользоваться в академических целях.

1. **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**
2. ГОСТ 19.101-77 Виды программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
3. ГОСТ 19.102-77 Стадии разработки. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
4. ГОСТ 19.103-77 Обозначения программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
5. ГОСТ 19.104-78 Основные надписи. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
6. ГОСТ 19.105-78 Общие требования к программным документам. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
7. ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
8. ГОСТ 19.201-78 Техническое задание. Требования к содержанию и оформлению. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
9. ГОСТ 19.603-78 Общие правила внесения изменений. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
10. ГОСТ 19.604-78 Правила внесения изменений в программные документы, выполненные печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
11. ГОСТ 15150-69 Машины, приборы и другие технические изделия. Исполнения для различных климатических районов. Категории, условия эксплуатации, хранения и транспортирования в части воздействия климатических факторов внешней среды. – М.: Изд-во стандартов, 1997.
12. ГОСТ 19.602-78 Правила дублирования, учета и хранения программных документов, выполненных печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
13. ГОСТ 19.301-79 Программа и методика испытаний. Требования к содержанию и оформлению. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
14. Small-C Wikipedia.
15. J.G. Harstone. // C compilers for the BBC Computer.
16. Д. Хендрикс. // Компилятор языка Си для микроЭВМ.
17. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. // Компиляторы: принципы, технологии и инструментарий.
18. Свердлов С. // Конструирование компиляторов, учебное пособие.

# **ПРИЛОЖЕНИЕ 1**

**ТЕРМИНОЛОГИЯ**

**Компилятор** — программа, переводящая написанный на языке программирования текст в набор машинных кодов.

**Лексический анализ** (токенизация) – это процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы — с целью получения на выходе идентифицированных последовательностей, называемых «токенами».

**Синтаксический анализ** (парсинг) в лингвистике и информатике - это процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

**Синтаксис** языка программирования — набор правил, описывающий комбинации символов алфавита, считающиеся правильно структурированной программой (документом) или её фрагментом.

**Препроцессор** — компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы.

**Метаязык** — язык, предназначенный для описания другого языка, называемого объектным языком.

**Терминальные** **символы** – это символы, которые могут появляться в выходных данных производственных правил формальной грамматики и которые не могут быть изменены с помощью правил грамматики.

**Нетерминал** (нетерминальный символ) — объект, обозначающий какую-либо сущность языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

**Формальная грамматика** или просто грамматика в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита.

**Регулярные выражения** – это формальный язык, используемый в компьютерных программах, работающих с текстом, для поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов.

**Лексема** – это последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется лексическим анализатором как экземпляр токена.

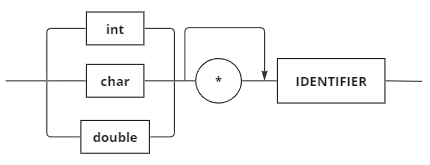
**Шаблон (pattern)** — это описание вида, который может принимать лексема токена. В случае ключевого слова шаблон представляет собой просто последовательность символов, образующих это ключевое слово.

**Токен** – это пара, состоящая из имени токена и необязательного атрибута.

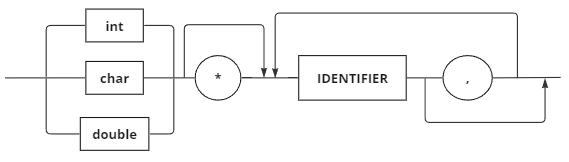
# **ПРИЛОЖЕНИЕ 2**

**ОПИСАНИЕ СИНТАКСИСА ЯЗЫКА SMALL-C**

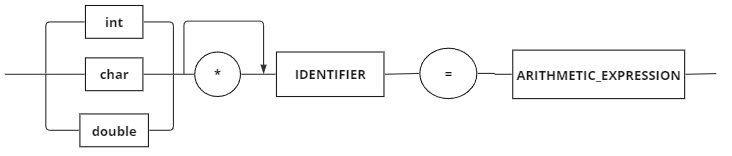
**Одиночное объявление переменной:**



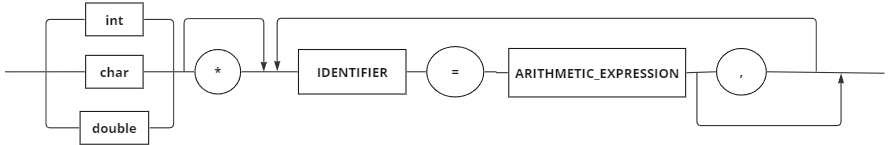
**Множественное объявление переменных:**



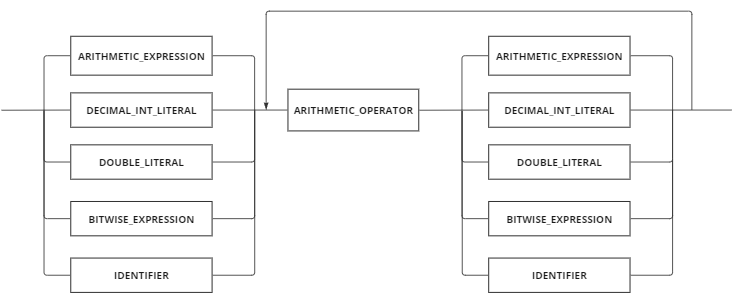
**Одиночное определение переменной:**



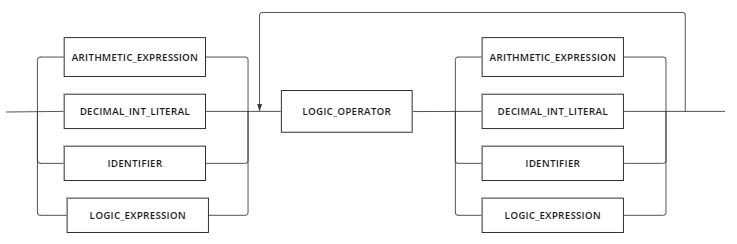
**Множественное определение переменных:**



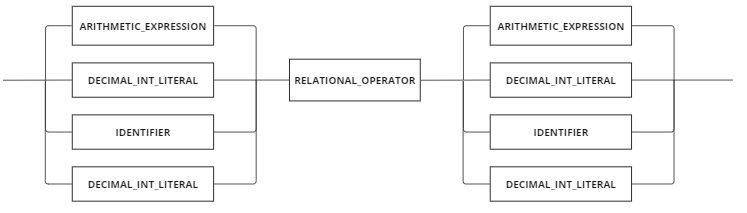
**Арифметическое выражение:**



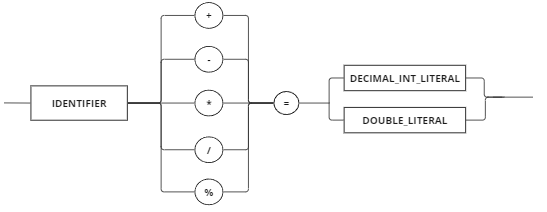
**Логическое выражение:**



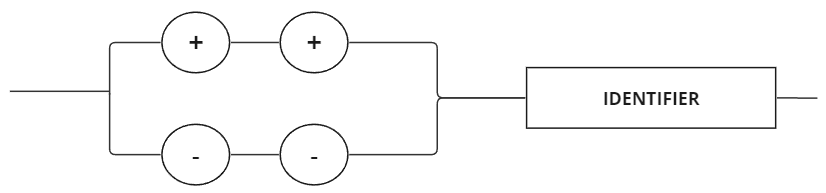
**Условное выражение:**



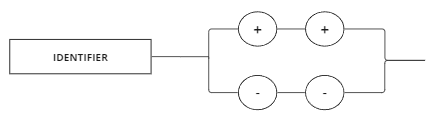
**Выражение арифметического присваивания:**



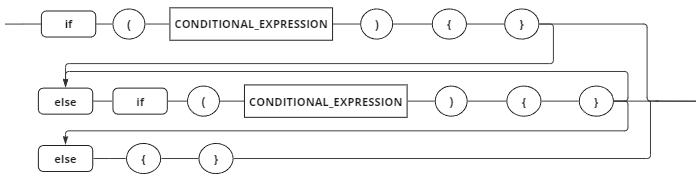
**Префиксное выражение:**



**Постфиксное выражение:**



**Условный оператор:**



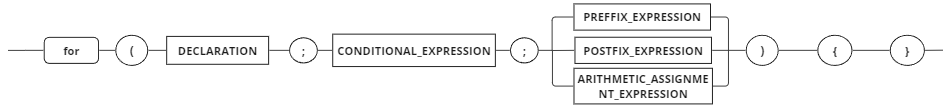
**Цикл while:**



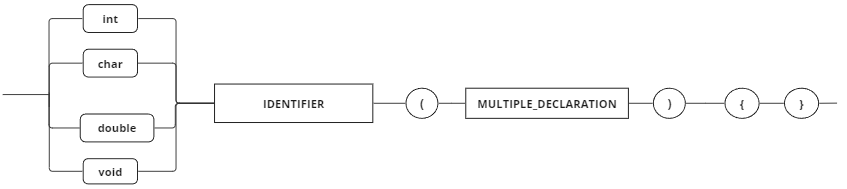
**Цикл do-while:**



**Цикл for:**



**Определение функции:**



**ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Лист регистрации изменений | | | | | | | | | |
| Номера листов (страниц) | | | | | Всего листов (страниц в докум.) | № документа | Входящий № сопроводительного докум. и дата | Подп. | Дата |
| Изм. | Измененных | Замененных | Новых | Аннулированных |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |