Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра компьютерной безопасности

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
канд. техн. наук, доцент
_____ В.Н. Тренькаев
«17» 01 2022 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА СПЕЦИАЛИСТА (ДИПЛОМНАЯ РАБОТА)

РАЗРАБОТКА КОМПИЛЯТОРА ДЛЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ ЛЯПАС

по специальности 10.05.01 Компьютерная безопасность,
специализация (профиль) «Анализ безопасности компьютерных систем»

Насртдинов Альберт Ильмирович

Научный руководитель ВКР
канд.физ.-мат.наук, доцент
_____ И.А. Панкратова
«17» 01 2022 г.

Консультант программист
лаборатории компьютерной
криптографии
_____ Е.М. Коршиков
«17» 01 2022 г.

Автор работы
студент группы № 1165
_____ А.И. Насртдинов
«17» 01 2022 г.

Томск – 2022

Министерство науки и высшего образования Российской Федерации.
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Наименование учебного структурного подразделения

УТВЕРЖДАЮ
Руководитель ООП
канд. техн. наук, доцент

_____ В.Н. Тренькаев

«02» 10 2021 г.

ЗАДАНИЕ

по выполнению выпускной квалификационной работы специалиста обучающемуся Насртдинову Альберту Ильмировичу
по направлению подготовки 10.05.01 Компьютерная безопасность, специализация (профиль) «Анализ безопасности компьютерных систем»

1 Тема выпускной квалификационной работы
Разработка компилятора для языка программирования ЛЯПАС

2 Срок сдачи обучающимся выполненной выпускной квалификационной работы:
а) в учебный офис / деканат – 17.01.2022 б) в ГЭК – 28.01.2022

3 Исходные данные к работе:
Объект исследования – язык программирования ЛЯПАС
Предмет исследования – компилятор для языка программирования ЛЯПАС
Цель исследования – разработка компилятора языка ЛЯПАС на языке ЛЯПАС
Задачи:

1. Разработать лексический анализатор языка ЛЯПАС
2. Разработать детерминированный синтаксический анализатор для языка ЛЯПАС
3. Формально описать грамматику языка ЛЯПАС
4. Разработать дерево синтаксического разбора
5. Разработать промежуточный код для дальнейшей компиляции под целевую платформу
6. Разработать и описать необходимые структуры данных и подпрограммы для работы с ними

Методы исследования: теоретический, экспериментальные на базе ЭВМ.
Организация или отрасль, по тематике которой выполняется работа, – лаборатория компьютерной криптографии ТГУ

4 Краткое содержание работы
Алгоритмы, программы и описания, решающие поставленные задачи

Руководитель выпускной квалификационной работы
канд.физ.-мат.наук, доцент _____ / Панкратова И.А.

Задание принял к исполнению
студент гр. 1165 _____ / Насртдинов А.И.

АННОТАЦИЯ

Дипломная работа содержит 35 страниц, 1 рисунок, 4 литературных источника.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ, КОМПИЛЯТОРЫ.

Объект исследования: язык программирования ЛЯПАС

Цель работы: разработка компилятора языка ЛЯПАС на языке ЛЯПАС

Метод исследования: теоретический и экспериментальный на базе ЭВМ.

Результат: Разработан и реализован прототип компилятора для языка программирования ЛЯПАС. Описаны структура и особенности компилятора и каждого его компонента.

Реализованы и описаны новые структуры данных, особенности реализации в контексте языка ЛЯПАС.

Разработан и описан выходной промежуточный код для дальнейшей компиляции под целевую архитектуру и ОС.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

На кафедре ведутся работы по возрождению отечественного языка программирования ЛЯПАС. Сейчас язык ЛЯПАС расширен до ЛЯПАСа-Т (добавлена работа с большими числами и длинными векторами) и ЛЯПАСа-М (добавлена поддержка модульного программирования). В настоящий момент есть действующий и активно используемый в учебных целях транслятор, работающий под управлением ОС Linux.

Существующий компилятор производит исполняемые файлы, жестко привязанные к ОС Linux, т.е. исполнение этих программ невозможно на машине без ОС или на машине с другой ОС. Также компилятор имеет ряд архитектурных проблем, мешающих развитию как компилятора, так и языка.

В рамках работы использовалась теория лексического и синтаксического анализов. Были разработаны новые структуры данных как интерпретации стандартного типа языка ЛЯПАС - одномерного комплекса.

Для удобства отладки системы были реализованы модули визуализации дерева разбора и промежуточного кода.

1 Обзор проблемы

1.1 Существующая реализация

Существующий компилятор запрограммирован на языке C++ с использованием библиотеки регулярных выражений. Исходный текст программы подается на вход регулярным выражениям, когда находится совпадение, осуществляется трансляция найденного куска в ассемблер. Затем код на языке ассемблер компилируется в исполняемый файл с использованием утилит ОС Linux (nasm и ld). Такой подход прост в реализации, однако влечет за собой следующие проблемы:

1. От того, какое регулярное выражение проверяется первым, зависит результат трансляции.
2. Отсутствуют границы между лексическим, синтаксическим, семантическим и другими анализами.
3. Регулярные выражения заданы так, что допускают корректное распознавание незначащих конструкций, что влечет за собой возможную потерю производительности
4. Так как исходный текст программы транслируется в ассемблер, теряется связь с исходным текстом программы, тем самым затрудняется исправление ошибок.

1.2 Обоснование выбора языка

В качестве языка программирования для написания компилятора выбран язык ЛЯПАС. Для этого есть две важные причины:

1. Частичная независимость от стороннего ПО. Мы зависимы от ОС Linux и существующего компилятора.
2. Дальнейшая раскрутка компилятора. Опишем алгоритм раскрутки.
    a. Существующий компилятор назовем К1, новый разрабатываемый компилятор назовем К2
    b. Компилируем К2 компилятором К1
    c. На некотором этапе развития компилятора К2 у нас будет возможность скомпилировать К2, используя компилятор К2
    d. Начиная с этого момента, мы зависим только от управляющей ОС

1.3 Постановка задачи

Учитывая все проблемы, было принято решение о разработке нового компилятора с использованием формальных подходов к компиляции.

Цель: разработка компилятора языка ЛЯПАС на языке ЛЯПАС.

Задачи:

1. Разработать лексический анализатор языка ЛЯПАС
2. Разработать детерминированный синтаксический анализатор для языка ЛЯПАС
3. Формально описать грамматику языка ЛЯПАС
4. Разработать дерево синтаксического разбора
5. Разработать промежуточный код для дальнейшей компиляции под целевую платформу
6. Разработать и описать необходимые структуры данных и подпрограммы для работы с ними

2 Предлагаемое решение

2.1 Лексический анализ

Лексическим анализом называют процесс аналитического разбора входной последовательности символов на распознанные группы - лексемы (токены) [1].

Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п. [1].

Лексический анализатор не является обязательной частью компилятора, но его использование обусловлено несколькими причинами [1]:

1. Упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокращается объем обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и удаляет всю незначащую информацию
2. Анализатор отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка – при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой анализатор

2.2 Синтаксический анализ

Синтаксическим анализом называют процесс сопоставления линейной последовательности лексем (токенов) формального языка с его формальной грамматикой.
Есть два типа распознающих алгоритмов:

1. Нисходящий распознаватель
2. Восходящий распознаватель

Каждый из них имеет свои преимущества и недостатки.

В качестве анализатора выбран нисходящий детерминированный синтаксический LL(1) распознаватель [1].

2.3 Грамматика

В рамках данной работы была разработана LL(1) грамматика, соответствующая

описанию языка:

SUBPROGRAM -> HEAD BODY RET SUBPROGRAM | ε

RET -> **

HEAD -> NAME (INARGS / OUTARGS)

NAME -> name

INARGS -> INARG NEXTINARG | ε

NEXTINARG -> , INARG NEXTINARG | ε

INARG -> id | complex

OUTARGS -> OUTARG NEXTOUTARG | ε

NEXTOUTARG -> , OUTARG NEXTOUTARG | ε

OUTARG -> id | complex

BODY -> EXPR NEXTEXPR | ε

NEXTEXPR -> EXPR NEXTEXPR | ε

EXPR -> assign ASSIGNABLEOPERAND | OPERAND | complex INDEX

| swap (SWAPEXPR) | zeroing ASSIGNABLEOPERAND

| maximize ASSIGNABLEOPERAND

| lowerone | inversion | weighing | ARITHMETICOP ARITHMETICOPERAND

| * STAREXPR | / SLASHEXPR | inc ASSIGNABLEOPERAND

| dec ASSIGNABLEOPERAND | JUMPOP const

| cmp (LEFTCONDOPERAND CONDITIONOP RIGHTCONDOPERAND) const

| par const | at COMPLEXEXPR

STAREXPR -> name (CALLINARGS / CALLOUTARGS)

| OPERAND | complex INDEX

SLASHEXPR -> complex COMPLEXSLASHEXPR

| string CONSOLEACTION console | OPERAND

COMPLEXSLASHEXPR -> CONSOLEACTION console | INDEX

COMPLEXEXPR -> + complex (const) | zeroing complex | string g complex

SWAPEXPR -> id id | complex SWAPFIRST SWAPSECOND

SWAPFIRST -> id | . const

SWAPSECOND -> id | const

CONSOLEACTION -> l | g

JUMPOP -> jz | jnz | jump

INDEX -> . const | id

ARITHMETICOP -> + | - | ; | and | or | xor | g | l

CONDITIONOP -> e | ne | g | ge | l | le

OPERAND -> id | const | length | capacity | random | singleconst | time

ASSIGNABLEOPERAND -> id | complex INDEX | length | random

CALLINARGS -> CALLINARG NEXTCALLINARG | ε

NEXTCALLINARG -> , CALLINARG NEXTCALLINARG | ε

CALLINARG -> id | const | complex CALLINDEX

CALLOUTARGS -> CALLOUTARG NEXTCALLOUTARG | ε

NEXTCALLOUTARG -> , CALLOUTARG NEXTCALLOUTARG | ε

CALLOUTARG -> id | complex CALLINDEX

CALLINDEX -> . const | id | ε

ARITHMETICOPERAND -> OPERAND | complex INDEX

LEFTCONDOPERAND -> OPERAND | complex INDEX

RIGHTCONDOPERAND -> OPERAND | complex INDEX

2.4 Синтаксическое дерево

Выходом синтаксического анализатора является дерево разбора. Дерево разбора программы на языке ЛЯПАС имеет строго заданную структуру, представленную на рисунке 1.

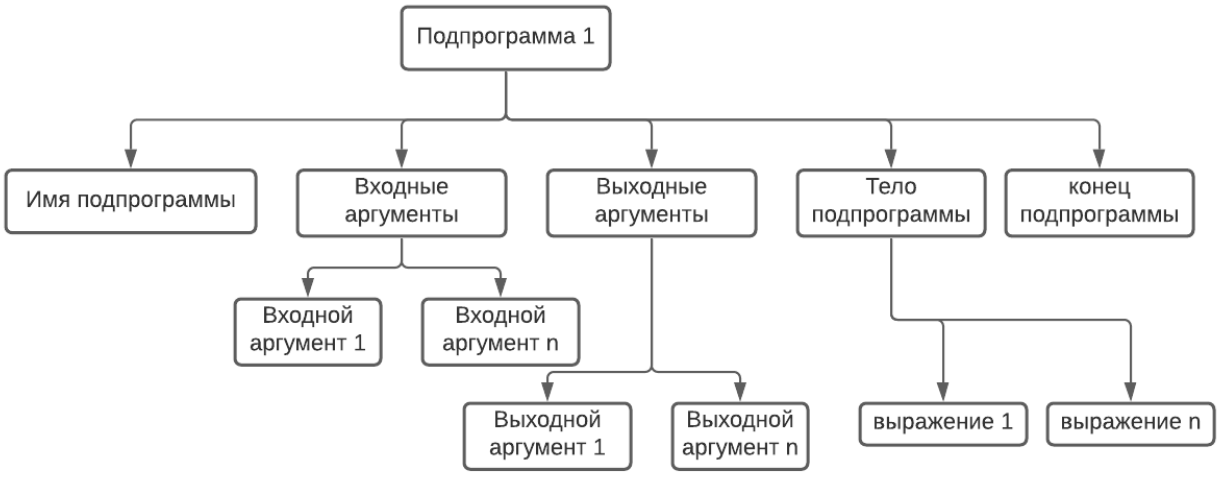


Рисунок 1 - Структура дерева разбора

Каждый узел дерева представлен следующей структурой: идентификатор, значение и список дочерних узлов. Ниже описаны все узлы с их идентификаторами и описанием значений и дочерних узлов.

Подпрограмма

Идентификатор: 0

Значение: номер подпрограммы

Потомки:

1. Имя подпрограммы
2. Входные аргументы
3. Выходные аргументы
4. Тело подпрограммы
5. Конец подпрограммы

Имя подпрограммы

Идентификатор: 1

Значение: токен названия подпрограммы

Потомки: отсутствуют

Входные аргументы

Идентификатор: 2

Значение: отсутствует

Потомки:

1. Переменная, комплекс

Количество потомков определяется количеством входных аргументов подпрограммы.

Выходные аргументы

Идентификатор: 3

Значение: отсутствует

Потомки:

1. Переменная, комплекс

Количество потомков определяется количеством входных аргументов подпрограммы.

Тело подпрограммы

Идентификатор: 4

Значение: отсутствует

Потомки: количество и тип потомков определяются использованными в подпрограмме конструкциями языка.

Конец подпрограммы

Идентификатор: 5

Значение: отсутствует

Потомки: отсутствуют.

Переменная

Идентификатор: 6

Значение: токен с именем переменной

Потомки: отсутствуют.

Комплекс

Идентификатор: 7

Значение: токен с именем комплекса

Потомки: отсутствуют.

Элемент комплекса

Идентификатор: 8

Значение: токен с именем комплекса

Потомки:

1. Переменная или константа

Константа

Идентификатор: 9

Значение: токен со значением константы

Потомки: отсутствуют.

Сложение

Идентификатор: 10

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Вычитание

Идентификатор: 11

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Умножение

Идентификатор: 12

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Целочисленное деление

Идентификатор: 13

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Деление по модулю

Идентификатор: 14

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Побитовое ИЛИ

Идентификатор: 15

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Побитовое И

Идентификатор: 16

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Побитовое исключающее ИЛИ

Идентификатор: 17

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Сохранение внутренней переменной

Идентификатор: 18

Значение: отсутствует

Потомки:

1. Переменная, элемент комплекса, мощность комплекса, ГСПЧ

Обмен значений переменных

Идентификатор: 19

Значение: отсутствует

Потомки:

1. Переменная
2. Переменная

Обмен значений элементов комплекса

Идентификатор: 19

Значение: отсутствует

Потомки:

1. Комплекс
2. Переменная
3. Переменная

Выбор одного из двух обменов значений определяется количеством потомков.

Обнуление значения

Идентификатор: 20

Значение: отсутствует

Потомки:

1. Переменная, элемент комплекса, мощность комплекса

Установление максимального значения

Идентификатор: 21

Значение: отсутствует

Потомки:

1. Переменная, элемент комплекса, мощность комплекса

Номер младшей единицы

Идентификатор: 22

Значение: отсутствует

Потомки: отсутствуют.

Инверсия

Идентификатор: 23

Значение: отсутствует

Потомки: отсутствуют.

Взвешивание

Идентификатор: 24

Значение: отсутствует

Потомки: отсутствуют.

Присвоение внутренней переменной

Идентификатор: 25

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Левый сдвиг

Идентификатор: 26

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Правый сдвиг

Идентификатор: 27

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т

Положительное приращение значения

Идентификатор: 28

Значение: отсутствует

Потомки:

1. Переменная, элемент комплекса, мощность комплекса

Отрицательное приращение значения

Идентификатор: 29

Значение: отсутствует

Потомки:

1. Переменная, элемент комплекса, мощность комплекса

Переход по нулю

Идентификатор: 30

Значение: отсутствует

Потомки:

1. Константа

Переход по не нулю

Идентификатор: 31

Значение: отсутствует

Потомки:

1. Константа

Безусловный переход

Идентификатор: 32

Значение: отсутствует

Потомки:

1. Константа

Сравнение

Идентификатор: 33

Значение: отсутствует

Потомки:

1. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т
2. Равно, не равно, меньше, меньше или равно, больше, больше или равно
3. Переменная, константа, элемент комплекса, мощность комплекса, емкость комплекса, единичная константа, ГСПЧ, Т
4. Константа

Метка

Идентификатор: 34

Значение: отсутствует

Потомки:

1. Константа

Создание комплекса

Идентификатор: 35

Значение: отсутствует

Потомки:

1. Комплекс
2. Константа

Обнуление элементов комплекса

Идентификатор: 36

Значение: отсутствует

Потомки:

1. Комплекс

Добавление строки к комплексу

Идентификатор: 37

Значение: отсутствует

Потомки:

1. Комплекс
2. Строка

Строка

Идентификатор: 38

Значение: токен со значением строки

Потомки: отсутствуют.

Мощность комплекса

Идентификатор: 39

Значение: токен с номером комплекса

Потомки: отсутствуют.

Емкость комплекса

Идентификатор: 40

Значение: токен с номером комплекса

Потомки: отсутствуют.

ГСПЧ

Идентификатор: 41

Значение: отсутствует

Потомки: отсутствуют.

Единичная константа

Идентификатор: 42

Значение: токен с номером разряда единицы

Потомки: отсутствуют.

Т

Идентификатор: 43

Значение: отсутствует

Потомки: отсутствуют.

Равно

Идентификатор: 44

Значение: отсутствует

Потомки: отсутствуют.

Не равно

Идентификатор: 45

Значение: отсутствует

Потомки: отсутствуют.

Больше

Идентификатор: 46

Значение: отсутствует

Потомки: отсутствуют.

Больше или равно

Идентификатор: 47

Значение: отсутствует

Потомки: отсутствуют.

Меньше

Идентификатор: 48

Значение: отсутствует

Потомки: отсутствуют.

Меньше или равно

Идентификатор: 49

Значение: отсутствует

Потомки: отсутствуют.

Вызов подпрограммы

Идентификатор: 50

Значение: отсутствует

Потомки: отсутствуют.

Вызов консоли

Идентификатор: 51

Значение: отсутствует

Потомки:

1. Комплекс или строка
2. Запись или Чтение

Запись

Идентификатор: 52

Значение: отсутствует

Потомки: отсутствуют.

Чтение

Идентификатор: 53

Значение: отсутствует

Потомки: отсутствуют.

2.5 Промежуточный код

Промежуточный код содержит две основные единицы: команды и аргументы. Команды определяют операцию, аргументы определяют набор данных, над которыми данная операция должна быть выполнена.

Разработанный промежуточный код является префиксным кодом, то есть однозначно разбивается на последовательность команд и аргументов. Структура промежуточного кода команд имеет следующий вид: первый байт - идентификатор команды, далее идут байты аргументов. Аргументы, как и команды, могут иметь идентификаторы, если это необходимо.

Ниже приведено описание аргументов:

Индекс подпрограммы (subprogram index)

Идентифицирующий байт: не требуется. Значение индекса хранится как четыре байта с порядком байтов little endian.

Операнд (operand)

Операндами являются следующие лексемы и их идентифицирующие байты: переменная (0x00), константа (0x01), мощность комплекса (0x02), емкость комплекса (0x03), ГСПЧ (0x04), единичная константа (0x05), T (0x06), элемент комплекса (0x07).

Значение или индекс операнда хранится в четырёх байтах с порядком байтов little endian.

Присваиваемый операнд (assignable operand)

Присваиваемыми операндами являются те операнды, в которые можно записать значение, а именно: переменная, мощность комплекса, ГСПЧ, элемент комплекса.

Идентифицирующий байт: не требуется. Значение или индекс операнда хранится в четырёх байтах с порядком байтов little endian.

Условие сравнения (condition)

В языке ЛЯПАС представлены следующие условия сравнения: равно (0x00), не равно (0x01), меньше (0x04), больше (0x02), меньше или равно (0x05), больше или равно (0x03).

Строка (string)

Строка - набор символов в кодировке UTF-8 в одинарных кавычках. Идентифицирующий байт: 0x09

Действие с консолью (action)

Определены два действия: чтение (0x00) и запись (0x01).

Комплекс (complex)

Идентифицирующий байт: 0x08

Описание команд:

Подпрограмма

subprogram <subprogram index>

Определяет начало подпрограммы. Идентифицирующий байт: 0x00. Аргументом является индекс подпрограммы, который формируется контекстом синтаксического анализа.

Возврат

ret

Определяет конец подпрограммы. Идентифицирующий байт: 0x01. Аргументов нет.

Сложение

adda <operand>

Прибавить к значению внутренней переменной значение операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x02. Аргумент - операнд.

Вычитание

suba <operand>

Отнять от значения внутренней переменной значение операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x03. Аргумент - операнд.

Умножение

mula <operand>

Умножить значение внутренней переменной на значение операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x04. Аргумент - операнд.

Целочисленное деление

diva <operand>

Вычислить целую часть от деления внутренней переменной на значение операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x05. Аргумент - операнд.

Деление по модулю

moda <operand>

Вычислить остаток от деления значения внутренней переменной на значение операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x06. Аргумент - операнд.

Побитовое ИЛИ

ora <operand>

Побитовое ИЛИ значения внутренней переменной и значения операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x07. Аргумент - операнд.

Побитовое И

anda <operand>

Побитовое И значения внутренней переменной и значения операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x08. Аргумент - операнд.

Побитовое исключающее ИЛИ

xora <operand>

Побитовое исключающее ИЛИ значения внутренней переменной и значения операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x09. Аргумент - операнд.

Сохранение значения внутренней переменной

savea <assignable operand>

Переместить значение внутренней переменной в присваиваемый операнд. Идентифицирующий байт: 0x0A. Аргумент - присваиваемый операнд.

Обмен значений

swap <variable> <variable>

Обмен значений переменных или элементов комплекса. Идентифицирующий байт: 0x0B. Аргументы - переменные.

swap <complex> <index> <index>

Первый аргумент - номер комплекса. Второй и третий - индексы элементов комплекса.

Обнуление значения присваиваемого операнда

zeroing <assignable operand>

Поместить нулевое значение в присваиваемый операнд. Идентифицирующий байт: 0x0C. Аргумент - присваиваемый операнд.

Присваивание максимального значения присваиваемому операнду

maximize <assignable operand>

Поместить максимальное значение в присваиваемый операнд. Идентифицирующий байт: 0x0D. Аргумент - присваиваемый операнд.

Номер младшей единицы

lowerone

Вычислить номер младшей единицы значения внутренней переменной. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x0E. Аргументов нет.

Инверсия

inversion

Вычислить инверсию значения внутренней переменной. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x0F. Аргументов нет.

Взвешивание

weighing

Вычислить вес значения внутренней переменной. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x10. Аргументов нет.

Загрузка значения операнда во внутреннюю переменную

loada <operand>

Загрузить значение операнда во внутреннюю переменную. Идентифицирующий байт: 0x11. Аргумент - операнд.

Левый сдвиг

lsha <operand>

Левый сдвиг значения внутренней переменной на значение операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x12. Аргумент - операнд.

Правый сдвиг

rsha <operand>

Правый сдвиг значения внутренней переменной на значение операнда. Результат поместить во внутреннюю переменную. Идентифицирующий байт: 0x13. Аргумент - операнд.

Положительное приращение присваиваемого операнда

inc <assignable operand>

Увеличить значение присваемого операнда на 1. Идентифицирующий байт: 0x14. Аргумент - присваиваемый операнд.

Отрицательное приращение присваиваемого операнда

dec <assignable operand>

Уменьшить значение присваемого операнда на 1. Идентифицирующий байт: 0x15. Аргумент - присваиваемый операнд.

Переход по нулевому значению внутренней переменной

jz <constant>

Переход на метку при нулевом значении внутренней переменной. Идентифицирующий байт: 0x16. Аргумент - десятичная константа.

Переход по ненулевому значению внутренней переменной

jnz <constant>

Переход на метку при ненулевом значении внутренней переменной. Идентифицирующий байт: 0x17. Аргумент - десятичная константа.

Безусловный переход

jmp <constant>

Безусловный переход на метку. Идентифицирующий байт: 0x18. Аргумент - десятичная константа.

Сравнение

cmp <operand> <operand> <condition> <constant>

Сравнить значения операндов, при выполнении условия сравнения переход на метку. Идентифицирующий байт: 0x19. Первый и второй аргументы - сравниваемые операнды. Третий операнд - условие сравнения. Четвертый операнд - номер метки, представленный десятичной константой

Метка

label <constant>

Определяет локальную метку. Идентифицирующий байт: 0x1A. Аргумент - десятичная константа.

Обнуление комплекса

zeroing_complex <complex>

Присвоить всем элементам комплекса нулевое значение. Идентифицирующий байт: 0x1B. Аргумент - комплекс.

Добавление строки к комплексу

append_string <complex> <string>

Добавить строку к комплексу. Идентифицирующий байт: 0x1C. Первый аргумент - комплекс, второй аргумент - строка для добавления.

Вызов подпрограммы

call <subprogram index> <const> <argument1> … <const> <argument1> …

Вызов подпрограммы. Идентифицирующий байт: 0x1D. Первый аргумент - индекс подпрограммы. Второй аргумент - количество передаваемых входных аргументов (n), последующие n аргументов - передаваемые входные аргументы. Следующий аргумент - количество принимаемых выходных аргументов (m). Следующие m аргументов - получаемые выходные аргументы.

Работа с консолью

console <complex> <action>

console <string> <action>

Чтение с консоли в комплекс. Запись комплекса или строки в консоль. Идентифицирующий байт: 0x1E. Первый аргумент - комплекс или строка, второй аргумент - действие с консолью (запись или чтение).

2.6 Соглашения об изменениях

2.6.1 Комплексы

В текущем компиляторе комплексы хранятся в следующем виде:

1. 1 байт для хранения типа комплекса (символьный или логический)
2. 4 байта для хранения ссылки на начало комплекса в куче
3. 4 байта для хранения мощности
4. 4 байта для хранения емкости

Как можно видеть, сам комплекс хранится в куче. Мы решили перенести комплексы из кучи в стек. У данного решения есть плюсы и минусы.

Плюсы:

1. Чтобы использовать кучу, нам необходима управляющая ОС. При хранении комплексов в стеке мы отвязываем работу с комплексами от ОС, на что в конечном этапе мы нацелены.

Минусы:

1. Невозможно увеличить емкость комплекса. Теперь программисту необходимо заранее рассчитывать размер комплекса.

2.6.2 Stack Frame

В существующем компиляторе 26 переменных (a-z), 99 комплексов, мощности, емкости и прочие данные расположены на стеке.

В разрабатываемом компиляторе мы будем использовать следующую структуру стека:

1. Память на стеке выделяется не под все переменные и прочие данные, а только под используемые в данной подпрограмме.
2. Локальные переменные хранятся в стеке, начиная с адреса EBP-8 и далее к младшим адресам (EBP-16 и т.д.)
3. К аргументам подпрограммы, начиная с седьмого, доступ осуществляется по адресам, начиная с EBP+16 и далее к старшим адресам (EBP+24 и т.д.)

2.6.3 Соглашение об определении подпрограмм

Текущий компилятор определяет точку входа в программу как самую верхнюю подпрограмму. Вне зависимости от места определения подпрограммы в файле, ее можно вызывать из любой подпрограммы. В разрабатываемом компиляторе точка входа

определена подпрограммой с именем head. Также вызывать подпрограммы теперь можно только из подпрограмм, описанных ниже.

Пример:

Допустимо:

func(a/b) a*2⇒b **

head(/) *func(1/a) **

Ошибка:

head(/) *func(1/a) **

func(a/b) a*2⇒b **

2.6.4 Соглашение о вызове подпрограмм

В текущей реализации компилятора входные и выходные аргументы передаются через стек. В новой версии компилятора мы будем использовать модифицированную версию соглашения о вызове System V AMD64 ABI. А именно:

1. Первые 6 аргументов передаются через регистры (RDI, RSI, RDX, RCX, R8, R9), остальные через стек
2. Аргументы передаются слева направо
3. Ввиду возможности возврата нескольких значений из подпрограммы, мы ограничиваем количество возможных возвращаемых до 7. И возвращаться они будут через регистры (RAX, RDI, RSI, RDX, RCX, R8, R9)
4. Переменные передаются по значению, комплексы по ссылке
5. Очистка стека происходит в вызывающей подпрограмме

Регистры (RBX, RSP, RBP, R12–R15) являются stable регистрами

3 Программная реализация

3.1 Структуры данных

В языке ЛЯПАС существует всего 3 типа данных: переменные и 2 типа комплексов. Для реализации лексического и синтаксического анализаторов этих типов данных недостаточно. Необходимы более сложные структуры данных, такие как массивы определенного вида структур, стек и другие.

Вышеперечисленные структуры были реализованы с помощью существующих типов - комплексов и подпрограмм, которые определенным образом работают с этими комплексами.

3.1.1 Массив токенов

Данная структура использует два комплекса, F1 и L2. В F1 хранится символьное представление токенов. Структура L2 представлена ниже:

1. L2(0) - количество токенов
2. L2(1 + 4 * i) - указатель на начало токена в F1, где i - индекс токена
3. L2(2 + 4 * i) - длина токена, где i - индекс токена
4. L2(3 + 4 * i) - id токена, где i - индекс токена
5. L2(4 + 4 * i) - позиция в тексте, где был найден токен, где i - индекс токена

Для работы с массивом используются следующие подпрограммы:

1. init_token_array(/F1,L2) - инициализирует массив
2. get_token_id(L1,i/d) - возвращает id токена по его индексу
3. get_token_position(L1,i/p) - возвращает позицию токена в тексте
4. get_token(F1,L2,i/F3) - возвращает символьное представление токена по индексу
5. add_token(F1,L2,F3,i,p/F1,L2) - добавляет токен в конец массива

3.1.2 Стек

Данная структура использует один комплекс L1 и 6 подпрограмм:

1. stack_push(L1,L2/L1) - добавляет все элементы комплекса L2 в стек
2. stack_push_single(L1,x/L1) - добавляет один элемент в стек
3. stack_pop(L1,k/L1,L2) - извлекает k элементов из стека в L2
4. stack_pop_single(L1/L1,x) - извлекает один элемент из стека
5. stack_peek(L1/x) - возвращает элемент на вершине стека

6. stack_has_values(L1/r) - проверяет наличие значений в стеке

3.1.3 Массив правил грамматики

Структура использует два комплекса, L1 и L2. В L1 хранятся правила грамматики. В L2 хранятся данные со следующей структурой:

1. L2(0) - количество правил
2. L2(1 + 3 * i) - указатель на начало правила в L1, где i - индекс токена
3. L2(2 + 3 * i) - длина правила, где i - индекс токена
4. L2(3 + 3 * i) - номер альтернативы, где i - индекс токена

Для работы со структурой реализованы следующие подпрограммы:

1. init_rules_array(/L1,L2) - инициализирует массив
2. get_rule(L1,L2,l,i/L3,r) - возвращает правило по значению левой части и номеру альтернативы
3. get_rule_by_id(L1,L2,i/L3,r) - возвращает правило по индексу
4. add_rule(L1,L2,L3,i/L1,L2) - добавляет правило в массив

3.1.4 Массив структур фиксированного размера

Структура - набор 32-битных полей.

Описание хранения массива в памяти:

1. L1 - массив (32-битные ячейки)
2. L1.0 - количество элементов
3. L1.1 - размер одной структуры
4. L1.2..n - структуры длины L1.1

Подпрограммы для работы:

1. init_fixed_size(n / L1) - инициализация массива
2. add_fixed_size(L1,L2/r) - добавление структуры в массив
3. remove_fixed_size(L1, i / r) - удаление элемента из массива
4. get_fixed_size(L1,i/L2,r) - получить элемент в виде комплекса
5. get_field_fixed_size(L1, i, j / f) - получить j-ое поле i-го элемента массива
6. update_fixed_size(L1, i, L2/) - изменить i элемент массива
7. update_field_fixed_size(L1, i, j, f/) - изменить j-ое поле i-го элемента массива
8. clear_fixed_size(L1 /) - очистить массив
9. get_size_fixed_size(L1/n) - возвращает количество элементов в массиве

3.1.5 Таблица синтаксического анализа

Для работы детерминированного синтаксического анализа необходима управляющая таблица. В разрабатываемом компиляторе для реализации таблицы используется массив структур фиксированного размера, описанный выше. Работа с синтаксической таблицей осуществляется с помощью трёх подпрограмм:

1. init_syntax_table(/L1) - инициализация таблицы
2. add_syntax_table_item(L1,n,t,i/r) - добавить переход в таблицу. Здесь n - нетерминал на вершине стека, t - текущий терминал во входной цепочке, i - номер правила, который необходимо применить
3. try_get_syntax_table_transition(L1,n,t/i,r) - получить переход для заданных нетерминала и терминала. Если переход есть в таблице, в i будет храниться номер правила, а в r - значение больше нуля. В случае, если перехода нет, в r будет значение 0

3.1.6 Контекст синтаксического анализатора

Данная структура данных служит дополнением к генерируемому промежуточному коду.

Структура контекста представлена ниже:

1. L1 - массив указателей, является массивом структур фиксированного размера.
    a. 0 элемент - указатель на начало названия подпрограммы
    b. 1 элемент - длина подпрограммы
    c. 2 элемент - указатель на начало входных аргументов подпрограммы
    d. 3 элемент - количество входных аргументов подпрограммы
    e. 4 элемент - указатель на начало выходных аргументов подпрограммы
    f. 5 элемент - количество выходных аргументов подпрограммы
    g. 6 элемент - указатель на начало переменных подпрограммы
    h. 7 элемент - количество переменных подпрограммы
    i. 8 элемент - указатель на начало комплексов подпрограммы
    j. 9 элемент - количество комплексов подпрограммы
2. F2 - массив названий подпрограмм
3. L3 - массив входных аргументов (массив структур фиксированного размера)
    a. 0 элемент - тип аргумента (переменная/комплекс)
    b. 1 элемент - значение (индекс переменной или комплекса, значение константы)

4. L4 - массив выходных аргументов (массив структур фиксированного размера)
    a. 0 элемент - тип аргумента (переменная/комплекс)
    b. 1 элемент - значение (индекс переменной, индекс комплекса)
5. F5 - массив переменных
6. L6 - массив комплексов (массив структур фиксированного размера)
    a. 0 элемент - тип комплекса (символьный или логический)
    b. 1 элемент - индекс комплекса
    c. 2 элемент - емкость комплекса
    d. 3 элемент - тип доступа (по ссылке или по значению)
7. L7 - массив указателей на начало и конец строк
8. F8 - массив строк

Для удобной работы с контекстом реализованы следующие подпрограммы:

1. init_context(L1,F2,L3,L4,F5,L6,L7,F8/) - инициализация контекста
2. add_subprogram(L1,F2,L3,L4,F5,L6,F7/i,r) - добавить подпрограмму в контекст
3. get_subprogram_index_by_name(L1,F2,F3/i,r) - получить индекс подпрограммы по имени
4. get_input_args_start(L1,i/s) - получить индекс начала входных аргументов
5. get_input_args_length(L1,i/l) - получить количество входных аргументов для подпрограммы
6. add_input_arg(L1,L3,i,t,v/j,r) - добавить входной аргумент подпрограммы
7. get_input_arg(L3,i/t,v) - получить выходной аргумент подпрограммы
8. get_output_args_start(L1,i/s) - получить индекс начала выходных аргументов
9. get_output_args_length(L1,i/l) - получить количество выходных аргументов для подпрограммы
10. add_output_arg(L1,L4,i,t,v/j,r) - добавить выходной аргумент подпрограммы
11. get_output_arg(L4,i/t,v) - получить входной аргумент подпрограммы
12. get_variables_start(L1,i/s) - получить индекс начала переменных подпрограммы
13. get_variables_length(L1,i/l) - получить количество переменных подпрограммы
14. get_or_add_variable(L1,F5,i,v/j) - добавить переменную или получить ранее добавленную переменную
15. get_complexes_start(L1,i/s) - получить индекс начала комплексов подпрограммы
16. get_complexes_length(L1,i/l) - получить количество комплексов подпрограммы
17. add_complex(L1,L6,i,t,d,c,a/j,r) - добавить комплекс в подпрограмму
18. get_complex(L6,i/t,d,c,a) - получить комплекс по индексу

19. get_complex_id_by_name(L1,L6,i,t,d/j,r) - получить индекс комплекса подпрограммы по имени
20. get_or_add_string(L7,F8,F10/j) - добавить строку или получить индекс ранее добавленную строку
21. get_string_by_id(L7,F8,i/F10) - получить строку по индексу

Такая организация контекста и подпрограмм возможна, так как все данные, необходимые для формирования контекста, поступают всегда в определенном порядке.

3.1.7 Дерево

Для реализации данной структуры используются два логических комплекса и следующие подпрограммы:

1. init_tree(L1,L2/L1,L2) - инициализация дерева
2. add_node(L1,L2,p,i,v/L1,L2,n) - добавление нового узла к дереву
3. get_node_info(L1,L2,n/i,v) - получить идентификатор и значение узла
4. update_node_info(L1,L2,n,i,v/) - обновить идентификатор и значение узла
5. get_node_child_count(L1,L2,n/c) - получить количество дочерних узлов
6. get_child_node_index(L1,L2,p,j/i) - получить индекс j-го дочернего узла
7. get_parent_node_index(L1,L2,n/p) - получить индекс родительского узла

3.2 Структура компилятора

Программа разделена на четыре части:

1. Чтение исходного текста программы на языке ЛЯПАС
2. Лексический анализ исходного текста и разбиение его на токены
3. Синтаксический анализ последовательности токенов, полученных на предыдущем шаге и формирование дерева разбора
4. Обход дерева разбора с проверкой семантики языка и дальнейшим формированием промежуточного кода

3.2.1 Чтение исходного текста

С использованием ассемблерных вставок [3] программа компилятора получает название файла с исходным текстом и записывает его в комплекс F1:

```
{mov esi, [esp + 1436]} {xor eax, eax}
§1 {mov al, byte [esi]} {push esi} @>F1 {pop esi} {inc esi} ↦1
```

В спецификации ОС Linux указано, что при запуске программы по адресу esp+8 будет находиться указатель на первый аргумент [4]. Учитывая это и особенности существующего компилятора [3], получаем адрес esp+1436. Затем содержимое файла считывается в комплекс F2:

```
*fopen(F1,0/n) *freadf(n,F2,0,10000/k)
k⇒Q2 *log(F2/) *fclose(n/)
```

3.2.2 Лексический анализ

В качестве лексического анализатора реализована подпрограмма lex. Ниже представлен основной цикл работы подпрограммы:

```
§1 ↑(j≥n)4 ↑(F1j=32)2
        *get_variable(F1,j/r,l,F4) 1⇒i r↦3
        *get_constant(F1,j/r,l,F4) 2⇒i r↦3
        …
   /'Unexpected symbol at position '>C *n2s(j/F4) *log(F4/)→4
§3 *add_token(F2,L3,F4,i,j/F2,L3)  j+l⇒j →1 §4 **
```

Так как язык ЛЯПАС не поддерживает механизм регулярных выражений, нет возможности формально описать все компоненты языка. Поэтому основной цикл состоит из последовательного вызова подпрограмм, которые пытаются из набора исходных символов собрать токен.

Подпрограммы вида get_(lexeme) представляют собой вызов одной из подпрограмм:

1. get_single(F1,p,v/r,l,F2) - проверяет, что текущий просматриваемый символ в исходном тексте совпадает с указанным символом, и возвращает его в виде комплекса.
2. get_entry(F1,p,F2/r,l,F3) - проверяет, что последовательность символов в исходном тексте, начиная с текущего, совпадает с указанной последовательностью, и возвращает последовательность в виде комплекса. Необходимость этой подпрограммы связана с тем, что операторы языка ЛЯПАС представляют собой unicode-символы, которые кодируются более чем одним байтом.

Если одна из подпрограмм распознала токен, он добавляется в массив и указатель на символ в исходном тексте смещается на символ, следующий после конца токена. Если

ни одна из подпрограмм не смогла распознать текущий символ или последовательность символов, выводится сообщение об ошибке и анализ прекращается.

3.2.3 Синтаксический анализ

В качестве анализатора выбран нисходящий распознаватель, т.к. он относительно прост в реализации и универсален. Однако имеет ограничение на грамматику. Нисходящий распознаватель не может работать с леворекурсивными грамматиками.

Синтаксический анализатор представлен подпрограммой llparse, которая представляет собой алгоритм нисходящего детерминированного разбора [2] с модификациями. А именно:

1. После каждого примененного правила вызывается подпрограмма handle_applied_rule.
2. После каждого корректно принятого терминала вызывается подпрограмма handle_accepted_terminal.

В качестве аргументов данные подпрограммы принимают дерево синтаксического разбора и возвращают модифицированное дерево. Для корректного построения дерева дополнительно к дереву передается стек элементов следующей структуры:

1. номер последнего примененного правила
2. номер узла дерева разбора
3. количество терминалов и нетерминалов в правой части правила
4. количество обработанных терминалов и нетерминалов в правой части правила

При каждом вызове подпрограммы handle_applied_rule у элемента на вершине стека счетчик количества обработанных терминалов и нетерминалов увеличивается на 1. Затем вызывается подпрограмма - обработчик примененного правила в которую передаются: дерево разбора, номер узла дерева разбора. Подпрограмма возвращает измененное дерево и новый номер узла. После вызова подпрограммы в стек добавляется элемент со следующими значениями: номер текущего примененного правила, номер узла, который вернула подпрограмма-обработчик, количество терминалов и нетерминалов в правой части примененного правила и 0. Далее вызывается подпрограмма очистки стека. Элемент с вершины стека снимается в том случае, если количество обработанных терминалов и нетерминалов равно количеству терминалов и нетерминалов в правой части правила. Очистка происходит рекурсивно, пока соблюдается условие очистки.

При каждом вызове подпрограммы handle_accepted_terminal подпрограмма считывает данные с вершины стека и в зависимости от номера правила вызывает нужный обработчик. Затем у элемента на вершине стека счетчик количества обработанных

терминалов и нетерминалов увеличивается на 1 с дальнейшим вызовом подпрограммы очистки стека.

Для удобства отладки построения дерева была реализована вспомогательная программа на языке C# для построения дерева по левому выводу.

3.2.4 Генерация промежуточного кода

Следующим этапом является обход дерева разбора, анализ семантики и генерация промежуточного кода. Для генерации промежуточного кода реализована подпрограмма produce_byte_code. На вход ей подается дерево разбора и поток токенов. Подпрограмма возвращает сгенерированный промежуточный код и контекст синтаксического анализа, описанный выше.

Подпрограмма produce_byte_code реализована по шаблону “Посетитель”, т.е. последовательно вызываются подпрограммы-посетители для каждого узла дерева разбора. Каждая подпрограмма-посетитель формирует промежуточный код, наполняет контекст данными, проверяет семантику языка. Ниже приведено описание основных подпрограмм-посетителей:

Посетитель узла *подпрограмма*

1. Формирует команду *подпрограмма*
2. Вызывает подпрограмму-посетителя для узла *входные аргументы*
3. Вызывает подпрограмму-посетителя для узла *выходные аргументы*
4. Вызывает подпрограмму-посетителя для узла *тело подпрограммы*
5. Формирует команду *возврат*

Посетитель узла *входные аргументы*

1. Посещает все узлы с входными аргументами и добавляет их в контекст

Посетитель узла *выходные аргументы*

1. Посещает все узлы с выходными аргументами и добавляет их в контекст

Посетитель узла *тело подпрограммы*

1. Последовательно посещает все дочерние узлы и вызывает подпрограмму-посетитель для каждого узла в зависимости от типа узла.

ЗАКЛЮЧЕНИЕ

В рамках работы были достигнуты следующие результаты:

1. Изучена теория лексического анализа и разработан лексический анализатор
2. Изучена теория синтаксического анализа и разработан синтаксический анализатор
3. Разработаны новые структуры данных, такие как массив структур, стек, дерево и другие.
4. Формально описана грамматика языка ЛЯПАС
5. Разработан промежуточный код для дальнейшей трансляции и компиляции
6. Разработан модуль обхода синтаксического дерева и формирования промежуточного кода.
7. Подготовлен модуль для семантического анализа кода программы

Ознакомиться с исходным кодом программы можно в репозитории GitHub по ссылке https://github.com/Albert1912/Lyapas. Подпрограмма разбита на 16 файлов, каждый из которых содержит код реализации модуля или структуры.

Данную работу можно продолжить. На данном этапе компилятор формирует промежуточный код, не привязанный к архитектуре процессора и ОС. Следующим этапом является дальнейшая трансляция полученного промежуточного кода в машинный код под выбранную архитектуру. Таким образом, можно реализовать модуль трансляции промежуточного кода под машину без ОС.

СПИСОК ЛИТЕРАТУРЫ

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т 1.
2. Грис Д. Построение компиляторов для цифровых вычислительных машин, М., "Мир", 1975.
3. Агибалов Г. П., Липский В. Б., Панкратова И. А. О криптографическом расширении и его реализации для русского языка программирования // Прикладная дискретная математика. 2013. № 3. С. 93–104
4. NASM - Linux Getting Command Line Parameters — URL: https://www.dreamincode.net/forums/topic/285550-nasm-linux-getting-command-line-parameters/ — Дата доступа: 25 декабря 2021. Электрон. дан.

АНТИПЛАГИАТ
ОБНАРУЖЕНИЕ ЗАИМСТВОВАНИЙ

Отчет о проверке на заимствования №1

Автор: Насртдинов Альберт Ильмирович
Проверяющий: Насртдинов Альберт Ильмирович (spellwaver@mail.ru / ID: 9621098)

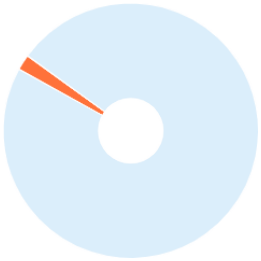
Отчет предоставлен сервисом «Антиплагиат» - users.antiplagiat.ru

ИНФОРМАЦИЯ О ДОКУМЕНТЕ

№ документа: 1
Начало загрузки: 26.01.2022 17:24:19
Длительность загрузки: 00:00:01
Имя исходного файла: BKP.pdf
Название документа: BKP
Размер текста: 46 кБ
Символов в тексте: 47068
Слов в тексте: 5371
Число предложений: 311

ИНФОРМАЦИЯ ОБ ОТЧЕТЕ

Начало проверки: 26.01.2022 17:24:21
Длительность проверки: 00:00:02
Комментарии: не указано
Модули поиска: Интернет Free



ЗАИМСТВОВАНИЯ	САМОЦИТИРОВАНИЯ	ЦИТИРОВАНИЯ	ОРИГИНАЛЬНОСТЬ
1,97%	0%	0%	98,03%

Заимствования — доля всех найденных текстовых пересечений, за исключением тех, которые система отнесла к цитированиям, по отношению к общему объему документа.
Самоцитирования — доля фрагментов текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника, автором или соавтором которого является автор проверяемого документа, по отношению к общему объему документа.
Цитирования — доля текстовых пересечений, которые не являются авторскими, но система посчитала их использование корректным, по отношению к общему объему документа. Сюда относятся оформленные по ГОСТу цитаты; общеупотребительные выражения; фрагменты текста, найденные в источниках из коллекций нормативно-правовой документации.
Текстовое пересечение — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.
Источник — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.
Оригинальность — доля фрагментов текста проверяемого документа, не обнаруженных ни в одном источнике, по которым шла проверка, по отношению к общему объему документа.
Заимствования, самоцитирования, цитирования и оригинальность являются отдельными показателями и в сумме дают 100%, что соответствует всему тексту проверяемого документа.
Обращаем Ваше внимание, что система находит текстовые пересечения проверяемого документа с проиндексированными в системе текстовыми источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности заимствований или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

№	Доля в отчете	Доля в тексте	Источник	Актуален на	Модуль поиска	Блоков в отчете	Блоков в тексте
[01]	1,28%	1,91%	Проектирование компилятора | Рефераты и сочинения https://referat-sochinenie.ru	21 Янв 2022	Интернет Free	2	4
[02]	0%	1,91%	Проектирование компилятора http://worldreferat.ru	16 Июл 2020	Интернет Free	0	4
[03]	0%	1,91%	Методическое пособие для лабораторных занятий По предмету - страница 2 http://uz.denemetr.com	26 Янв 2022	Интернет Free	0	4

Еще источников: 7
Еще заимствований: 0,69%