Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение Высшего профессионального образования

«СЫКТЫВКАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт точных наук и информационных технологий

Кафедра информационной безопасности

ДОПУСТИТЬ К ЗАЩИТЕ

Зав. кафедрой, к.ф.-м.н.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Л.С. Носов

 «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2013 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

**«Интерпретируемый язык для прототипирования криптоалгоритмов»**

Специальность 090104 – Комплексная защита объектов информатизации

|  |  |
| --- | --- |
| Научный руководительДоцент, к.т.н.  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_В.А. Полуботко«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2013 г. |
| Исполнитель:Студент 153 группы  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Б.Р.Липин«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2013 г. |

Сыктывкар 2013

**Содержание**

[Введение 3](#_Toc356528395)

[Список используемых терминов 6](#_Toc356528396)

[Глава 1. Постановка задачи 7](#_Toc356528397)

[1.1. Описание разрабатываемого языка 8](#_Toc356528398)

[Глава 2. Описание общих принципов работы интерпретатора 12](#_Toc356528399)

[2.1 Общие сведения 12](#_Toc356528400)

[2.2 Лексический анализ 14](#_Toc356528401)

[2.2.1 Регулярные выражения 15](#_Toc356528402)

[2.2.2 Ключевые слова 16](#_Toc356528403)

[2.2.3 Таблица ключевых слов 16](#_Toc356528404)

[2.2.4 Удаление пробельных символов и комментариев 17](#_Toc356528405)

[2.3.1 Контекстно свободные грамматики 18](#_Toc356528406)

[2.3.2 Ассоциативность операций 19](#_Toc356528407)

[2.3.3 Приоритет операций 20](#_Toc356528408)

[2.3.4 Парсер рекурсивного спуска 20](#_Toc356528409)

[2.3.5 Устранение левой рекурсии 21](#_Toc356528410)

[2.3.6 Используемые правила грамматики. 23](#_Toc356528411)

[2.3.7 Построение синтаксического дерева 24](#_Toc356528412)

[2.4.1 Таблица символов 25](#_Toc356528413)

[2.4.2 Таблица функций 27](#_Toc356528414)

[2.5 Слабая типизация 28](#_Toc356528415)

[2.6 Библиотека больших чисел 30](#_Toc356528416)

[2.7 Хеш функции 31](#_Toc356528417)

[2.8 Ассоциативные массивы 32](#_Toc356528418)

[2.9 Пользовательские Функции 33](#_Toc356528419)

[Глава 3. Практическое применение языка для подготовки специалистов 34](#_Toc356528420)

[Глава 4. Тестирование программы 35](#_Toc356528421)

[4.2 Поиск и устранение утечек памяти 35](#_Toc356528422)

[4.3 Реализация алгоритмов 36](#_Toc356528423)

[Заключение 37](#_Toc356528424)

[Используемая литература 38](#_Toc356528425)

[1 RSA 39](#_Toc356528426)

[2 Алгоритм Диффи – Хеллмана 40](#_Toc356528427)

[3 Схема Эль-Гамаля 42](#_Toc356528428)

**Введение**

Первые программы писались в машинных кодах, и носителем информации были перфокарты и перфоленты. Программисты должны были досконально знать архитектуру машины, под которую разрабатывались программы. Написание программ в машинном коде оправдывало себя так как компьютеры того времени были достаточно дорогими и медлительными и время программиста ценилось меньше времени работы компьютера.

Первой ласточкой в облегчении жизни программистов стал переход к программированию на ассемблере, в котором команды записывались в символическом виде. Такой незаметный, казалось бы, шаг имел огромное значение — т. к. теперь пропала необходимость запоминать принципы кодирования команд на аппаратном уровне.

Следующим шагом в развитии языков программирования стал fortran — первый язык высокого уровня. Его преимуществом над ассемблером было в первую очередь то, что fortran позволял абстрагироваться от особенностей конкретной машинной архитектуры.

По мере совершенствования техники усложнялись языки и появлялись новые парадигмы программирования. От процедурного программирования, где программа представляется как совокупность взаимосвязанных блоков кода, популярность перешла к объектно-ориентированным (программа представляется как совокупность посылающих друг другу сообщения объектов) и функциональным (процесс вычисления трактуется как вычисление значений функций) языкам.

На текущий момент существует множество языков высокого уровня. По сфере применения их можно разделить на

1. Языки общего назначения (например C/C++, java, python);
2. Специализированные (например matlab, octave, sql, pl/sql, awk, JavaScript).

Как следует из названия первая категория языков предназначена для решения широкого спектра возникающих задач.

Языки второй категории предназначены для удобного решения узкоспециализированных задач, их преимуществом по сравнению с языками общего назначения является наличие удобного инструментария для решения конкретного класса проблем. Конечно, результата можно достигнуть и с помощью языков общего назначения, но часто это требует больших усилий со стороны программиста.

В качестве примера рассмотрим язык Octave, который был разработан специально для решения линейных и нелинейных математических задач, а так же проведения других численных экспериментов. Его отличительными особенностями являются:

1. матрицы используются в качестве основных типов данных;
2. встроенная поддержка комплексных чисел;
3. мощные встроенные математические функции и большие библиотеки функций

Всё это позволяет добиться максимального комфорта для программиста при прототипировании решения математических задач.

Целью моей работы является изучение принципов работы компиляторов и разработка языка, содержащего в себе набор инструментов (специфичные функции, операторы), с помощью которых можно удобно реализовывать и тестировать уже существующие или разрабатывать свои алгоритмы шифрования.

Отличительными особенностями языка должны быть:

1. Нативная поддержка чисел произвольной длинны, что позволяет программисту отвлечься от низкоуровневых задач по контролю памяти и созданию или использованию библиотеки больших чисел;
2. Набор встроенных криптографических примитивов (хеш функции, симметричные и ассиметричные алгоритмы) позволяющих осуществлять быстрое прототипирование алгоритмов;
3. Использование библиотеки больших чисел, содержащей множество теоретико-числовых алгоритмов для операций с числами произвольной точности;
4. Слабая типизация, позволяющая преобразовывать содержание переменной в нужный программисту тип без вызова дополнительных функций;
5. Интерпретатор и все встроенные криптоалгоритмы должны быть реализованы на С или другом низкоуровневом языке, позволяющем добиться высокой производительности;
6. Си — подобный синтаксис языка, позволяющий быстро его изучить и начать применять на практике.

Задачи, которые должен будет решать данный язык:

1. Получение тестовых векторов для проверки работоспособности программ, реализуемых на других языках программирования;
2. Предоставление удобного набора инструментов для прототипирования алгоритмов.

Целевая аудитория, для которой разрабатывался язык - люди, изучающие криптографию либо тестирующие какие-либо криптографические алгоритмы.

**Список используемых терминов**

Лексема - последовательность допустимых символов языка программирования, имеющая определённый смысл для транслятора.

Токен – лексема, отнесённая к определённой категории

Регулярные выражения - формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов.

Область видимости — в программировании обозначает область в которой в данный момент можно обратиться по имени к переменным и функциям.

API (application programming interface) – набор функций и структур данных, предоставляемых приложением или библиотекой для использования во внешних продуктах.

Crypti – язык, разрабатываемый автором в данной дипломной работе

# Глава 1. Постановка задачи

Прототип программы — макет (черновая, пробная версия) программы, создаваемая обычно — с целью проверки пригодности предлагаемых для применения концепций, архитектурных и/или технологических решений.

Алгоритм по сути это последовательность операций производимых над входными данными для достижения определённого результата. Следует отметить что для криптографии часто используются их специфический набор (например нахождение обратного по модулю числа, возведение в степень по модулю, бинарное исключающее ИЛИ и так далее).

Проанализировав большинство популярных языков программирования было выявлено, что разработка и тестирование криптографических алгоритмов с их помощью вызывает проблемы ввиду их не оптимальности для решения таких задач.

Так же следует отметить что в официальных спецификациях криптографических алгоритмов приняты соглашения по поводу языка для описания алгоритмов [5].

Поэтому синтаксис языка близок к псевдокоду в спецификациях, и в то же время интуитивно понятен для большинства программистов.

В создаваемом языке основное внимание уделяется упрощению реализации алгоритмов шифрования с открытым ключом, так как реализация блочных шифров на языках низкого уровня не вызывает сильных затруднений.

Это не значит, что их невозможно реализовать на данном языке, просто в данный момент в языке нет необходимых для упрощения разработки блочных шифров функций (что, в прочем не мешает добавить их в будущем).

Предполагается, что созданный язык будет удобен для обучения криптографии, максимально быстрой и удобной разработки и генерации тестовых векторов.

Аналогичных разрабатываемых или готовых проектов найдено не было.

## 1.1. Описание разрабатываемого языка

В настоящее время написано много библиотек для поддержки шифрования и дешифровки различными методами. Таблица 1 содержит несколько примеров

Таблица 1. Примеры криптографических библиотек для распространённых языков программирования

|  |  |
| --- | --- |
| Язык программирования | Библиотеки |
| C/C++ | Openssl, cryptopp |
| Python | Pyopenssl, M2Crypto |
| Java | javax.crypto |
| С# | System.Security.Cryptography |

Часто для языков высокого уровня такая библиотека – обвёртка над C функциями библиотек наподобие openssl. И если перед программистом стоит задача просто создать программу шифрующую по определённым алгоритмом на одном из языков программирования, то целесообразней будет воспользоваться одной из доступных библиотек. Однако если задача – разобраться как работает тот или иной алгоритм и создать свой то, ни одна из библиотек не обеспечивает доступный инструментарий так как нужных функций в ней просто не содержится, что вынуждает программиста реализовывать свои функции на самом языке программирования без использования библиотек или пытатся интегрировать свой код в библиотеку. Что не вполне удобно ведь сами языки программирования разрабатывались для других целей. Конечно языки общего назначения могут использоваться для любых задач, и этому способствует широкий набор библиотек, но для прототипирования и экспериментов они подходят не в полной мере. Этим можно объяснить широкую популярность таких языков matlab и octave.

Одна из особенностей разрабатываемого языка – нативная поддержка чисел произвольной длины. Например сравним количество кода, которое потребуется для бинарного сдвига на С с использованием openssl:

SSL\_library\_init();

a = BN\_new();

r = BN\_new();

BN\_set\_word(a, 2);

BN\_lshift(r, a, 1024);

s = BN\_bn2dec(r);

printf("2 << 1024 = %s\n", s);

OPENSSL\_free(s);

BN\_free(a);

BN\_free(r);

И на crypti:

2<<1024

Конечно некоторые языки высокого уровня так же поддерживают длинные числа. Однако далеко не все поддерживают слабую типизацию. А это важно, так как при реализации алгоритмов часто приходится конвертировать значение из одного типа в другой (из октетной строки в число и наоборот). Например, рассмотрим процедуру шифрования RSA-OAEP-Encrypt[5], реализованную на разработанном языке.

def [C, error] RSAES\_OAEP\_Encrypt(n, e, M, P) {

 C = 0

 k = size(n)

 EM, error = EME\_OAEP\_Encode(M, P, k - 1)

 if (error) {

 error = "eme\_oaep\_encode " error

 return

 }

 C, error = RSAEP(n, e, EM)

 if (error) {

 error = "RSAEP " error

 return

 }

}

Согласно стандарту EME\_OAEP\_Encode при успешном выполнении возвращает октетную строку EM, которая должна быть преобразована в число и передана функции RSAEP. В нашем случае конвертация происходит неявно, что упрощает код и избавляет его от захламления лишними вызовами функций преобразования.

Идея использования функций, возвращающих множество аргументов, заимствована из языков octave и matlab. Это позволяет упростить чтение исходного кода и предоставляет возможность функции возвращать более чем один аргумент. Пример:

def [c, error] rsaep(m, exp, n) {

 if (m >= n) {

 error = "message representative out of range"

 c = 0

 return

 }

 c = mod\_exp(m, exp, n)

 error = ""

}

Ещё одной особенностью является поддержка одномерных ассоциативных массивов. Идея реализации приближена к массивам в awk[3]. За счёт таких массивов можно эмулировать любую структуру. Например:

Arr[“name”] = “Monty”

Arr[“surname”] = “Python”

Arr[“info”] = “spam”

Arr[“age”] = 42

Кроме того, как и в AWK можно эмулировать многомерные массивы разделяя индексы кавычками. Для составления итогового индекса составные элементы конкатенируются, разделяясь символом 034. Пример:

Arr[“hello”] = 42;

Arr[“hello”, Monty”] = “spam”

Arr[“Egg and”, “Bacon and”] = “sausage and Spam"

Так как С один из самых известных и широко распространённых языков программирования синтаксис разрабатываемого был максимально к нему приближен. Исключение составляет только определение функций и возможность параллельного присваивания. Примеры параллельного присваивания:

A, B = B, A

def [a,b] ret\_2\_args() {

 a = b = 4;

}

c, d = ret\_2\_args();

В языке отсутствует сборщик мусора. Хотя переменные внутри вложенных областей видимости удаляются сразу после выхода из этой области то, что определено в глобальной области видимости существует до завершения программы если программист явно не запросит её удаление. Это можно сделать с помощью ключевого слова del.

Для упрощения внутренней структуры интерпретатора и повышения скорости работы язык поддерживает только процедурную парадигму программирования, без каких либо признаков столь популярного сейчас объектно-ориентированного программирования.

# Глава 2. Описание общих принципов работы интерпретатора

## 2.1 Общие сведения

По способу выполнения языки можно разделить на интерпретируемые и компилируемые. В компилируемых языках исходный код преобразуется в машинные инструкции и сохраняется в отдельный исполняемый файл (Рисунок 1). В интерпретируемых языках исходный код выполняется сразу, без преобразования в исполняемые файлы (Рисунок 2).



Рисунок 1. Схема работы компилятора



Рисунок 2. Схема работы интерпретатора

Опишем подробнее работу интерпретатора. Перед тем как выполнить инструкции компилятор преобразует их в промежуточное представление, причём существует две стадии формирования этого представления:

* + 1. Анализ. Выражения разбиваются на составные части, проверяются на соответствие синтаксическим и семантическим правилам;
		2. Синтез. Непосредственно вычисление и исполнение выражения. Промежуточные результаты выводятся на экран или сохраняются в таблице символов.

Процесс интерпретации делится на несколько фаз:

* 1. Лексический анализ;
	2. Синтаксический анализ;
	3. Семантический анализ;
	4. Промежуточное представление;
	5. Выполнение.

Каждая фаза на выходе генерирует промежуточное представление, которое передаётся на следующий этап. Графически этот процесс изображён на рисунке 3.



Рисунок 3. Фазы интерпретации

Как видно из рисунка конечным промежуточным представлением является синтаксическое дерево. Оно формируется на фазе синтаксического анализа, и затем передаётся семантическому анализатору для семантической проверки созданного дерева[1].

## 2.2 Лексический анализ

Лексический анализатор считывает символы из входной строки и группирует их в «объекты токенов». Вместе с терминальными символами, которые используются в синтаксическом анализе, объекты токенов несут дополнительную информацию в форме значений атрибутов.

Иными словами лексический анализатор видит код программы как последовательность токенов. Для примера рассмотрим выражение:

foo = 42 + 23;

Лексический анализатор воспринимает эту строку как:

< ID, foo> <=> <42> <+> <23>

Лексема foo отображается в токен <ID, foo> где ID указывает на то что данный токен идентификатор.

Символ присваивания отразится в токен <=>. В данном случае значение атрибута токена не важно, и поэтому опускается.

Числа 42 и 23 отображаются в токены <42> и <23> соответственно.

Символ + отобразится в токен <+>

Имеется ряд причин, по которым фаза анализа разделяется на лексический и синтаксический анализ.

* + 1. Наиболее важной причиной является упрощение разработки. Отделение лексического анализатора от синтаксического часто позволяет упростить как минимум один из фаз анализа. Например, включить в синтаксический анализатор работу с комментариями и пробельными символами существенно сложнее, чем удалить их лексическим анализатором.
		2. Увеличивается эффективность интерпретатора. Отдельный лексический анализатор позволяет применять более специализированные методики, предназначенные исключительно для решения лексических задач.
		3. Увеличивается переносимость компилятора. Особенности входных устройств могут ограничивать возможности лексического анализатора.

Лексический анализатор реализован как функция, возвращающая по одному токену за раз, которую вызывает синтаксический анализатор. Таким образом, если лексический анализатор встретил лексему foo, то синтаксический анализатор получит токен TOK\_ID со значением атрибута установленным в строку “foo”.

Структура, описывающая токен, выглядит так:

struct lex\_item {

 tok\_t id;

 union {

 char \*name;

 struct variable \*var;

 };

};

## 2.2.1 Регулярные выражения

Регулярные выражения помогают определить токены, которые должен распознавать лексический анализатор. Для конструирования регулярных выражений будут использоваться следующие символы:

|  |  |
| --- | --- |
| Символ | Описание |
| + | указывает на один или более предшествующих элементов |
| ? | указывает на ноль или один предшествующий элемент |
| \* | указывает на ноль или более предшествующих элементов |

Теперь рассмотрим сами выражения:

* Пробельные символы лексическим анализатором пропускаются, для них не генерируются токены. Обозначим их как:

ws → (blank|tabs)

* Цифры

digit → [0-9]

* Числа

digits → digit+

* Английские буквы

letter → [A-Za-z\_]

* Идентификаторы

id → letter[letter | digit]\*

* Операции

assign\_op → =

arith\_op → +|-|\*|/|%

rel\_op→ < | > | >= | <= | == | !=

* Основные символы

basic\_sym → { | (| [ | ] | } | ) | . | , | ; | &

* Ключевые слова

while → while

del → del

**2.2.2 Ключевые слова**

Каждый язык программирования имеет набор ключевых слов, которые является таким же идентификатором id. Когда лексический анализатор находит идентификатор, он просматривает таблицу ключевых слов, и если находит совпадение, то возвращает соответствующий токен, в противном случае возвращается TOK\_ID с именем идентификатора в качестве значения атрибута.

Таким образом, каждое ключевое слово имеет свой токен. Например:

for → TOK\_FOR;

if → TOK\_IF;

def → TOK\_DEF;

**2.2.3 Таблица ключевых слов**

Для хранения данных об известных ключевых словах используется хеш массивы. Перед началом работы таблица инициализируется доступными ключевыми словами. Для внутреннего представления ключевых слов используется следующая структура, где id – соответствующий ключевому слову токен, name – строковое представление токена.

typedef struct {

 tok\_t id;

 char \*name;

} keyword\_table\_item\_t;

Поиск ключевых слов в таблице осуществляет функция keyword\_table\_lookup, ниже приведён её интерфейс:

int keyword\_table\_lookup(char \*name);

Функция возвращает TOK\_UNKNOWN в случае если токен с именем name не был найден. Упрощённо эта часть лексического анализатора представлена ниже:

if ((kword = keyword\_table\_lookup(s)) != TOK\_UNKNOWN) {

 lex\_item.id = kword;

 return kword;

}

lex\_item.id = TOK\_ID;

lex\_item.name = s;

return TOK\_ID;

**2.2.4 Удаление пробельных символов и комментариев**

Большинство языков программирования допускают произвольное количество пробельных символов между токенами, аналогично в процессе синтаксического анализа игнорируются комментарии, так что они также могут рассматриваться как пробельные символы.

Если пробельные символы удаляются лексическим анализатором, то синтаксический анализатор никогда не столкнётся с ними.

**2.3 Синтаксический анализ**

Схема взаимодействия лексического и синтаксического анализатора представлена на рисунке 4.



Рисунок 4. Взаимодействие лексического и синтаксического анализатора.

Синтаксический анализатор или парсер знает обо всех конструкциях языка. Парсер получает токены от лексического анализатора и преобразует их в однозначное дерево. То есть после работы парсера код программы отображается в древовидную структуру.

**2.3.1 Контекстно свободные грамматики**

Работа парсера построена вокруг правил. Данные правила описывают конструкции языка. То есть являются его формальным описанием. Для записи этих правил мы используем контекстно свободные грамматики.

Грамматики состоят из следующих компонентов.

* + - 1. Терминалов (или токенов);
			2. Нетерминалов (или синтаксических переменных);
			3. Продукций (правила разбора);
			4. Один из нетерминалов должен быть стартовым символом

Терминалами называются основные символы языка. Нетерминалы состоят из терминалов, которые организуются некоторым образом, что бы описать синтаксическую конструкцию. Продукции имеют следующую структуру: нетерминал с левой стороны продукции, называемый левой частью, и телом продукции с правой, представляющим из себя набор нетерминалов. Обе части разделяются знаком, в правой части можно использовать знак | (or), чтобы описать альтернативный вариант.

Рассмотрим простой пример. Создадим грамматику для описания выражения 1 –3 + 2

list → digit + list

list → digit – list

list → digit

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Каждая из этих грамматик является продукцией. С левой стороны находится заголовок, а справа тело продукции. List и digit являются нетерминалами. List является стартовым символом, потому что его продукция дана первой. Согласно этой грамматике символы + - 0 1 2 3 4 5 6 7 8 9 являются терминалами.

Нашей задачей является с помощью имеющейся грамматики «породить» приведённый выше пример. Грамматика порождает строку, начиная со стартового символа, повторно заменяя нетерминалы телами продукций этих нетерминалов. Это и есть задача синтаксического анализатора – выяснение для полученной строки терминалов способа её вывода из стартового символа грамматики. Если строка не может быть выведена из стартового символа, синтаксический анализатор должен сообщить об ошибке.

**2.3.2 Ассоциативность операций**

Следующий момент, о котором стоит упомянуть является ассоциативность операций. Ассоциативность возникает тогда, когда мы имеем операции одинакового приоритета и нам нужно решить в каком порядке вычислять выражение. Согласно правилам выражение 9 + 5 + 3 эквивалентно (9 + 5) + 3, аналогично 9 - 5 - 3 эквивалентно (9 - 5) - 3. Когда операнд подобно 5 имеет операторы с двух сторон, необходимо некоторое соглашение что бы решить какой оператор относится к данному операнду. Для человека данный момент очевиден. Но его следует сделать понятным также и для машины. Мы говорим что оператор + лево-ассоциативный, потому что операнд имеющий по обе стороны знак плюс, принадлежит оператору который находится слева.

Другим примером служат право-ассоциативные операторы. Например, операция присваивания. То есть выражение a = b = c эквивалентно a = (b = c).

**2.3.3 Приоритет операций**

Рассмотрим выражение 2 + 3 \* 4. Существуют две возможные интерпретации этого выражения: (2 + 3) \* 4 и 2 + (3 \* 4). Очевидно, необходим набор правил для разрешения этой неоднозначности.

Операция \* имеет больший приоритет, чем +, если она получает свои аргументы раньше. Таким образом, деление и умножение имеют более высокий приоритет, чем вычитание и сложение.

Рассмотрим грамматику для арифметических выражений. На данный момент мы имеем 4 арифметических операции, каждая из которых является лево-ассоциативной. Операции \* и / имеют приоритет выше, чем + и -. Так же в грамматике используются скобки, чтобы показать, что приоритет выражения в скобках выше.

expr → expr + ter m | expr - ter m | ter m

term → term \* factor | ter m / factor | factor

factor → digit | ( expr ) | E

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

C помощью символа E мы обозначаем пустую продукцию.

**2.3.4 Парсер рекурсивного спуска**

Данный вид парсера обрабатывает грамматику в один проход слева на право. Парсер получает токены от лексического анализатора, сканируя последовательно входную строку токен за токеном. Синтаксический анализатор сверяет текущий токен, с его правилами грамматики. Если находятся совпадения, то он двигается дальше, проверяя приходящие токены на соответствие грамматике. Это делается до тех пор пока не будет достигнут конец, либо найдена ошибка. Данный вид парсера также называется предикативным. Это название он получил в силу того, что следуя выбранному правилу грамматики, он предполагает что следующий приходящий токен, должен быть такой же, как и в выбранном правиле. То есть если текущий токен не входит в выбранное правило, это определенно ошибка пользователя.

Поясним всё сказанное на примере. Предположим, есть выражение

while ( i < 10) i += 1

Парсер последовательно получает токены от лексического анализатора. В данном случае первым токеном который будет получен от лексического анализатора будет TOK\_WHILE, Далее следуя правилам для конструкции while будет ожидаться токен TOK\_LPAR, на следующем шаге он будет проверять правило для expr, в данном случае это операция сравнения i < 10 которая разрешена грамматикой, далее таким же образом возвращается токен TOK\_RPAR и обрабатывается выражение i+= 1. Если бы на каком-то из этапов возникла ошибка, обработка была бы прервана и пользователю было возвращено сообщение об ошибке.

Описанная выше схема называется синтаксически управляемой трансляцией. Если взглянуть на исходный файл парсера, то можно заметить что функции, обрабатывающие выражение for имеют схожие имена с названиями продукций в грамматике, а так же одинаковый порядок вызова.

**2.3.5 Устранение левой рекурсии**

Для парсера рекурсивного спуска существует возможность зацикливания. Эта проблема возникает всвязи с так называемой левой рекурсией.

Взглянем на одну из продукций, описанных выше:

term → term \* factor | term

Проблема заключается в том, что в теле продукции имеется терминал с таким же именем что и в правой части. Так как элементы продукций отображаются в код это означает что первую функцию, которую должен вызвать парсер из функции term() является term(). Таким образом, парсер попадает в бесконечный цикл.

Для решения данной проблемы нужно переписать продукции данного вида. Для описания рассмотрим нетерминал A с двумя продукциями:

A → Aa | B

где a и B являются последовательность терминалов и нетерминалов, которые не начинаются с A. Продукции подобного вида называются лево-рекурсивными. Для устранения рекурсии перепишем продукцию таким образом:

A → BR

R→ aR | E

где нетерминал R и его продукция являются право-рекурсивными. Теперь рассмотрим пример вначале пункта:

A <-> term

a <-> \* factor

B <->factor

Используя данное правило можно переписать грамматику следующим образом:

term → factor rest\_term

rest\_term → \* factor | E

Где A это term, a это \* term, B является factor, R соответствует rest\_term.

Теперь перепишем определённые выше правила для операций +, -, \*, /

expr → term rest\_expr

rest\_expr → + term rest\_expr

 | - term rest\_expr

 | E

term → factor rest\_term

rest\_term → \* factor rest\_term

 | / factor rest\_term

 | E

factor→ digit | (expr)

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**2.3.6 Используемые правила грамматики.**

Ниже приведён список используемых в crypti правил грамматики.

program\_start → global\_expr

global\_expr → process\_function

 | process\_import

 | stmts

stmts → statement stmts | E

statement→ process\_scope

 |process\_if

 |process\_for

 |process\_while

 |process\_break

 |process\_continue

 |process\_return

 |process\_do

 |process\_del

 |expr

expr → assign

assign → logic\_or = assign

logic\_or→ logic\_and || logic\_or

logic\_and → bool\_or && logic\_and

bool\_or → bool\_xor | bool\_or

bool\_xor → bool\_and ^ bool\_xor

bool\_and → equity & bool\_and

equity→ rel\_op == equity

rel\_op→ shift\_expr > rel\_op

 | shift\_expr < rel\_op

shift\_expr→ add\_expr << shift\_expr

 | add\_expr >> shift\_expr

add\_expr→ mul\_expr + add\_expr

 | mul\_expr – add\_expr

mul\_expr→ id \* mul\_expr

 | id / mul\_expr

**2.3.7 Построение синтаксического дерева**

Задачей парсера является построение дерева. Главная функция program\_start возвращает корень дерева, листья которого являются терминалами, а внутренние узлы – нетерминалы либо NULL, если в процессе синтаксического анализа были обнаружены ошибки. Так же следует отметить, что дерево должно быть однозначным, т.е. для каждого выражения может быть только единственное представление в виде дерева.

Для примера рассмотрим, как строится узел для ключевого слова if

ast\_node\_t \*

ast\_node\_if\_new(ast\_node\_t \*\_if, ast\_node\_t \*body, ast\_node\_t \*\_else)

{

 ast\_node\_if\_t \*res;

 res = (ast\_node\_if\_t \*)

 ast\_node\_new(AST\_NODE\_IF, sizeof(\*res), ast\_node\_free);

 res->\_if = \_if;

 res->body = body;

 res->\_else = \_else;

 return AST\_NODE(res);

}

Функция ast\_node\_if\_new вызывается при синтаксическом разборе, в неё передаётся узел, отвечающий за условие, само тело условного оператора и опциональный узел для ключевого слова else.

**2.4 Обход дерева**

Последняя фаза работы интерпретатора – обход дерева созданного на фазе синтаксического анализа. Обход дерева с целью получения результата называется обходом в глубину. То есть, начиная с корня, мы спускаемся на максимальную глубину и обрабатываем листья, потом поднимаемся на предыдущий уровень.

Приведём пример:

static void traverse\_op\_as(ast\_node\_t \*tree)

{

 ast\_node\_op\_as\_t \*optree;

 eval\_t \*left, \*right, \*res;

 traverse(tree->left);

 traverse(tree->right);

 optree = (ast\_node\_op\_as\_t \*)tree;

 right = stack\_pop();

 left = stack\_pop();

 res = eval\_process\_op(left, right, optree->opcode);

 set\_value\_node(tree->left, res);

}

По данной функции видно, что сначала обрабатываются левый и правый дочерние узлы, а уже потом выполняется сама операция.

**2.4.1 Таблица символов**

Таблица символов в нашем случае служит для хранения информации о значениях переменных. Так как язык динамически типизированный, тип переменной определяется на момент присваивания ей значения.

Для каждой области видимости поддерживается своя таблица символов, то есть переме нные из текущей таблицы «перекрывают» переменные из старых таблиц. Таблица создаётся и удаляется при обходе созданного дерева.

Каждая программа имеет по крайней мере одну таблицу – для глобальной области видимости.

Все таблицы держатся в односвязном списке, где каждый элемент ссылается на предыдущий, и последним элементом является глобальная таблица (пример на рисунке 5).



Рисунок 5. Схема хранения таблиц символов

Следующая функция показывает пример работы с областями видимости

void traverse\_scope(ast\_node\_t \*tree)

{

 ast\_node\_t \*next;

 res\_type\_t res;

 struct hash\_table \*idtable;

 idtable = id\_table\_create();

 id\_table\_push(idtable);

 next = tree->child;

 while (next != NULL) {

 res = traverse\_body(next);

 switch (res) {

 case RES\_ERROR:

 case RES\_CONTINUE:

 case RES\_BREAK:

 case RES\_RETURN:

 goto finalize;

 default:

 next = next->child;

 continue;

 }

 }

finalize:

 id\_table\_pop();

 id\_table\_free(idtable);

}

Как мы видим перед началом выполнения цикла создаётся новая область видимости (функция id\_table\_create) и добавляется в конец списка уже существующих (функция id\_table\_push). Все последующие переменные будут созданы в новой области видимости. При завершении обработки инструкций внутри области видимости функция удаляет эту область видимости и очищает в ней все переменные.

Элементы таблицы символов описываются следующей структурой.

typedef struct id\_item {

 id\_type\_t type;

 char \*name;

 union {

 struct variable \*var;

 arr\_t \*arr;

 };

} id\_item\_t;

Где type – тип данной переменной (массив или единичное значение), name строка, содержащяя имя переменной, var, arr значение переменной (выбирается в зависиомости от type)

Поиск символов осуществляется с помощью функций

id\_item\_t \*id\_table\_lookup(char \*name);

id\_item\_t \*id\_table\_lookup\_in(struct hash\_table \*table, char \*name);

id\_item\_t \*id\_table\_lookup\_all(char \*name);

Первая функция ищет символ в текущей таблице, вторая в указанной, а третья во всех доступных.

**2.4.2 Таблица функций**

Так же как переменные, функции хранятся в своей таблице. Существуют 2 типа функций – библиотечные и пользовательские. Основное отличие первых от вторых в том, что их обработчиком является код, написанный на C и их нельзя переопределить. Структура, описывающая информацию о функции, представлена ниже:

typedef struct {

 char \*name;

 char \*\*args;

 int nargs;

 char \*\*retargs;

 int nret;

 int is\_lib;

 union {

 libcall\_handler\_t handler;

 void \*body;

 };

} func\_t;

args, nargs – хранят имена аргументов и их количество.

retargs, nret – имена и количество возвращаемых аргументов

is\_lib – флаг, хранящий информацию о том библиотечная ли это функция

handler, body – непосредственно обработчики функций.

Интерфейс функций для добавления, удаления и поиска элементов очень похож на интерфейс функций таблицы символов, поэтому описан не будет.

**2.5 Слабая типизация**

Одним из важных свойств разрабатываемого языка является слабая типизация. То есть значение переменной может принимать разные типы в зависимости от места применения. Например, в выражении

2 + ''0x12''

Результатом будет 0x14 (или 20 в десятичной системе) так как строка ''0x12'' в данном выражении воспринимается как число 0x12.

Все переменные и литералы любого типа представлены внутри интерпретатора в качестве одной структуры variable, которая содержит в себе структуры, отвечающие за представление переменной в каждом из типов

Программно слабая типизация реализуется так:

typedef enum {

 VAR\_BIGNUM = 0x1,

 VAR\_OCTSTRING = 0x2,

 VAR\_STRING = 0x4,

} var\_type\_t;

struct variable {

 var\_type\_t type;

 mpl\_int bnum;

 octstr\_t octstr;

 str\_t str;

};

Флаг *type* в структуре содержит информацию об актуальных на настоящий момент типах. При изменении значения флаг сбрасывается, а при конвертации в другой тип к флагу с помощью бинарного ИЛИ добавляется идентификатор этого типа. Отличительной особенностью этого подхода является то, что при многочисленном использовании значения переменной конвертация в необходимый тип будет происходить только в первый раз.

Сама конвертация выполняется с помощью функций

str\_t \*var\_cast\_to\_str(struct variable \*var);

octstr\_t \*var\_cast\_to\_octstr(struct variable \*var);

mpl\_int \*var\_cast\_to\_bignum(struct variable \*var);

Если флаг с необходимым типом уже установлен, то возвращается указатель на запрашиваемую структуру. Если же флага нет то конвертация внутри функций происходит посредством вызова функции с таким интерфейсом:

void \*convert\_value(struct variable \*dst\_var, int to\_type, struct variable \*src\_var, int from\_type);

Эта функция просматривает таблицу структур

struct type\_conv {

 int from\_type;

 int to\_type;

 type\_converter\_t func;

};

И находит необходимую для конвертации из from\_type в to\_type функцию.

Если же у структуры variable в флаге type уже установлен более чем один флаг, возникает некоторая неоднозначность из какого же типа нужно конвертировать. Таблица ниже описывает правила приоритетности при конвертации.

Таблица 4. Приоритет конвертации.

|  |
| --- |
| Приоритет конвертации (от высшего к низшему) |
| Bignum |
| Octstring |
| String |

Для операций над переменными используется единое API, содержащее в себе функции для работы с числами, строками и октетными строками, так что для клиентского кода работа со struct variable выглядит просто и лаконично, так как отпадает необходимость проверки или самостоятельной конвертации из одного типа в другой.

**2.6 Библиотека больших чисел**

В качестве библиотеки больших чисел использовалась модификация библиотеки libmpl[4]. Libmpl написана на С и содержит в себе реализацию типа mpl\_int, позволяющего хранить числа с произвольной точностью, так же в библиотеке содержатся функции, позволяющие производить элементарные арифметические и битовые операции над mpl\_int, а так же взятие модуля, нахождение наибольшего общего делителя и другие. В коде интерпретатора в основном эти функции используются при обработке арифметических операторов на стадии обхода построенного дерева. Пример использования:

int varop\_div(struct variable \*c, struct variable \*a, struct variable \*b)

{

 mpl\_int \*ap, \*bp, \*cp;

 ap = var\_cast\_to\_bignum(a);

 bp = var\_cast\_to\_bignum(b);

 cp = var\_bignum\_ptr(c);

 mpl\_div(cp, NULL, ap, bp);

 var\_force\_type(c, VAR\_BIGNUM);

 return 0;

}

Данная функция была упрощена в целях экономии места. varop\_div преобразует, переданные аргументы в тип mpl\_int и производит вычисление частного a и b.

**2.7 Хеш функции**

Хеш-функции в интерпретаторе реализованы в двух вариантах: полном и укороченном.

Интерфейс укороченной реализации выглядит так:

def [digest] hash\_name(str)

Функция принимает на вход строку и возвращает вычисленный хеш.

Можно было бы ограничиться лишь этим вариантом, однако при написании алгоритмов может возникнуть ситуация что программисту нужно считать очень много данных (гигабайты) либо данные поступают порционно и было бы удобнее вычислять хеш таким же образом, для этого предназначены функции с более сложным интерфейсом.

def [ret] hash\_name\_init(id\_str)

def [] hash\_name\_update(id\_str, new\_data)

def [digest] hash\_name\_finalize(id\_str)

При порционном вычислении хеша необходимо привязать контекст хеша к идентификатору. Для этого функция hash\_name\_init заводит новую структуру в таблице хешей и связывает её с переданной строкой и возвращает 0 либо 1, в случае если данное имя уже используется или превышено максимальное количество используемых контекстов.

hash\_name\_update обрабатывает новую порцию данных для хеша, ассоциированного с id\_str;

hash\_name\_finalize возвращает полученный хеш.

## 2.8 Ассоциативные массивы

Языком поддерживаются ассоциативные массивы. Идея нативной поддержки ассоциативных массивов заимствована из языка AWK[6]. Следует помнить что такие массивы не обязательно должны быть сплошными и каждый новый элемент появляется только в процессе присваивания. В качестве внутренней реализации используется хеш таблица. Сама библиотека массивов не поддерживает составные ключи, вызывающий код (в нашем случае код обхода дерева) сам задаёт политику. При получении составного индекса производится конкатенация строк. Интерфейс реализующей эту логику функции представлен ниже

static char \*create\_index(ast\_node\_t \*\*ind, int dims)

где ind массив размера dims с индексами.

 В свою очередь библиотека массивов поддерживает простой интерфейс.

arr\_t \*arr\_new()

void arr\_free(arr\_t \*arr)

void arr\_set\_item(arr\_t \*arr, char \*key, struct variable \*var)

struct variable \*arr\_get\_item(arr\_t \*arr, char \*key)

ret\_t arr\_remove\_item(arr\_t \*arr, char \*key)

Как очевидно из названия первые 2 функции отвечают за создание и удаление массива, а остальные 3 за присвоение взятие и удаление значений.

## 2.9 Пользовательские Функции

Пользователь crypti может объявлять свои функции. Синтаксис объявления функций несколько похож на объявление функций в python[7] (ключевое слово def) и octave (список возвращаемых аргументов):

def [ret\_values\_list] function\_name(arg\_list) {
body

}.

где:
ret\_value\_list – перечисленный через запятую список возвращаемых значений.

arg\_list – список аргументов функции.

body – тело функции.

Как было сказано выше после определения все функции хранятся в специальной таблице. Информация о функции включает в себя список аргументов функции, имена возвращаемых значений и дерево, являющееся представлением тела функции. Следует помнить что для успешного завершения необходимо чтобы при выходе из функции были определены все значения возвращаемых аргументов, иначе возникнет ошибка во время выполнения.

**Глава 3. Практическое применение языка для подготовки специалистов**

Созданный язык можно применять для подготовки специалистов по защите информации. Во время обучения основам криптографии студент сталкивается с рядом сложностей. Основные из них – то, что довольно сложно воспринимать сухую теорию без практических примеров, а так же невозможность сразу тестировать данные преподавателем алгоритмы. Созданный язык должен в этом помочь, так как в него встроено большинство необходимых функций, а так же существует возможность работы в интерактивном режиме (обрабатывается каждая новая строка, поступившая на стандартный ввод, а результат печатается на стандартный вывод). Поэтому реализация алгоритмов на нём проста и не требует много времени. Так же язык можно использовать при индивидуальной подготовке. Была разработана лабораторная работа (с самой лабораторной работой можно ознакомиться в приложении).

В качестве задания в лабораторной работе предлагается реализовать и протестировать работоспособность нескольких алгоритмов на языке crypti. А именно шифрование по RSA, схема шифрования и электронной подписи по алгоритму Эль-Гамаля. Перед реализацией каждого алгоритма студент может ознакомиться с теоретической выкладкой. Для шифрования по RSA и Эль-Гамалю приводятся возможные интерфейсы функций шифрования и дешифрования, даются подсказки по реализации. Реализация ЭЦП по Эль-Гамалю оставляется на самостоятельное изучение. Для каждого алгоритма приводится ссылка на вариант реализации (может использоваться преподавателем для проверки правильности выполнения лабораторной работы). Выбор именно этих алгоритмов для реализации в лабораторной работе обусловлен их широкой распространённостью и простотой реализации. Цель данной лабораторной работы - научить применять теоретические знания в области криптографии для составления алгоритмов и программ, реализующих функции шифрования и дешифрования исходного текста с помощью различных криптографических алгоритмов.

**Глава 4. Тестирование программы**

Тестирование работы интерпретатора было разбито на несколько этапов:

1. Тестирование работы операторов, приоритетности и ассоциативности операторов
2. Поиск и устранение утечек памяти
3. Реализация и тестирование алгоритмов.

Тестирование первых двух пунктов можно проделать, введя команду make test в корне проекта. С помощью неё собираются все необходимые бинарные файлы и запускаются скрипты проверки.

**4.1 Тестирование работы операторов, приоритетности и ассоциативности операторов**

Тестирование примитивов языка выполняется с помощью скрипта test/crypti/calc\_test.sh. По сути, он по очереди передаёт интерпретатору файлы с исходным кодом и сравнивает вывод с ожидаемым. Сейчас написано 10 исходных файла для тестирования проверяющих корректную работу операторов, их ассоциативности, работу некоторых встроенных функций (включая функции форматированного вывода и функции хеширования).

**4.2 Поиск и устранение утечек памяти**

Так как языком, на котором реализовывался интерпретатор, был Си, возникла необходимость проверки работы программы на утечки памяти.

Утечка памяти - определяемая как сбой при освобождении ранее выделенной памяти, — это одна из наиболее трудно обнаруживаемых ошибок в приложениях C/C++. Небольшая утечка памяти сначала может остаться незамеченной, но постепенно нарастающая утечка памяти может приводить к различным симптомам, от снижения производительности до аварийного завершения приложения из-за нехватки памяти.

Для поиска и устранения утечек использовался valgrind. Для автоматизации тестирования написан простой shell скрипт. Основная его часть приведена ниже.

for ((i=1; $i<$N + 1; i=$i+1))

do

 cat $path/test\_$i.txt | valgrind $path/../../$PNAME 1>$out 2>$temp\_file

 cat $temp\_file | grep "in use at exit"

 cat $temp\_file | grep "total heap usage:"

done

где test$i.txt имя файла содержащего инструкции для интерпретатора.

Вывод valgrind фильтруется и пользователь, запустивший скрипт, увидит число вызовов malloc и free а так же количество не освобождённой памяти, доступной после завершения работы интерпретатора.

При обнаружении не освобождённой памяти можно отдельно проанализировать с помощью valgrind вызвавший ошибку скрипт с помощью опций --leak-check=full --show-reachable=yes, определяющими в каком конкретно месте происходит утечка памяти.

На текущий момент тесты не обеспечивают стопроцентного покрытия кода, однако можно с уверенностью сказать, что большая часть ошибок была отловлена и устранена с помощью этого метода.

**4.3 Реализация алгоритмов**

Краткое описание тестируемых алгоритмов, и приведены исходные коды на crypti с результатами приведены в приложении A.

На текущий момент реализованы и протестированы:

1. RSA
2. RSA-OAEP
3. Алгоритм Диффи-Хеллмана
4. Схема Эль-Гамаля

**Заключение**

Цель данной работы была достигнута, а именно:

* + 1. Были изучены и применены на практике принципы теории компиляторов для создания интерпретатора;
		2. Для интерпретатора был разработан набор встроенных функций, позволяющих упростить разработку и тестирование криптоалгоритмов. А именно:

2.1 Криптографические хеш функции (md5, sha1, sha256, whirpool);

2.2 Функции работы с печатными и октетными строками;

2.3Полезные для целей криптографии функции (нахождение обратного числа по модулю, возведение в степень по модулю, нахождение наибольшего общего делителя);

2.4 Функции форматированного вывода (print, printf)

3. Был написан набор скриптов для проверки правильности вычислений интерпретатора и отлавливания утечек памяти

4. Была написана и протестирована небольшая библиотека для шифрования и дешифровки по методу RSA-OEAP

Исходный код и описание языка доступны на сайте github.com (<https://github.com/dzruyk/crypti.git>) и в приложении на компакт диске.

**Используемая литература**

* 1. «Компиляторы: Принципы, технологии и инструментарий» / А.В. Ахо, М.С. Лам, Р. Сети, Д.Д. Ульман – издательский дом «Вильямс» 2001.
	2. A.J. Menezes Handbook of applied cryptography / A.J. Menezes, P.C. van Oorschot, S.A. Vanstone - CRC Press 2006.
	3. A.V. Aho The AWK programming language / A.V. Aho, B.W. Kernighan, P.J. Weinberger – AT&T Bell Laboratories 1988.
	4. B. Schneier Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C – Willey Computer Publishing 1996.
	5. RSAES-OAEP Encryption Scheme Algorithm. Specification and supporting documentation – RSA Laboratories
	6. Gawk - GNU Project - Free Software Foundation (FSF) [Электронный ресурс] – Режим доступа: <http://www.gnu.org/software/gawk/> [Дата обращения 14.05.2013]
	7. Python Programming Language - Official Website. – Режим доступа: <http://www.python.org/> [Дата обращения 14.05.2013]
	8. Lua [Электронный ресурс] – Режим доступа: <http://www.lua.ru/> [Дата обращения 14.05.2013]
	9. iPetrucho/bclite [Электронный ресурс] – Режим доступа: <https://github.com/iPetrucho/bclite> [Дата обращения 14.05.2013]
	10. LitePAC [Электронный ресурс] – Режим доступа: <http://www.litepac.org/> [Дата обращения 14.05.2013]

**Приложение А. Реализация алгоритмов на crypti**

**1 RSA**

Описание:

Для создания ключей шифрования выбираются два больших простых числа p и q одинаковой длинны. Вычисляется $n=pq$.

Затем выбирается e взаимно простое с $(p-1)(q-1)$.

И вычисляется d: $ed=1 mod\left(p-1\right)(q-1)$

Публичным ключом является пара (e, n)

Закрытым ключом является d.

Для шифрования сообщения m необходимо:

$$c=m^{e} mod n$$

Для расшифрования:

$$m^{'}=c^{d}mod n$$

Исходный код на crypti:

def [c, error] RSAEP(n, exp, m) {

 if (m >= n) {

 error = "message representative out of range"

 c = 0

 return

 }

 c = mod\_exp(m, exp, n)

 error = ""

}

def [m, error] RSADP(n, d, c) {

 if (c >= n) {

 error = "message representative out of range"

 m = 0

 return;

 }

 m = mod\_exp(c, d, n);

 error = ""

}

n = p \* q

fi = (p - 1) \* (q - 1)

e = 3

d = mod\_inv(e, fi)

m = `RSA`

printf("Message before encryption %s\n", m "")

c, e = RSAEP(n, e, m)

m1, e = RSADP(n, d, c)

printf("Encrypted string is %s\n", m1 | 0x00);

Тестируемые значения:

p = 3557

q = 2579

Вывод программы:

Message before encryption RSA

Encrypted string is RSA

Строки до и после шифрования идентичны, что подтверждает правильность работы алгоритма.

**2 Алгоритм Диффи – Хеллмана**

Описание:

Алгоритм позволяет двум или более пользователям обменяться без посредников ключом, который может быть использован затем для симметричного шифрования.

Стороны договариваются об открытых числах: большом простом числе p и g, таком что 0 < g < p.

Для создания общего ключа выполняются следующие действия:

Первая сторона выбирает случайное число a и посылает второй стороне A:

$$A=g^{a}mod p$$

Вторая сторона вычисляет B и посылает первой:

$$B=g^{b}mod p$$

Затем первая сторона вычисляет

$$K1=B^{a}mod p$$

А вторая сторона:

$$K2=A^{b}mod p$$

Получившиеся K1 и K2 сравнимы с $g^{ab}mod p$.

Исходный код на crypti:

def [E] Alice(p, g, key) {

 E = mod\_exp(g, key, p)

}

def [E] Bob(p, g, key) {

 E = mod\_exp(g, key, p)

}

A = Alice(p, g, a)

B = Bob(p, g, b)

K1 = Alice(p, B, a)

K2 = Bob(p, A, b)

if (K1 == K2) {

 printf("[+] Diffie - Hellman key exchange ok\n");

 printf("[+] Result key is %x\n", K1);

} else {

 printf("[-] Diffie - Hellman key excange fail\n");

 printf("[-] K1 = %d, K2 = %d\n", K1, K2);

}

Тестируемые значения:

p=0x8b79f180cbd3f282de92e8b8f2d092674ffda61f01ed961f8ef04a1b7a3709ff748c2abf6226cf0c4538e48838193da456e92ee530ef7aa703e741585e475b26cd64fa97819181cef27de2449cd385c49c9b030f89873b5b7eaf063a788f00db3cb670c73846bc4f76af062d672bde8f29806b81548411ab48b99aebfd9c2d09

g=0x029843c81d0ea285c41a49b1a2f8e11a56a4b39040dfbc5ec040150c16f72f874152f9c44c659d86f7717b2425b62597e9a453b13da327a31cde2cced600915252d30262d1e54f4f864ace0e484f98abdbb37ebb0ba4106af5f0935b744677fa2f7f3826dcef3a1586956105ebea805d871f34c46c25bc30fc66b2db26cb0a93

a = 0x87e0beefd8122561e9c329d764c6e3b3dafe538a

b=0x4fc9904887ac7fabff87f054003547c2d9458c1f6f584c140d7271f8b266bb390af7e3f625a629bec9c6a057a4cbe1a556d5e3eb2ff1c6ff677a08b0c7c509110b9e7c6dbc961ca4360362d3dbcffc5bf2bb7207e0a5922f77cf5464b316aa49fb62b338ebcdb30bf573d07b663bb7777b69d6317df0a4f636ba3d9acbf9e8ac

Вывод программы:

[+] Diffie - Hellman key exchange ok

[+] Result key is \x30\x5E\x29\x02\x15\x5D\x47\xF1\xCC\x34\x1C\x28\x79\xDF\x35\x85\x9C\x23\xD8\x12\x20\x89\x9D\x50\xB6\x94\x1B\x4F\xBD\xAE\x58\x50\x95\x1F\x4E\x30\x2E\x6E\xC3\xF7\x7E\x28\x99\x07\x46\x0B\x17\x4D\x2D\xE0\x77\x5F\x30\x75\xC2\x45\x03\x40\xB6\x99\x85\x2F\xBD\x1D\x11\xC5\x4A\x81\xF7\xA9\x73\xF9\x30\xBA\x4C\x82\x6C\x28\xA0\x2F\xF2\xA0\x38\x74\x18\xC4\x0F\xE4\xE5\x31\xBC\xDE\x31\xF1\xC3\x51\x50\x70\xBC\xCB\x6A\x55\xC4\x07\x73\x63\xD6\x82\x48\x50\xD2\xC4\x68\x7C\x21\xDD\x2C\xAA\xA7\xEA\xC3\x0D\x83\xAF\x67\x41\x7B\x6C

Строки до и после шифрования идентичны, что подтверждает правильность работы алгоритма

**3 Схема Эль-Гамаля**

Описание:

Пусть имеются абоненты A, B, … которые хотят передавать друг другу зашифрованные сообщения, не имея никаких защищённых каналов связи.

Для всей группы абонентов выбираются некоторое большое простое число p и число alpha, a такое что 0 < alpha < p.

Затем каждый абонент выбирает а такое что 0 < a < p - 1.

После чего вычисляется $h= alpha^{a}$

Публичным ключом является тройка (p, alpha, h)

Секретным ключом абонента – a.

Для шифрования сообщения отправитель выбирает случайное число k такое, что 0 < k < p – 1. Затем вычисляет

$$c1=alpha^{k} mod p$$

$$c2= h^{k}m mod p$$

c1, c2 – шифротекст, для расшифровки необходимо вычислить[2]:

$$m=\frac{c2}{c1^{x}mod}p$$

Исходный код на crypti:

//h is alpha \*\* a

def [c1, c2] elgamal\_enc(p, alph, h, m) {

 k = rand(1, p - 2)

 c1 = mod\_exp(alph, k, p)

 c2 = (m \* mod\_exp(h, k, p)) % p

}

def [m] elgamal\_dec(p, a, c1, c2) {

 tmp = mod\_exp(c1, p - 1 - a, p)

 m = (tmp \* c2) % p

}

encrypted = `top secret`

printf("before encryption: %s\n", encrypted);

c1, c2 = elgamal\_enc(p, alpha, h, encrypted)

m = elgamal\_dec(p, a, c1, c2)

printf("after decryption message: %s\n", m | 0x00);

Тестируемые значения:

p=0xF1B18AE9F7B4E08FDA9A04832F4E919D89462FD31BF12F92791A93519F75076D6CE3942689CDFF2F344CAFF0F82D01864F69F3AECF566C774CBACF728B81A227

alpha = 0x07

a=0x14E60B1BDFD33436C0DA8A22FDC14A2CCDBBED0627CE68

Вывод программы:

before encryption: top secret

after decryption message: top secret

Строки до и после шифрования идентичны, что подтверждает правильность работы алгоритма.