Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт цифровых технологий, электроники и физики (ИЦТЭФ)

Кафедра вычислительной техники и электроники (ВТиЭ)

Отчет по производственной практике

(Научно-исследовательская работа)

РАЗРАБОТКА ЯЗЫКА ПРОГРАММИРОВАНИЯ:

ИНТЕРПРЕТАТОР

Выполнил студент 506 группы:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.В. Осипенко

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2024 г.

Проверил: ст. преп. кафедры ВТиЭ.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ П.Н. Уланов

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2024 г.

РЕФЕРАТ

Полный объем работы составляет 17 страниц, включая 3 рисунка.

В рамках научно-исследовательской работы было произведено исследование устройства интерпретатора для языка программирования, разработан собственный интерпретатор для простого демонстрационного языка программирования и проведено тестирование его работоспособности на примере простой программы на данном языке программирования.

Цель работы – разработать интерпретатор для простого статически типизированного языка программирования.

В результате выполнения научно-исследовательской работы был разработан и протестирован интерпретатор для простого языка программирования.

Ключевые слова: Go, рекурсивный интерпретатор, АСД, EBNF, программное обеспечение.

СОДЕРЖАНИЕ

[Введение 4](#_539pho8uv2vx)

[1. Устройство интерпретатора и принцип его работы 5](#_sele8pq79mde)

[2. Разработка интерпретатора 6](#_elvz5brpcaw)

[3. Тестирование интерпретатора 12](#_24mowbiwrp2p)

[Заключение 16](#_diutxh2gbnyc)

[Список использованной литературы 17](#_wgjbpd8v0rn5)

# Введение

На сегодняшний день любая программа написана с помощью какого-либо языка программирования (далее ЯП), от самого низкоуровневого, например ассемблер, до высокоуровневых представителей, таких как: Python, Ruby, Go, Rust, C/C++ и т.д.. Сам по себе ЯП представляет собой [1-3]:

● Спецификацию;

● Программу компилятор или интерпретатор.

Компилятор (или интерпретатор) являются главным элементом любого ЯП, определяющее основные показатели языка, как: быстродействие, поддерживаемые платформы и функциональные возможности. В следствии чего это направление можно назвать одним из основополагающих и до сих пор актуальных [1-3].

В данной научно-исследовательской работе (далее НИР) объектом исследования является интерпретатор для языка программирования.

Цель работы – разработать интерпретатор для простого статически типизированного языка программирования..

Задачи НИР:

1. Изучить базовое устройство интерпретатора и принципы его работы;

2. Разработать интерпретатор для простого статически типизированного ЯП;

3. Протестировать работоспособность созданного интерпретатора.

# Устройство интерпретатора и принцип его работы

**Интерпретатор** – это программа, которая принимает на вход элементы абстрактного синтаксического дерева (далее АСД) в определенном (корректном) порядке и и выполняет предопределенные действия для соответствующих элементов АСД основываясь на семантике ЯП. Из данного определения можно заметить, что в отличии от компилятора, интерпретатор принимает входные данные, необходимые во время выполнения программы, например чтение данных через консоль или файл. Также видно, что интерпретатор, по своим действиям, напоминает принцип работы ЦПУ компьютера, за исключением того, что он работает с элементами АСД, а не машинными инструкциями [1]. Представление интерпретатора приведено на рисунке 1.1.

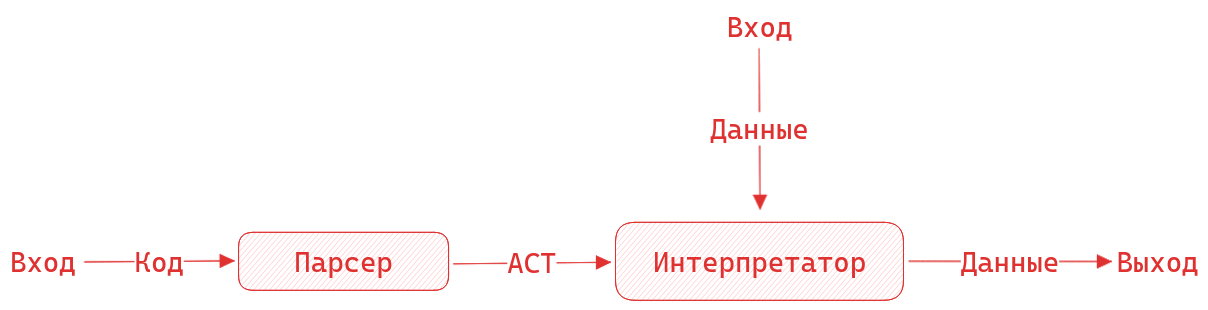


Рис. 1.1 Абстрактное представление интерпретатора

По своей структуре, интерпретатор представляет из себя совокупность функций, каждая из которых относится к определенной категории элементов АСД. Эти функции будут принимать в качестве аргументов АСД и различные вспомогательные параметры, такие как состояние интерпретатора и символьную таблицу. В качестве возвращаемых значений могут выступать результаты вычисления выражений и обновленные состояние интерпретатора и символьные таблицы [2].

Для хранения значений в интерпретаторе используются символьные таблицы, которые представляют из себя словари, соотносящие некоторое уникальное имя (переменной, функции, константы, …) с некоторым значением (строка, число, булево значение, функция, …). Данные таблицы обычно делят на определенные контексты, в рамках которых они используются: глобальные (контекст программы, модуля), локальные (контекст функции, цикла, ветвления), окружение объекта (контекст некоторой структуры, объекта). В интерпретаторе мы имеем один глобальный контекст и множество локальных. Каждый вызов функции создает новый локальный контекст, который содержит в себе входные параметры и локальные переменные. Интерпретатор отслеживает данный контекст вплоть до конца выполнения функции, после чего возвращается обратно к глобальному или предшествующему локальному контексту предыдущей функции [3].

# Разработка интерпретатора

В качестве ЯП, для которого будет разрабатываться интерпретатор, будет использоваться простой статически типизированный язык, грамматика которого представлена в листинге 2.1. Данный язык имеет 6 предопределенных типов и 1 пользовательский, позволяет создавать переменные, константы, функции, записи (пользовательский тип) и модули. В качестве конструкций управления потоком выполнения представлены цикл while-else и ветвление if-elif-else.

**Листинг 2.1:** Синтаксис и грамматика ЯП.

Module = "(" "@module" (":main" | ":code") Global+ ")" .

Global = ConstantDecl | FunctionDecl | RecordDefn | ImportDecl .

ConstantDecl = "(" "@const" <symbol> Atom ")" .

Atom = IntAtom | BoolAtom | FloatAtom | StringAtom .

IntAtom = <integer> .

BoolAtom = <boolean> .

FloatAtom = <float> .

StringAtom = <string> .

FunctionDecl = "(" "@fn" <symbol> ("[" VariableDefn+ "]")? ("<" Type ">")? ":do" Local+ ")" .

VariableDefn = <symbol> "::" Type .

Type = IntType | BoolType | FloatType | StringType | ListType | RecordType .

IntType = "int" .

BoolType = "bool" .

FloatType = "float" .

StringType = "string" .

ListType = "list" "<" Type ">" .

RecordType = SymbolExpr .

SymbolExpr = <symbol> ("/" SymbolExpr)? .

Local = ConstantDecl | VariableDecl | ReturnStmt | SetStmt | SymbolCall | IfStmt | WhileStmt .

VariableDecl = "(" "@var" <symbol> ":of" Type Expression ")" .

Expression = IntAtom | BoolAtom | FloatAtom | StringAtom | SymbolCall | NewExpr | SymbolExpr .

SymbolCall = "(" SymbolExpr Expression\* ")" .

NewExpr = "(" "@new" Type "{" Expression\* "}" ")" .

ReturnStmt = "(" "@return" Expression? ")" .

SetStmt = "(" "@set" SymbolExpr Expression ")" .

IfStmt = "(" "@if" Expression ":then" Local+ Elif\* (":else" Local+)? ")" .

Elif = ":elif" Expression "=>" Local+ .

WhileStmt = "(" "@while" Expression ":do" Local+ (":else" Local+)? ")" .

RecordDefn = "(" "@record" <symbol> ":fields" VariableDefn+ ")" .

ImportDecl = "(" "@import" StringAtom ":as" <symbol> ")" .

Для разработки интерпретатора был выбран язык **Go** [4], разработанный компанией Google. Данный выбор обусловлен простотой языковых конструкций и в тоже время богатыми функциональными возможностями, быстрой и кроссплатформенной компиляции, большим сообществом разработчиков и личными предпочтениями.

Лексический и синтаксический анализаторы были созданы с помощью сторонней библиотеки **participle** [5]. Данная библиотека позволяет быстро и без каких-либо трудностей создать парсер. Единственное что необходимо было сделать, это создать элементы АСД и их связи, все остальное (разбор и анализ входного кода) библиотека выполнит сама.

Перед созданием интерпретатора необходимо определить основные сущности времени выполнения и связи между ними. Основной единицей является интерфейс **Type**, которая используется для представления типа данных внутри интерпретатора. С помощью данного интерфейса можно получить доступ к метаданным хранимого в ней типа, например получить его наименование, тип элемента (для списков), тип входного параметра функции или ее результата выполнения. Определение данного интерфейса приведено в листинге 2.2.

**Листинг 2.2:** Интерфейс Type.

type Type interface {

Kind() Kind // вид типа

Name() string // название типа

String() string // строковое представления типа

Compare(Type) bool // сравнение типа с другим типом

Item() Type // получить тип элементов списка

NumIns() int // количество входов функции

In(int) Type // тип определенного входа функции

Out() Type // тип выхода функции

NumFields() int // количество полей записи

Field(string) (FieldType, int) // получить поле записи и его номер с помощью его названия

FieldByIndex(int) FieldType // получить поле записи с помощью его номера

}

Следующей сущностью является **Litteral**, которая является представлением базового объекта (цифры, строки, булева значения). Данный интерфейс напрямую зависим от Type, позволяя тем самым производить различные проверки в рамках семантики языка, например соответствия объекта с типом выхода функции, и корректности его внутренней структуры, в частности однотипность элементов списка. Данный интерфейс позволяет получить необходимые метаданные об объекте и взаимодействовать с ним. Его определение приведено в листинге 2.3.

Для представления различных символьных сущностей, таких как переменная, константа, функция используется **Symbol**. Данный интерфейс зависим от двух предыдущих, так как его объекты, например переменная, могут содержать в себе тип и литерал. Основное применение он находит в символьных таблицах, где выступает как сущность с соответствующим ему уникальным ключом (наименованием символа). Определение интерфейса приведено в листинге 2.4.

**Листинг 2.3:** Интерфейс Litteral.

type Litteral interface {

Kind() Kind // вид литерала

String() string // строковое представление

Type() Type // тип литерала

ValueInt() int64 // как целочисленное значение

ValueFloat() float64 // как значение с плавающей точкой

ValueBool() bool // как булево значение

ValueString() string // как строка

Len() int // длина списка

Item(int) Litteral // элемент списка

Field(string) Symbol // поле записи

}

**Листинг 2.4:** Интерфейс Symbol.

type Symbol interface {

String() string // строковое представление символа

Kind() Kind // вид символа

Name() string // имя символа

Type() Type // тип символа

Value() Litteral // значение символа

Set(Litteral) // установить новое значение переменной

Fn() \*ast.FunctionDecl // получить пользовательскую функцию

Builtin() func(...Litteral) Litteral // получить встроенную функцию

Path() string // получить путь импортируемого модуля

}

В качестве символьной таблицы (контекста) выступает интерфейс **Environment**. Он позволяет добавлять, искать, обновлять символ внутри себя и искать символы внутри родительского (предшествующего) контекста. Его определение приведено в листинге 2.5.

Это все основные сущности, используемые во время выполнения интерпретатора. Из взаимосвязь приведен на рисунке 2.1.

**Листинг 2.4:** Интерфейс Symbol.

type Environment interface {

String() string // строковое представление окружения

Name() string // имя окружения

Parent() Environment // родительский элемент окружения

SearchLocal(string) Symbol // поиск среди местных символов

Search(string) Symbol // поиск среди местных и родительских символов

Insert(Symbol) // добавление нового местного символа

}

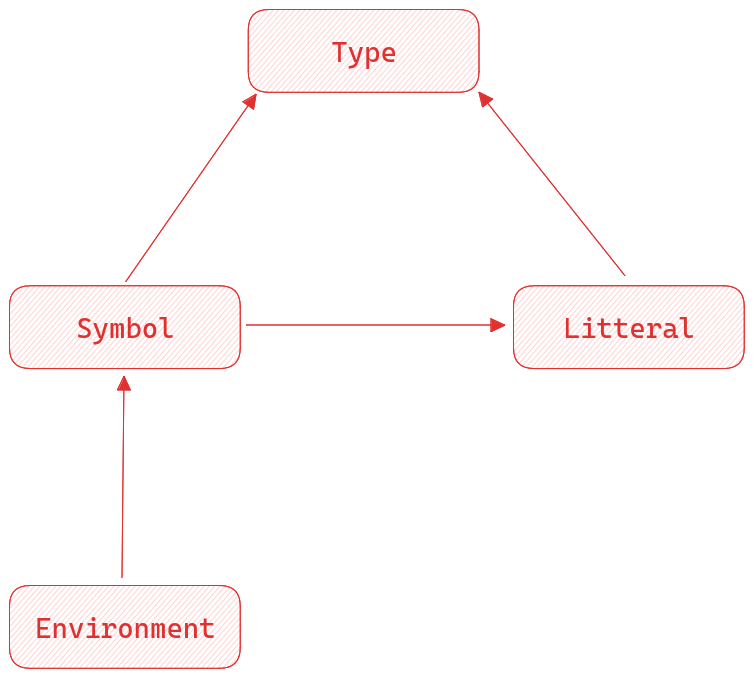


Рис. 2.1 Взаимосвязь элементов среды выполнения.

Далее на очереди идет совокупность функций, которые определяют интерпретатор и обрабатывают соответствующие им элементы АСД. Вся эта система функции и их взаимосвязь представлены на рисунке 2.2.

Функции выполнения совокупностей (объединений) элементов АСД:

**EvalGlobal** – выбирает соответствующую функцию для элемента объединения **Global**;

**EvalLocal** –выбирает соответствующую функцию для элемента объединения **Local**;

**EvalAtom** – создает соответствующий **Litteral** для элемента объединения **Atom**;

**EvalType** – создает соответствующий **Type** для элемента объединения **Type**;

**EvalExpression** – выбирает соответствующую функцию для элемента объединения Expression;

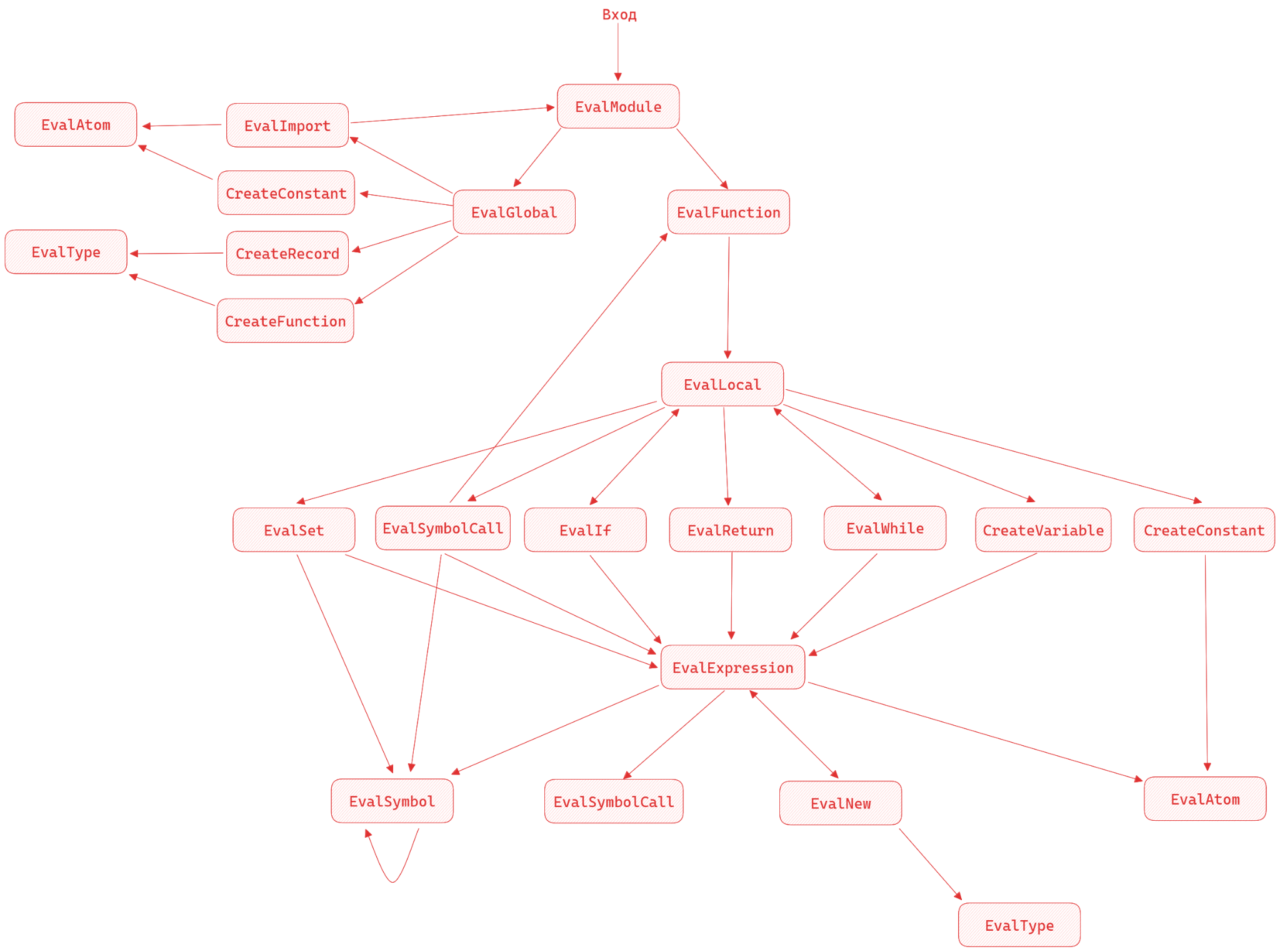


Рис. 2.2 Карта вызовов функций интерпретатора.

Функции выполнения конкретных элементов АСД:

**EvalModule** – входная точка в интерпретатор. Осуществляет выполнения модуля и создает соответствующее ему глобальное окружение (контекст);

**EvalFunction** – осуществляет выполнение пользовательских функций и создает для него локальное окружение;

**EvalSymbol** – осуществляет преобразование символа и его поиск в последовательности окружений, модулей и записей. Одна из краеугольных и часто используемых функций;

**EvalSymbolCall** – осуществляет выполнение вызова пользовательской или встроенной функции;

**EvalWhile** – осуществляет выполнение цикла while-else, создавая новое локальное окружение для каждой итерации цикла;

**EvalIf** – осуществляет выполнение ветвления if-elif-else, создавая новое локальное окружение;

**EvalNew** – осуществляет создание объектов списков и пользовательских записей;

**EvalReturn** – осуществляет выход из функции и удаление связанных с ней окружений;

**EvalSet** – осуществляет присвоение переменной нового значения того же типа;

**EvalImport** – осуществляет создание глобального окружения для соответствующего модуля и создает ему псевдоним в рамках текущего глобального модуля.

Функции создания **Symbol** из элементов АСД:

**CreateConstant** – создает константу в рамках текущего модуля;

**CreateVariable** – создает переменную в рамках текущего модуля;

**CreateFunction** – создает функцию в рамках текущего модуля;

**CreateRecord** – создает пользовательский тип записи в рамках текущего модуля.

# Тестирование интерпретатора

Интерпретатор написан, пришла пора его тестировать. В качестве тестового окружения используются встроенные модули и функции, представленные в листинге 3.1.

**Листинг 3.1:** Тестовое окружение.

-----< Environments

builtin |>

exit -> fn[int]<void>

int -> import(int)

io -> import(io)

list -> import(list)

string -> import(string)

float -> import(float)

string |>

format -> fn[string ...string]<string>

float |>

add -> fn[...float]<float>

sub -> fn[...float]<float>

lt -> fn[...float]<bool>

toString -> fn[float]<string>

int |>

sub -> fn[...int]<int>

lt -> fn[...int]<bool>

toString -> fn[int]<string>

add -> fn[...int]<int>

io |>

println -> fn[...string]<void>

list |>

len -> fn[list<any>]<int>

get -> fn[list<any> int]<any>

insert -> fn[list<any> int any]<list<any>>

append -> fn[list<any> any]<list<any>>

В качестве первой тестовой программы воссоздадим рекурсивный алгоритм Фибоначчи, который должен посчитать его 30 число, получив на выходе 832040. Код данной программы приведен в листинге 3.2. Целью данного теста является проверка работы базовых функций языка, таких как создание символов (переменных, констант, функций), вызов функции, рекурсия, условия ветвления, возврат и передача значений. Можно заметить, что интерпретатор сработал верно, но время выполнения составило 6 секунд, что довольно много для данного алгоритма.

**Листинг 3.2:** Рекурсивный алгоритм Фибоначчи.

(@module :main

(@fn fib [n::int] <int> :do

(@if (int/lt n 2) :then

(@return n)

)

(@return (int/add

(fib (int/sub n 1))

(fib (int/sub n 2))))

)

(@fn main :do

(@const n 30)

(io/println (string/format "fib($$) = $$"

(int/toString n)

(int/toString (fib n)))

)

(exit 0)

)

)

Результат работы:

$ time make run\_test

fib(30) = 832040

make run\_test 5,98s user 0,22s system 116% cpu 5,337 total

В качестве второго теста была создана программа создания списка векторов, добавления элемента в данный список и вывод его на экран до и после. Цель данного теста проверить корректность работы системы импортирования, вложенных структур, работы списков и циклов. Код программы приведен в листингах 3.3-5. По результатом видно, что интерпретатор работает в рамках ожидания.

**Листинг 3.3:** Создания списка векторов и вывод их на экран.

(@module :main

(@import "vector.grim" :as vec)

(@import "test\_v2.grim" :as t)

(@fn main :do

(@var vecList :of list<vec/vector> (@new list<vec/vector> {}))

(@while (int/lt (list/len vecList) 2) :do

(@var tmp :of vec/vector (vec/newVector 0. 0. 0. 0.))

(@set vecList (list/append vecList tmp))

)

(t/printVectors vecList)

(@set vecList (list/insert vecList 5 (vec/newVector 1. 1. 1. 2.)))

(t/printVectors vecList)

(@return)

)

)

Результат работы:

$ time make run\_test

before |>

vectors:

#0: {(0.000000;0.000000);(0.000000;0.000000)}

#1: {(0.000000;0.000000);(0.000000;0.000000)}

after |>

vectors:

#0: {(0.000000;0.000000);(0.000000;0.000000)}

#1: {(0.000000;0.000000);(0.000000;0.000000)}

#2: {(1.000000;1.000000);(1.000000;2.000000)}

make run\_test 0,10s user 0,07s system 232% cpu 0,071 total

**Листинг 3.4:** Модуль, содержащий определения вектора и необходимые функции.

(@module :code

(@record point :fields

x::float

y::float

)

(@record vector :fields

p0::point

p1::point

)

(@fn newVector [x0::float x1::float y0::float y1::float] <vector> :do

(@return (@new vector {

(@new point {x0 y0})

(@new point {x1 y1})

}))

)

(@fn pointToString [pnt::point] <string> :do

(@return (string/format "($$;$$)"

(float/toString pnt/x)

(float/toString pnt/y)

))

)

(@fn vectorToString [vec::vector] <string> :do

(@return (string/format "{$$;$$}"

(pointToString vec/p0)

(pointToString vec/p1)

))

)

)

**Листинг 3.5:** Модуль, содержащий функцию вывода вектора на экран.

(@module :code

(@import "vector.grim" :as vec)

(@fn printVectors [vectors::list<vec/vector>] :do

(io/println "vectors:")

(@var cnt :of int 0)

(@while (int/lt cnt (list/len vectors)) :do

(io/println (string/format "#$$: $$"

(int/toString cnt)

(vec/vectorToString (list/get vectors cnt))

))

(@set cnt (int/add cnt 1))

)

)

)

По итогам можно сказать что тестирование прошло успешно, весь функционал работает в рамках ожидания.

# 

# Заключение

В ходе данной НИР были выполнены все поставленные задачи. В частности:

1. Изучено устройство и принцип работы интерпретатора;
2. Создан собственный интерпретатор для простого статически типизированного ЯП;
3. Произведено тестирование правильности работы интерпретатора.

Дополнительно получены навыки разработки и проектирования комплексных и многомодульных программных систем, использования системы контроля версий.

По итогам работы цель “разработать интерпретатор для простого статически типизированного языка программирования” была достигнута.

Весь исходный код расположен в открытом доступе на Github репозитории по адресу: <https://github.com/SpectralJager/grimlang>

## 

# Список использованной литературы

1. Modern Compiler Design: 2nd ed. /Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, Koen Langendoen. – New York: Publisher Springer, 2012.- 843 p.
2. Torben Ægidius Mogensen*.* Basics of Compiler Design. – Copenhagen: Publisher lulu.com, 2010.-319 p.
3. Terence Parr. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages: 1st ed. – Raleigh: Publisher Pragmatic Bookshelf, 2010. 389 p.
4. The Go Programming Language [Электронный ресурс]. – Режим доступа: <https://go.dev/>, свободный. – Загл. с экрана. - Яз. англ.
5. A parser library for Go - GitHub [Электронный ресурс]. – Режим доступа: [https://github.com/ alecthomas/participle](https://github.com/alecthomas/participle), свободный. – Загл. с экрана. - Яз. англ.