Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана» (МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Разработка объектно-ориентированного языка со сборкой мусора»

Студент ИУ9-72(Б) _____ М.М. Масягин

(Подпись, дата)

Руководитель курсового проекта _____ А.В. Синявин

(Подпись, дата)

Москва, 2019 г.

Оглавление

Введение

На сегодняшний день развитие информационных технологий напрямую связано с разработкой новых языков программирования и повышением уровня их абстракций. Так, если 60 лет назад практически все программное обеспечение писалось на Assembler`e, а прикладные программисты оперировали регистрами и машинными словами, то сейчас существуют сотни высокоуровневых языков, а разработчики рассуждают в рамках классов и целых аппаратно-программных комплексов.

Основным отличием современных высокоуровневых языков от их предшественников является поддержка объектно-ориентированной парадигмы и автоматическая сборка мусора. Поддержка объектов повышает читаемость и переиспользуемость кода, а сборка мусора избавляет программиста от необходимости вручную очищать память. Все это делает написание программ проще и дешевле [1, 2].

Обратной стороной «упрощения» прикладных языков программирования является усложнение их трансляторов и интерпретаторов. Так, различные оптимизационные пассы и подсистемы сборки мусора, являясь крайне нетривиальными в реализации, в разы увеличивают объем исходного кода виртуальных машин и компиляторов.

Целью данной курсовой работы является разработка интерпретатора высокоуровневого скриптового языка программирования с поддержкой объектов и сборки мусора и исследование на его примере современных алгоритмов автоматического управления памятью.

1 Спецификация языка

Прежде чем начать разработку интерпретатора, необходимо составить подробную спецификацию целевого языка. Спецификации даже очень простых языков программирования могут занимать десятки и сотни страниц, поэтому далее будут описаны лишь ключевые особенности.

1.1 Синтаксис и основные управляющие конструкции

Язык, в дальнейшем GCI — «Garbage Collection Interpreter», является C-подобным и имеет синтаксис, сходный с JavaScript, Golang и непосредственно с C. Блоки кода заключаются между открывающей и закрывающей фигурными скобками («{», «}»), операции разделяются точкой с запятой («;»). Поддерживаются операторы ветвления: if-else, и операторы циклов: while, break и continue, в том числе и с многократным вложением. Инструкции циклов и условных операторов всегда должны ограничиваться фигурными скобками, как в Golang. Для создания новых переменных используется ключевое слово «let», как в JavaScript. Для максимальной простоты и предсказуемости работы языка отсутствуют побочные результаты операций, *goto*-метки и операторы ++ и --. Также в *let*-блоке допускается за раз объявление лишь одной переменной, инициализация которой обязательна. Более подробно с грамматикой GCI можно ознакомиться в приложениях 1 и 2.

1.2 Система типов

GCI имеет сильную динамическую неявную типизацию. Поддерживаются следующие типы данных:

- Примитивные:
  - Целые знаковые 8-байтные числа; диапазон значений: [−9223372036854775808, +9223372036854775807];
- Составные:

- Массивы — структуры данных, хранящие набор значений, идентифицируемых по целочисленному индексу. Индексация элементов начинается с нуля. Для массивов доступны операции: получения длины, добавления значения в конец и удаления значения по индексу;
- Объекты — ассоциативные массивы, ключами которых являются допустимые в языке идентификаторы, известные на момент запуска программы. Для объектов доступны две операции: добавления новой пары ключ-значение и удаление пары по ключу.

Элементами массивов и полями объектов могут быть другие массивы и объекты. Копирование примитивных типов данных происходит по значению, а составных — по ссылке. Рекурсивные ссылки полей объектов на сами объекты и элементов массива на сами массивы являются валидными.

На листинге 1 приведен пример программы на GCI, вычисляющий сумму квадратов четных элементов массива. В примере использовано большинство функций языка.

```
function test() {
	let a = {
		b : [1, 2, 3, 4, 5, 6, 7, 8, 9, -1]
	}; // Инициализация объекта и вложенного в него массива.
	a.b[9] = 10;
	a.c = a; // Циклическая зависимость.

	let i = -1;

	let sum = 0;

	while (i < 1000) { // While.
		i = i + 1;

		if (i == 10) { // if-then-else.
			Break; // break.
		} else if (a.b[i] % 2 != 0) {
			continue; // continue.
		} else {
			sum = sum + a.c.b[i] * a.c.b[i];
		}
```

```
        }

        return sum;
}
```

Листинг 1 — пример программы на языке GCI,
демонстрирующий основные возможности языка

2 Frontend интерпретатора

На сегодняшний день существует большое число генераторов синтаксических анализаторов. Наиболее распространенными из них являются: Yacc, GNU Bison, ANTLR и Lemon. Они позволяют по входному набору правил грамматики сгенерировать код парсера на C\C++ или Java. Несмотря на это, большинство frontend`ов для современных языков используют вручную написанные парсеры. Даже frontend GCC, изначально основывавшийся на GNU Bison, с версии 3.4 был переведен на метод рекурсивного спуска [3, 4]. Это обусловлено тем, что генераторы парсеров усложняют сборку и добавляют новые зависимости в проекты, в которых они используются. Помимо этого поведение автоматически созданных парсеров не всегда бывает предсказуемым, а их возможности по обработке ошибок далеки от идеала.

Таким образом, было решено написать frontend для GCI с нуля.

2.1 Лексический анализатор

Для разбиения текста исходной программы на лексемы используется объектно-ориентированный лексический анализатор [5].

Каждый токен является наследником базового абстрактного класса «Token», содержащего в себе информацию о начальной и конечной позиции токена.

Чтение входного потока символов и разбиение его на лексемы происходит в классе «Lexer». Определение типа текущего токена и его создание реализуется отдельным методом класса «Lexer» с помощью синтаксической конструкции switch. Это позволяет получать новые лексемы из входного потока по мере надобности, что упрощает реализацию «параллельного» лексического и синтаксического анализа программы.

При обнаружении некорректной лексемы, она пропускается, а в stderr пишется соответствующее предупреждение.

Время работы лексера — линейное.

На рисунке 1 изображена диаграмма классов, используемых в лексере. Она наглядно демонстрирует устройство анализатора.

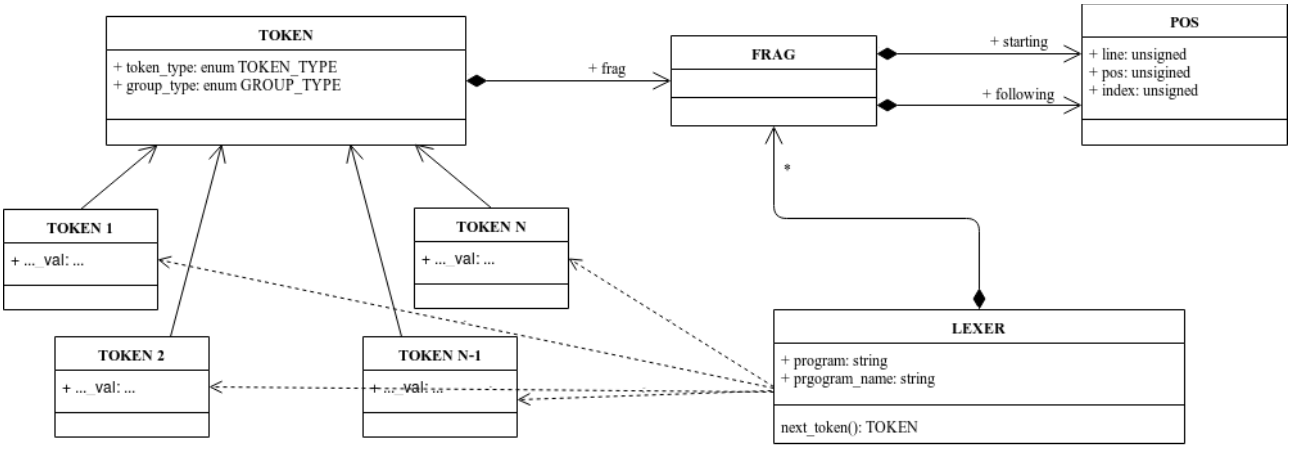


Рисунок 1 - диаграмма классов объектно-ориентированного лексера

2.2 Синтаксический анализатор

Для построение синтаксического дерева используется метод рекурсивного спуска [1]: каждому нетерминалу грамматики соответствует своя функция-«парсер». Функции рекурсивно вызывают друг другу, начиная с функции-«парсера» для всей программы, и создают объекты классов, соответствующих данным нетерминалам. Так как грамматика языка является практически LL(1) (практически, потому что различить «variable» и «function_call» невозможно по единственному текущему токену-идентификатору), то используется предикативный анализ, что позволяет избежать возвратов. Таким образом, время работы парсера является линейным.

3 Backend интерпретатора

Backend интерпретатора получает на вход AST-дерево, созданное во frontend`e, предобрабатывает его и исполняет полученный код.

Помимо непосредственного исполнения кода в backend`e интерпретатора могут производиться различные оптимизации, генерироваться машинный код для наиболее активно используемых участков программы (JIT) и т. д. Так как целью курсовой работы в первую очередь является изучение алгоритмов сборки мусора, то backend GCI состоит из трех частей: генератора байт-кода, виртуальной машины и сборщика мусора с аллокатором. JIT-компиляция и возможные оптимизационные пассы не рассматриваются. В данном разделе описаны генератор байт-кода и виртуальная машина GCI.

3.1 Генератор байт-кода и виртуальная машина

На сегодняшний день большинство интерпретаторов по принципу работы можно отнести к одному из трех классов:

- Интерпретаторы, использующие обход дерева;
- Стековые виртуальные машины;
- Регистровые виртуальные машины;

Интерпретаторы первого типа являются наиболее простыми в реализации, так как не требуют фазы генерации байт-кода. Представление их состояния во время выполнения напрямую отображается на синтаксическое дерево программы. Главным недостатком подобных интерпретаторов являются низкая скорость работы и высокие накладные расходы по памяти, вызванные необходимостью многократно обходить дерево разбора, даже для выполнения простейших операций. Так, для вычисления выражения «1 + 2» помимо одной операции сложения необходимы десятки операций разыменования указателя, а вместо 16 байт памяти (два 8-байтных числа) задействуется свыше 100. На рисунке 2 изображен процесс обхода участка AST-дерева для выражения «1 + 2».

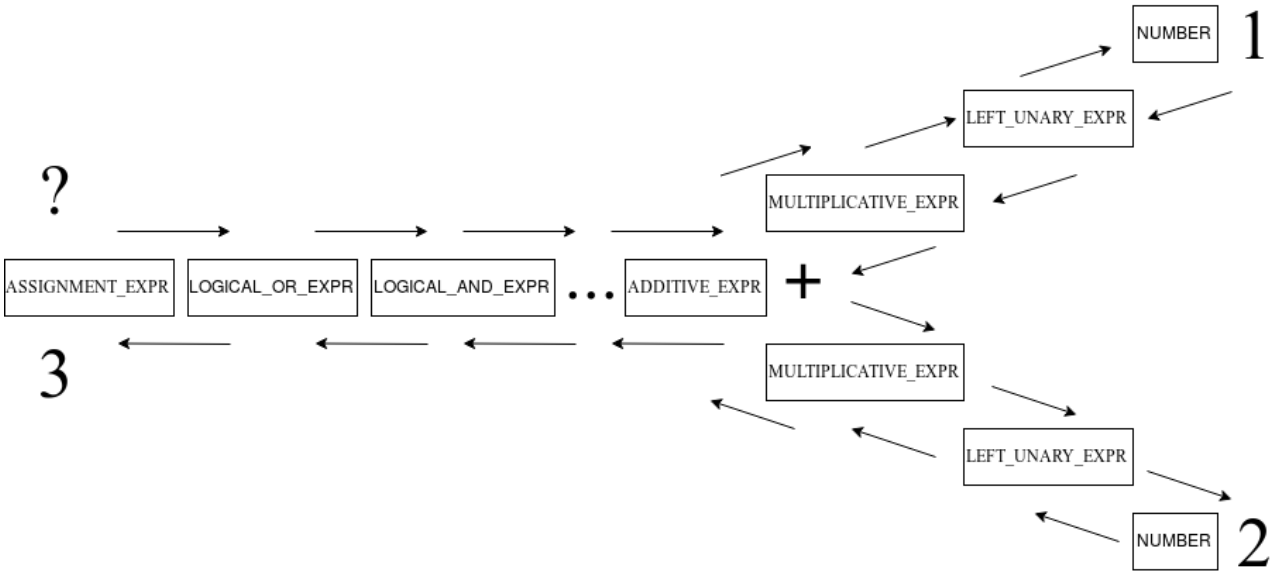


Рисунок 2 - процесс обхода участка AST-дерева для выражения «1 + 2».

Наиболее известными примерами «интерпретаторов AST-дерева» являются учебные Lox [6] и LSBASI [7].

Интерпретаторы второго и третьего типа не взаимодействуют с AST-деревом напрямую. Вместо этого они исполняют байт-код — компактное представление исходной программы в виде максимально простых n-арных (обычно унарных, бинарных и тернарных) операций. Основным преимуществом использования байт-кода по сравнению с прямым обходом дерева является повышение скорости выполнения. Это связано с отсутствием накладных расходов на большое число указателей в AST-дереве и простоту оптимизации байт-кода. Сам байт-код либо генерируется непосредственно перед началом исполнения программы, либо может быть взят уже готовым (даже с другого компьютера).

Ключевое различие между интерпретаторами второго и третьего типов (между стековыми и регистровыми виртуальными машинами) заключается в способе хранения данных.

Стековые машины хранят все свои данные в одном или нескольких стеках. Для обращения к ним используется адресация относительного положения требуемой ячейки от текущей вершины стека. Если у операции есть

результат, то он помещается в стек, становясь его новой вершиной. Кроме данных, в стеке могут храниться адреса, используемые при возврате из вызванных подпрограмм в вызывающие. Существуют реализации, в которых адреса хранятся в отдельном стеке.

Наиболее известными примерами стековых виртуальных машин являются Oracle JVM и Microsoft CLR.

Регистровые виртуальные машины, в отличие от стековых, для хранения данных используют выделенный набор ячеек памяти с фиксированными именами-номерами — регистры. Инструкции байт-кода взаимодействуют с данными на регистрах, при необходимости загружая отсутствующие значения из памяти или выгружая ненужные в память.

В качестве примера регистровых виртуальных машин можно привести Dalvik от Google, LLVM bitcode и Lua 5.0.

При прямом сравнении регистровых и стековых машин можно сделать вывод, что команды стековых ВМ имеют меньшую длину, так как они не требуют указания адресов регистров, однако они активнее используют память в связи с постоянными вызовами стековых функций Push() и Pop() для обработки операций. На рисунке 3 демонстрируется разница между стековым и регистровым подходами.

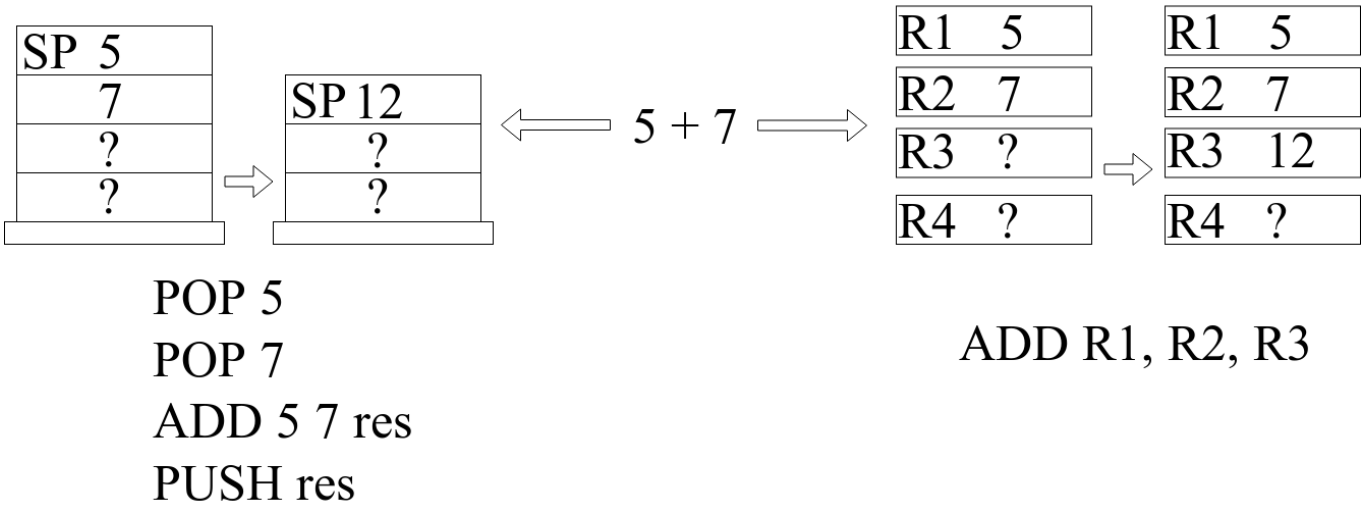


Рисунок 3 — операция сложения в стековой и в регистровой ВМ

Также стоит отметить, что при трансляции с исходного языка регистровые ВМ используют алгоритмы распределения регистров, реализация которых крайне сложна. В случае стековых ВМ вершина стека всегда одна, и она всегда «доступна». Таким образом, преобразование AST-дерева в байт-код проще реализовать для стековой ВМ, чем для регистровой.

При сравнении производительности регистровые виртуальные машины обгоняют стековые, однако зачастую не настолько, чтобы оправдать усложнение реализации [8].

После оценки всех за и против было решено использовать стековую виртуальную машину.

Она использует один стек для данных и два регистра:

- SP (Stack Pointer) — указатель на вершину стека;
- IP (Instruction Pointer) — указатель на текущую инструкцию байт-кода;

На рисунке 4 представлено схематичное устройство виртуальной машины.

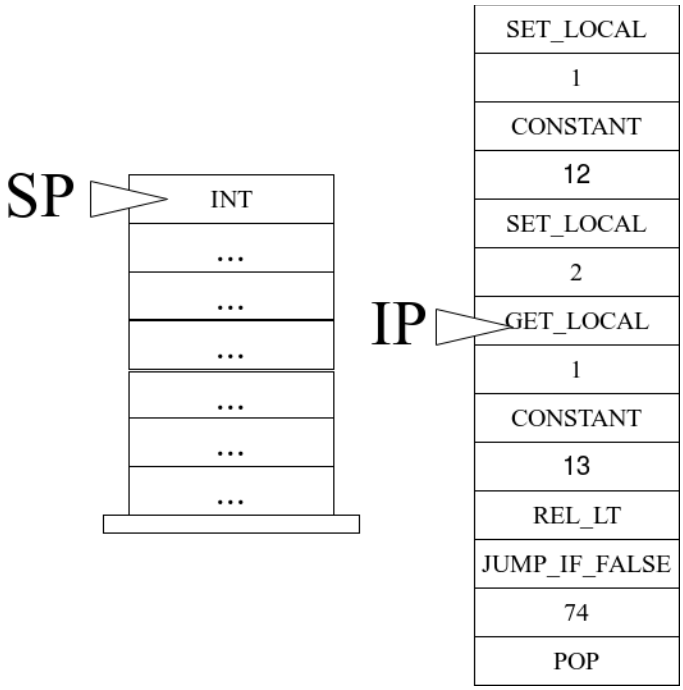


Рисунок 4 - схематичное устройство виртуальной машины

Наиболее важными командами в контексте объектно-ориентированности GCI и его автоматического управления памятью являются:

- *CREATE_OBJ* — команда создания объекта в куче;
- *CREATE_ARR* — команда создания массива в куче;
- *GET_HEAP* — команда получения значения из кучи;
- *SET_HEAP* — команда установки значения в куче;

CREATE_OBJ и *CREATE_ARR* запрашивают у аллокатора новый блок памяти. В качестве параметров они передают начальный размер объекта.

Так как GCI поддерживает многомерные массивы и вложенные объекты, а также циклические ссылки, то *GET_HEAP* и *SET_HEAP* являются наиболее сложными командами байт-кода. В качестве параметров они принимают: idx — индекс стековой переменной, len — число аргументов и непосредственно аргументы — пары флаг-число.

Если флаг выставлен в значение 0, то следующее за ним число интерпретируется как смещение (*shift*) относительно вершины стека. В ячейке стека с позицией *SP — shift* лежит индекс массива. Важно отметить, что при генерации байт-кода инструкции для индексных выражений должны генерироваться непосредственно перед инструкциями индексации, поэтому выражение вида «*return a[i][j]*» превращается в последовательность инструкций:

GET_LOCAL 1

GET_LOCAL 2

GET_HEAP 0 2 0 1 0 0

RETURN

Если флаг выставлен в значение 1, то следующее за ним число интерпретируется как имя поля объекта. В том что во время исполнения в качестве имен полей используются числа нет ничего удивительного. Сравнение чисел является более дешевой операцией, чем сравнение строк, поэтому на этапе генерации байт-кода все константы, как числовые, так и символьные, выносятся в отдельный пул констант, а байт-код оперирует не ими самими, а их

индексами в пуле констант. Подобная схема используется в JVM [9]. Таким образом, код «*let a = {}; a.b = 1*» трансформируется в инструкции:

CREATE_OBJ 0

SET_LOCAL 0

SET_HEAP 0 1 1 1,

причем в пуле констант есть метка:

FIELDREF_CNST IDX = 1 VALUE = b.

Также пул констант используется и для числовых идентификаторов.

Таким образом, смешанное выражение, состоящее из обращений как к элементам массивов, так и к полям объектов, например, *«a.b[i][1]»* транслируется в:

GET_LOCAL 1

CONSTANT 1

GET_HEAP 0 3 1 2 1 1 1 0,

причем в пуле констант имеются метки:

INT_CNST IDX = 1 VALUE = 1

FIELDREF_CNST IDX = 2 VALUE = b.

Полный список команд ВМ приведен в приложении 3.

4 Сборщик мусора

В данном разделе будут рассмотрены проблемы, с которыми сталкиваются программисты, при разработке программ на языках программирования с ручным управлением памятью, описаны способы борьбы с ними и приведено подробное описание алгоритмов сборки мусора и аллокации, используемых в GCI.

4.1 Ошибки, связанные с ручным управлением памятью

Как показывает практика, наиболее распространенными ошибками при использовании таких языков как C, C++ и Fortran являются ошибки, связанные с ручным управлением памятью.

Большинство подобных ошибок можно отнести к одной из четырех групп:

- использование указателя как массива;
- применение free (delete) к объекту из стека;
- неверное выделение/освобождение динамической памяти;
- использование ссылки или указателя на «переехавший» или очищенный участок памяти;

Подробно рассмотрим каждую группу ошибок.

При использовании указателя (не инициализированного или на один элемент) как массива происходит обращение к недоступным для записи участкам памяти, что в свою очередь вызывает ошибку сегментации, также известную как ошибка адреса/ошибка шины. На листинге 2 приведен пример подобной ошибки на языке C.

```
#include <stdlib.h>

int main()
{
        int *arr = (*int) malloc(sizeof(int));
        for (int i = 0; i < 10; i++) {
```

```
                arr[i] = i; // ошибка сегментации.
        }

        return 0;
}
```

Листинг 2 — пример программы на языке C, демонстрирующий ошибку при использовании указателя как массива

Стековая память, в отличие от динамической, освобождается автоматически при выходе из текущего блока кода. Поэтому применение к ней операции free (delete) недопустимо — это приводит к ошибке сегментации. Проблема усугубляется тем, что большинство компиляторов, IDE и анализаторов кода не способны обнаружить данную ошибку при очистке стековой памяти по указателю, переданному в стороннюю функцию.

На листингах 3 и 4 приведены 2 программы. В первой из них компилятор (GNU GCC 5.4.0) обнаружит очистку стековой памяти, во второй — нет.

```
#include <stdlib.h>

int main()
{
        int arr[5];
        free(arr);
        /*
          На этапе компиляции с флагами -Wall -Wextra GCC выдаст предупреждение:
          kek.c: In function ‘main’:
          kek.c:6:5: warning: attempt to free a non-heap object ‘arr’ [-Wfree-nonheap-object]
                free(arr);
                ^
        */
}
```

Листинг 3 — пример программы, очищающей стековую память, при компиляции которой компилятор обнаруживает ошибку.

```
#include <stdlib.h>

void free_arr(int*arr);

int main()
{
```

```
        int arr[5];
        free_arr(arr);
}

void free_arr(int*arr)
{
        free(arr); // Ошибка сегментации
}
```

Листинг 4 — пример программы, очищающей стековую память, при компиляции которой компилятор НЕ обнаруживает ошибку.

Неверное выделение/освобождение динамической памяти является повсеместной проблемой для всех сравнительно низкоуровневых языков. Особенно актуальна она становится при написании программ, работающих 24/7 и оперирующих сложными структурами данных. И если ошибки, связанные с неверным/неполным выделением памяти, обнаруживаются достаточно быстро, так как они приводят к ошибкам сегментации, то ошибки, связанные с неполной очисткой памяти, могут существовать годами даже в крупных проектах. Даже использование таких инструментов, как Valgrind [10] и IBM Rational Purify, не всегда способно гарантировать валидность программы. Пример кода, имеющего утечку памяти, приведен на листинге 5.

```
#include <stdlib.h>

int main()
{
        int*arr = (int*) malloc(10 * sizeof(int));
        int i;
        for (i = 0; i < 10; i++) {
                arr[i] = i;
        }

        /*
          здесь должен быть вызов функции free(arr);
          на 64-битной системе его отсутствие вызовет утечку 40 байт памяти;
        */

        return 0;
}
```

Листинг 5 — пример программы на языке C, имеющей утечки памяти.

Наиболее трудноуловимыми являются ошибки, связанные со ссылками (указателями) на «переехавшие» и удаленные данные. Часто они возникают при работе с массивами, когда 2 указателя, изначально указывающих на один и тот же блок памяти, начинают указывать на два разных (а зачастую на один валидный, а другой невалидный), при изменении размера массива по одному из них.

На листинге 6 приведен пример кода, содержащего данную ошибку.

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
	int*arr1 = (int*) malloc(sizeof(int));
	int*arr2 = arr1;

	arr1[0] = 0;
	printf("arr1[0]: %d; arr2[0]: %d\n", arr1[0], arr2[0]); // arr1[0]: 0; arr2[0]: 0

	arr2 = (int*) realloc(arr2, sizeof(int) * 100000);
	arr1[0] = 1; // ошибка записи;
	printf("arr1[0]: %d; arr2[0]: %d\n", arr1[0], arr2[0]); // arr1[0]: 1; arr2[0]: 0 // ошибка
чтения

	return 0;
}
```

Листинг 6 — пример программы на языке C,
демонстрирующей проблему ссылок на «переехавшие» данные.

Очевидным способом борьбы со всеми вышеперечисленными ошибками является отказ от явного использования free (delete), отказ от арифметики указателей и поддержание всех ссылок (указателей) всегда актуальными. В этом программисту помогает сборка мусора — один из видов автоматического управления памятью, при котором обязанность освобождения памяти возлагается на среду исполнения программы.

4.2 Два основных подхода к сборке мусора

Все существующие на сегодняшний день алгоритмы сборки мусора по принципу работы можно разделить на 2 класса:

- Алгоритмы, использующие подсчет ссылок;
- Алгоритмы на основе отслеживания;

Первые вместе с каждым объектом ассоциируют счетчик ссылок: число, показывающее из скольких точек программы в данный момент достижим объект. Ниже приведены основные правила изменения счетчика ссылок:

- При создании объекта счетчик ссылок становится равен 1;
- При передаче объекта в процедуру счетчик увеличивается на 1;
- В случае присваивания между ссылками *«u = v»*, счетчик ссылок объекта *u* уменьшается на 1, а *v* увеличивается на 1;
- При возврате из процедуры счетчики ее объектов-параметров должны быть уменьшены на 1;
- При обнулении ссылки объект должен быть удален;

Главными преимуществами счетчиков ссылок являются простота реализации и возможность их использования даже в языках с ручным управлением памятью. Так, в C++11 были добавлены *«std::unique_ptr»*, *«std::shared_ptr»* и *«std::weak_ptr»* [11]. Также к плюсам относятся инкрементность работы и немедленная очистка недостижимой памяти.

Однако счетчики ссылок имеют ряд существенных недостатков: они не способны отследить циклические зависимости (даже использование умного *«std::weak_ptr»* требует от программиста как минимум понимания того, что в данном участке кода возможна циклическая ссылка), и они сильно нагружают рантайм.

На рисунке 5 приведен пример недостижимой циклической структуры данных, которая никогда не будет очищена при использовании счетчиков ссылок.

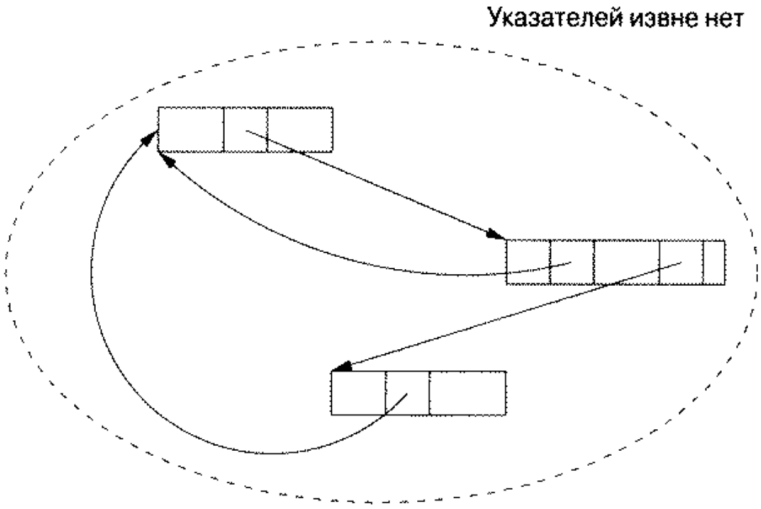


Рисунок 5 - пример циклических ссылок

Алгоритмы на основе отслеживания, в отличие от счетчиков ссылок, вместо сборки мусора в момент его появления периодически запускаются для поиска недостижимых объектов и очистки используемой ими памяти. Обычно они начинают работу при исчерпании свободной памяти или когда ее количество становится меньше некоторого граничного значения.

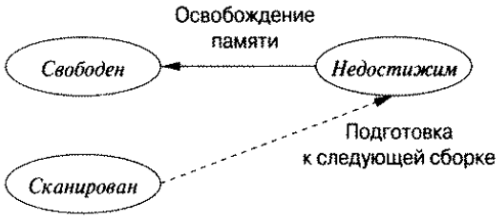
Существует большое число различных алгоритмов сборки мусора на основе отслеживания, однако все они могут быть описаны в терминах следующих состояний блоков памяти:

- *Свободен*: данный блок готов для выделения новой памяти; он не может хранить достижимый объект;
- *Недостижим*: все те блоки, достижимость которых НЕ была доказана путем отслеживания;
- *Несканирован*: все те блоки, достижимость которых было доказана, но их указатели еще не были просканированы;
- *Сканирован*: все те блоки, для которых были отслежены как они сами, так и все их указатели;

На рисунке 6 показаны 3 основных этапа отслеживающей сборки мусора:

а) Действие перед отслеживанием;

б) Отслеживание достижимости;



в) Сборка мусора;

Рисунок 6 — этапы работы сборщика мусора

Основными преимуществами отслеживающих сборщиков мусора являются в среднем низкие накладные расходы рантайма (так как сборка, если и происходит, то нечасто) и гарантированная очистка даже сложных циклических структур.

К их недостаткам можно отнести невозможность использования в системах реального времени, так как в момент сборки мусора они либо останавливают все потоки либо, либо как минимум значительно снижают производительность всего приложения.

4.3 Алгоритмы сборки мусора и аллокации в GCI

Так как GCI создается как скриптовый язык программирования, то вероятность его применения в системах реального времени крайне мала. Таким образом, наиболее логично использовать один из алгоритмов отслеживающей сборки мусора.

Важно понимать, что сборщик мусора никогда не работает «в одиночку». Все эффективные автоматические системы управления памятью с целью повышения производительности используют свой собственный аллокатор, а не вызывают *«malloc»*, *«realloc»* и *«free»* (*«new»* и *«delete»*) напрямую для каждого

создаваемого/удаляемого объекта. Подобным образом было решено поступить и в GCI.

Аллокатор, используемый в GCI, изначально с помощью *«malloc»* выделяет в куче большой участок памяти, после чего по запросу отдает из него новые блоки.

Каждый блок GCI представляет из себя «структуру» (на самом деле это массив байт) из 7 элементов. Она подробно описана на рисунке 7.

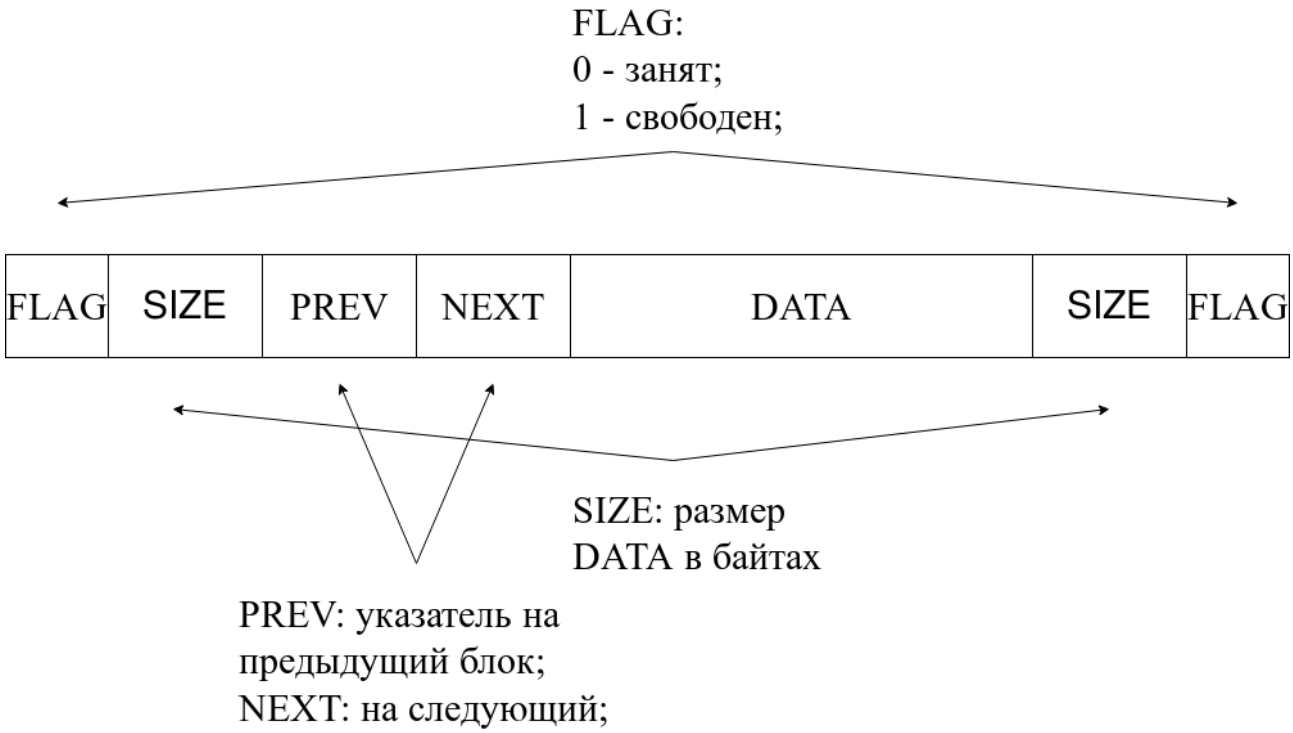


Рисунок 7 - блок аллокатора

Ячейки *FLAG* на границах блоков показывают занят текущий блок или нет, *SIZE* показывают размер блока *DATA*, а *PREV* и *NEXT* являются указателями на элементы списка свободных/занятых блоков. Важно отметить, что список свободных блоков отсортирован по возрастанию; список занятых — не отсортирован. Пары *FLAG-DATA* на концах блока называются *дескрипторами границ* [1].

Изначально весь аллокатор состоит из одного большого свободного блока.

При очередном запросе памяти происходит поиск свободного блока минимального размера, способного вместить в себя передаваемый размер данных + размер дескрипторов границ и 2-х указателей (такая стратегия аллокации, называемая стратегией «наилучшего подходящего», позволяет снизить фрагментацию):

- если размер найденного блока достаточен для хранения требуемой информации, но недостаточен для хранения информации об еще одном блоке (размером хотя бы 1 байт), то он помечается занятым и перемещается из списка свободных блоков в список занятых; вызывающей функции передается указатель на *DATA*-сегмент этого блока;
- если размер найденного блока достаточен для хранения и требуемой информации, и хотя бы еще одного байта (с учетом накладных расходов на 4 дескриптора границ и 4 указателя 2-х блоков), то он разбивается на 2 блока. Первый помечается занятым и добавляется в список занятых блоков, а второй — свободным и помещается в нужное место списка свободных блоков; вызывающей функции передается указатель на *DATA*-сегмент нового блока, помеченного занятым;

При нехватке памяти для выделения нового блока аллокатор возвращает *NULL*-указатель.

Так как размер объектов и массивов в GCI со временем может изменяться, то аллокатор поддерживает и запрос увеличения размера текущего блока памяти. При этом возможны два варианта работы функции:

- если справа от модифицируемого блока имеется свободный блок, достаточный для того, чтобы вместить добавочный размер модифицируемого блока, то функция отрабатывает практически как malloc: в правом свободном блоке выделяется дополнительная память для модифицируемого блока; указатель модифицируемого блока остается неизменным;

- если справа от модифицируемого блока нет подходящего свободного блока, то выделяется новый блок памяти и в него копируются данные старого блока; после этого старый блок становится свободным, и, если по соседству с ним имеются другие свободные блоки, объединяется с ними; указатель модифицируемого блока изменяется;

При нехватке памяти для увеличения размера блока аллокатор возвращает *NULL*-указатель.

Теперь перейдем непосредственно к описанию алгоритма сборки мусора. За основу был взят копирующий сборщик мусора Чейни [12, 1], использующий две области кучи (два аллокатора).

Изначально оба аллокатора, назовем их *A* и *B,* занимают размер *X*, а объекты создаются (или увеличиваются в размере) только в аллокаторе *A*. Рано или поздно наступает момент, когда свободная память в аллокаторе *A* заканчивается даже с учетом всех его «*realloc*»-оптимизаций.

В этот момент запускается алгоритм сборки мусора: все достижимые объекты из аллокатора *A* переносятся в память аллокатора *B* и уплотняются. Рассмотрим этот процесс подробнее:

1. Инициализируется хэш-таблица из указателей в указатели, отдающая при запросе по несуществующему ключу *NULL*-указатель;
2. Для всех ссылок из корневого множества-стека вызывается подпрограмма «*LookUpNewLocation*», описание которой будет приведено ниже;
3. До тех пор пока все объекты, перенесенные в аллокатор *B* не будут обработаны: ко всем ссылкам каждого объекта применить подпрограмму «*LookUpNewLocation*».

Подпрограмма «*LookUpNewLocation*» в свою очередь состоит из 3-х действий:

1. Проверка наличия ссылки в хэш-таблице;

2. Если ссылка отсутствует, то объект, доступный по ней, копируется из *A* в *B*, и в хэш-таблицу по ключу старой ссылки добавляется значение новой ссылки;
3. В вызывающую функцию возвращается обновленная ссылка;

Таким образом, на основе корневого множества последовательно, уровень-за-уровнем вложенности объектов в *B* переносятся все достижимые объекты.

Более наглядно этот процесс для одного отдельно взятого объекта изображен на рисунке 8.

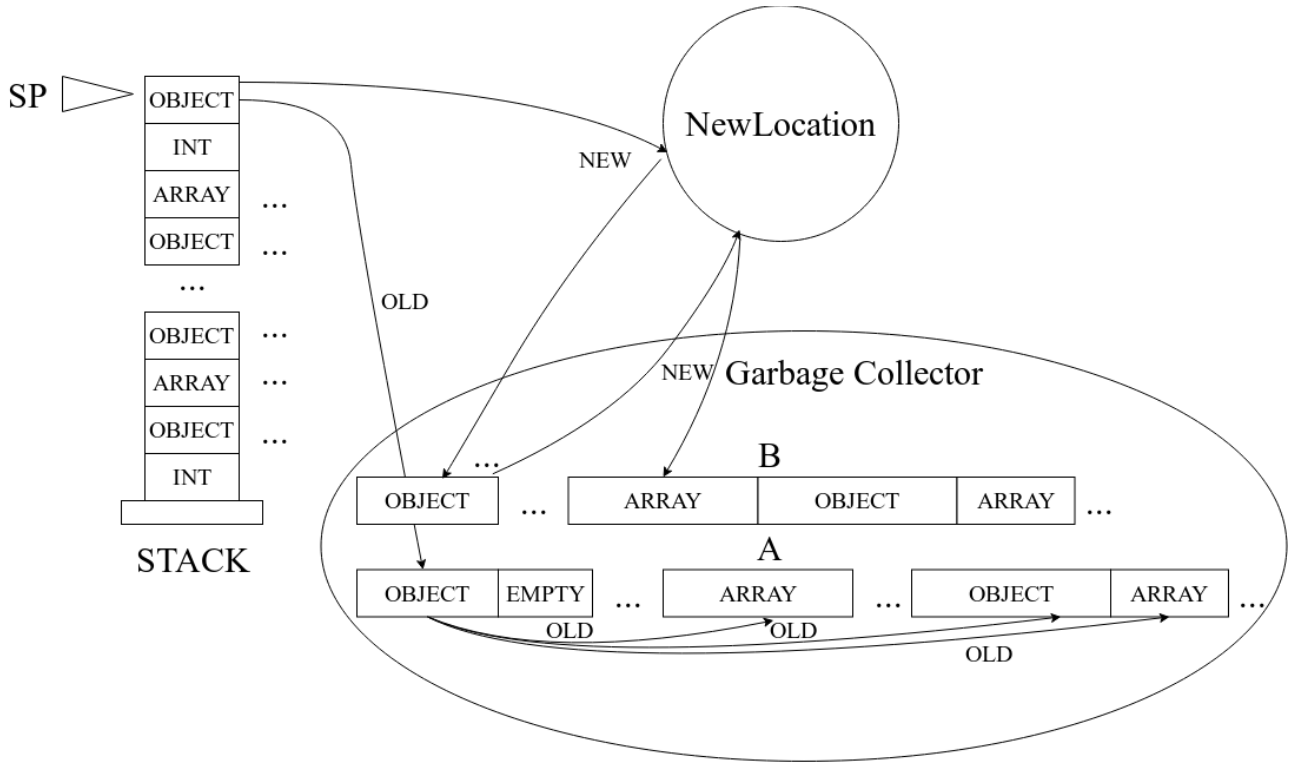


Рисунок 8 - перемещение достижимого объекта и его достижимых подобъектов

После того как перенос объектов был завершен, и в аллокаторе *B* содержатся лишь актуальные данные без «дыр» между ними, а в корневом множестве-стеке хранятся указатели на блоки *B,* мы можем поменять аллокаторы *A* и *B* местами (обменять указатели на них). Если при этом размер всех уплотненных актуальных объектов + размер нового объекта превышают половину размера аллокатора, то его размер увеличивается вдвое с помощью

вызова «*realloc*». Если «*realloc*» возвращает старый участок памяти, то сборка мусора завершается, если же блоки аллокатора переносятся в новое место, то требуется произвести еще одну сборку мусора для обновления всех указателей. Лишней сборки мусора можно избежать, если сохранить старый указатель на аллокатор. Так как оба списка блоков памяти аллокатора реализованы на массиве, то значения указателей в них имеют вид *old_ptr + shift.* Зная старый указатель *old_ptr* и новый указатель *new_ptr*, можно линейно пройти по всем блокам и заменить указатели *old_ptr + shift* на *new_ptr + shift,* тем самым избегая повторных дорогостоящих операций копирования. Все указатели в стеке также могут быть обновлены за линейное время.

Таким образом, время работы сборщика мусора пропорционально общему числу достижимых объектов. Основными его преимуществами являются низкая фрагментация памяти в процессе работы, и отсутствие взаимодействия с недостижимыми объектами при непосредственной сборке мусора, чего не могут обеспечить многие другие распространенные алгоритмы, например, «*Mark&Sweep*» или «*Baker*» [1].

К недостаткам данного алгоритма можно отнести высокие затраты на копирование больших объектов, переживающих несколько циклов сборки мусора, но с этой проблемой можно бороться, перенося долгоживущие объекты в отдельную кучу, сборка мусора в которой происходит реже, чем в основной.

5 Реализация

Интерпретатор написан на чистом C и может быть собран любым C99-совместимым компилятором, а при замене библиотечных функций *«snprintf»* и *«atoll»* на собственные реализации и C89-совместимым компилятором. Это делает его легко переносимым на большинство существующих платформ.

Для удобства работы с промежуточными представлениями программы: набором лексем, AST-деревом и байт-кодом все они могут быть легко экспортированы в формат xml. Также в xml-формате может писаться и покомандная трассировка программы. Все это делает работу с интерпретатором проще и приближает его к реальным промышленным интерпретаторам и компиляторам.

На листинге 7 приведен пример дампа байт-кода GCI в xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<bytecode>
        <constant_pool>
                <int_cnst idx="0">4</int_cnst>

                <int_cnst idx="4">0</int_cnst>
        </constant_pool>
        <op_codes>
                <op>CONSTANT 0</op>
                        ...
                <op>REL_LT</op>
                <op>JUMP_IF_FALSE 22</op>
                <op>POP<op>
                <op>GET_LOCAL 2</op>
                <op>GET_LOCAL 1</op>
                <op>GET_HEAP 0 index(0)</op>
                <op>ADDITIVE_PLUS</op>
                <op>SET_LOCAL 2</op>
                <op>GET_LOCAL 1</op>
                <op>CONSTANT 3</op>
                <op>ADDITIVE_PLUS</op>
                <op>SET_LOCAL 1</op>
                <op>JUMP -29</op>
                        ...
                <op>POP<op>
        </op_codes>
</bytecode>
```

Листинг 7 - пример дампа байт-кода GCI в xml.

6 Тестирование

Тестирование интерпретатора проводилось по 4 основным направлениям:

- Тестирование корректности работы управляющих конструкций языка;
- Тестирование корректности работы аллокатора и сборщика мусора;
- Тестирование всего интерпретатора на отсутствие утечек памяти даже при некорректных входных данных (для этого использовался Valgrind);
- Тестирование всего интерпретатора на программе, приближенной к реальной жизни, и активно задействующей сборку мусора. Рассмотрим ее подробнее.

Пусть даны две армии, имеющие на вооружении пехоту, танки и самолеты. Каждая боевая единица имеет три характеристики: здоровье, наносимый урон и вероятность уклонения от выстрела противника. Каждый ход случайная боевая единица одной армии может атаковать случайную боевую единицу другой. Если при этом здоровье атакуемой единицы становится *<= 0,* то она покидает поле боя. Проигравшей считается та армия, у которой боевые единицы закончатся раньше. Также с некоторой вероятностью к каждой армии может прийти подкрепление из случайного числа боевых единиц. Задача программы — вывести номер победителя при заранее известных характеристиках танков, пехоты и самолетов каждой армии, но случайном выборе боевых единиц и целей при каждом ходе.

Данная задача является хорошей демонстрацией работы языка, так как она использует большинство языковых конструкций, а боевые юниты, представляемые объектами, и армии — массивами, активно задействуют подсистему сборки мусора.

Многочисленные прогоны задачи не выявили нарушений в логике работы интерпретатора и показали, что его производительность сравнима с производительностью Python 3.

Заключение

В ходе написания курсовой работы был разработан интерпретатор скриптового языка программирования GCI, изучены современные подходы к автоматическому управлению памятью и интерпретации программ. Получен опыт написания большого проекта на чистом C без каких-либо зависимостей.

В силу высокого качества и структурированности исходного кода интерпретатор GCI является легко расширяемым и модифицируемым. Это позволяет использовать его как основу для диплома и других курсовых проектов. В качестве его развития можно, например, реализовать новые алгоритмы сборки мусора, поддержку зеленых потоков или JIT-компиляцию.

Список литературы

1. A. Aho, M. Lam, R. Sethi, D. Ullman. Compilers: principles, techniques and tools (2nd edition) // 2016
2. R. Jones, A. Hosking, E. Moss. The Garbage Collection Handbook // 2012
3. GCC 3.4 Release Series: Changes, New Features, and Fixes // URL: https://gcc.gnu.org/gcc-3.4/changes.html (дата обращения: 03.12.2019)
4. GCC 4.1 Release Series: Changes, New Features, and Fixes // URL: https://gcc.gnu.org/gcc-4.1/changes.html (дата обращения: 03.12.2019)
5. С. Скоробогатов, А. Коновалов. ИУ-9. Курс лекций по компиляторам // 2019
6. B. Nystrom. Crafting Interpreters // URL: https://craftinginterpreters.com/ (дата обращения: 27.11.2019)
7. R. Spivak Let`s Build A Simple Interpreter // URL: https://ruslanspivak.com/lsbasi-part1/ (дата обращения: 03.12.2019)
8. Yu. Shi, D. Gregg, A. Beatty. Virtual Machine Showdown: Stack Versus Registers // June 11-12, 2005, Chicago, Illinois, USA
9. Java Constant Pool // URL: https://docs.oracle.com/javase/specs/jvms/se9/html/jvms-4.html (дата обращения: 03.12.2019)
10. Valgrind // URL: http://valgrind.org/ (дата обращения: 03.12.2019)
11. C++11 memory // URL: https://ru.cppreference.com/w/cpp/memory (дата обращения: 03.12.2019)
12. C. Cheney. A Nonrecursive List Compacting Algorithm". Communications of the ACM. // 1970

Приложение 1. Таблица лексем

Лексема	Регулярное выражение
FUNCTION	function
LET	let
IF	if
ELSE	else
WHILE	while
BREAK	break
CONTINUE	continue
APPEND	append
DELETE	delete
RETURN	return
IDENT	[a-z]([a-z]|[0-9])*
OR	||
AND	&&
EQEQ	==
NEQ	!=
LT	<
GT	>
LE	<=
GE	>=
EQ	=
PLUS	+
MINUS	-
MUL	*
DIV	/
MOD	%
LPAREN	(
RPAREN	)
NUMBER	[0-9]*(.[0-9]*)?
LBRACKET	[
RBRACKET	]
LBRACE	{
RBRACE	}
COMMA	,

Лексема	Регулярное выражение
SEMI	;
DOT	.
COLON	:

Приложение 2. Грамматика языка

Правило	Вывод
unit	function_decl*
function_decl	FUNCTION IDENT LPAREN formal_parameters_list RPAREN body | FUNCTION IDENT LPAREN RPAREN body
formal_parameters_list	IDENT (COMMA IDENT)* RPAREN
body	LBRACE statement* RBRACE
statement	decl_statement | assign_statement | function_call_statement | if_statement | while_statement | break_statement | continue_statement | return_statement
decl_statement	LET IDENT EQ assignment_expr SEMI
assign_statement	variable EQ assignment_expr SEMI
function_call_statement	function_call SEMI
if_statement	IF LPAREN logical_or_expr RPAREN body ELSE if_statement | IF LPAREN logical_or_expr RPAREN body ELSE body | IF LPAREN logical_or_expr RPAREN body
while_statement	WHILE LPAREN logical_or_expr RPAREN body
break_statement	BREAK SEMI
continue_statement	CONTINUE SEMI
return_statement	RETURN assignment_expr SEMI | RETURN SEMI
variable	IDENT (variable_part)*
variable_part	DOT IDENT | LBRACKET logical_or_expr RBRACKET
assignment_expr	object_literal | array_literal | logical_or_expr
object_literal	LBRACE properties_list RBRACE | LBRACE RBRACE
array_literal	LBRACKET args_list RBRACKET | LBRACKET RBRACKET
properties_list	property (COMMA property)*
property	IDENT COLON assignment_expr
logical_or_expr	logical_and_expr (OR logical_and_expr)*
logical_and_expr	eq_expr (AND eq_expr)*
eq_expr	relational_expr (eq_op relationa_expr)?
eq_op	EQEQ | NEQ
relational_expr	additive_expr (relational_op additive_expr)?
relational_op	LT | GT | LE | GE

Правило	Вывод
additive_expr	multiplicative_expr (additive_op multiplicative_expr)*
additive_op	PLUS | MINUS
multiplicative_expr	left_unary_expr (multiplicative_op left_unary_expr)*
multiplicative_op	MUL | DIV | MOD
left_unary_expr	left_unary_op primary_expr | primary_expr
left_unary_op	PLUS | MINUS
primary_expr	function_call | variable | NUMBER | LPAREN logical_or_expr RPAREN
function_call	IDENT LPAREN args_list RPAREN | IDENT LPAREN RPAREN
args_list	assignment_expr (COMMA assignment_expr)*

Приложение 3. Команды виртуальной машины

Команда	Число параметров	Число стековых аргументов
POP	0	1
CONSTANT	1	0
CREATE_LOCAL	1	0
GET_LOCAL	1	0
SET_LOCAL	1	1
CREATE_OBJ	1	>= 0
INIT_OBJ_PROP	1	1
CREATE_ARR	1	>= 0
APPEND	0	2
DELETE	0	2
GET_HEAP	2 + n * 2	0
SET_HEAP	2 + n * 2	0
LOGICAL_OR	0	2
LOGICAL_AND	0	2
EQEQ	0	2
NEQ	0	2
LT	0	2
GT	0	2
LE	0	2
GE	0	2
PLUS	0	2
MINUS	0	2
MUL	0	2
DIV	0	2
MOD	0	2
NEGATE	0	1
JUMP_IF_FALSE	1	1
JUMP	1	0
RETURN	0	>= 0