Компьютерные инструменты в образовании, 2017
№ 2: 5–15
УДК: 004.4'423, 004.85
http://ipo.spb.ru/journal

инженерия программного обеспечения

MINIVALGRIND: ПРОСТОЙ ДЕТЕКТОР УТЕЧЕК ПАМЯТИ

Мухин А.М.[1], Чернышев Г.А.[1]

[1]СПбГУ, Санкт-Петербург, Россия

Аннотация

В данной работе описывается учебный проект по созданию детектора утечек памяти для интерпретируемого языка. Этот проект ориентирован на студента или группу студентов второго курса. Он разрабатывался в качестве альтернативной формы отчетности по практическим занятиям курса «программирование».

В работе описывается постановка задачи, общая схема проекта, обсуждаются подзадачи и методы их решения. Рассматриваются все стадии проекта: разработка языка, построение интерпретатора, реализация детектора утечек памяти. Кроме того, в статье описывается опыт применения данного проекта в учебном процессе, полученный в осеннем семестре 2016 года. Код проекта доступен на сайте GitHub.

Ключевые слова: *обучение программированию, интерпретатор, поиск утечек памяти.*

Цитирование: Мухин А.М., Чернышев Г.А. MiniValgrind: простой детектор утечек памяти // Компьютерные инструменты в образовании. 2017. № 2. С. 5–15.

1. ВВЕДЕНИЕ

Стандартный подход к организации практических занятий по курсу «Программирование» предполагает выполнение студентом некоторого набора достаточно маленьких и не связанных друг с другом заданий в течение семестра.

В этой работе мы рассматриваем альтернативный подход, когда вместо маленьких заданий студентам с высоким уровнем подготовки предлагается одно большое.

Такой подход имеет серьезные преимущества:

1. В рамках подхода можно давать более приближенные к реальности и, что самое важное, более творческие, наукоемкие задания. Такого рода задания полезны и интересны сильным студентам.
2. Подход позволяет попробовать на практике инструменты индустриального программирования, такие как системы контроля версий, продвинутые отладчики, профилировщики. Кроме того, это позволяет на «живом» проекте показать полезность аудита кода (code review), модульного тестирования (unit testing) и других индустриальных практик.
3. Наконец, результат большого проекта может быть использован для наполнения портфолио студента. В настоящее время многие индустриальные компании зачастую хотят видеть код, написанный кандидатом, еще до момента выдачи тестового задания. Наличие ссылки на открытый онлайн репозиторий (такой как GitHub)

стало де-факто стандартным требованием к резюме кандидата. Поэтому для студента, обучающегося по специальности, связанной с программированием, очень важно начать формирование портфолио как можно раньше.

Однако, присутствуют и сложности, которые надо учитывать:

1. Необходимо сделать так, чтобы задание было адекватной заменой стандартному подходу. Для его выполнения должно быть достаточно всё того же минимального набора знаний, описанного учебной программой, — конструкций языка программирования, структур данных, алгоритмов.
2. Необходимо следить за тем, чтобы изначально выданное задание было по силам студентам, а также следует оперативно решать возникающие трудности. Первая часть легко решается с помощью придания данному заданию статуса «по выбору» и предварительной оценки уровня студентов. Для решения второй трудности следует разбить проект на несколько этапов и по итогу выполнения устраивать очную «приемку» результатов. В случае необходимости следует провести работу по выявлению и устранению возникающих проблем, например, подсказать книгу или документацию, помочь разобраться со сложной ошибкой и т. д.
3. Кроме того, данное задание предполагает большую степень самостоятельности студентов, поэтому предварительный отбор исполнителей следует производить тщательнее.

В данной работе предлагается учебный проект по созданию детектора утечек памяти для интерпретируемого языка. В проект входит разработка грамматики языка программирования, построение его интерпретатора и, наконец, разработка динамического детектора утечек наподобие Valgrind [1]. Данный проект имеет достаточно широкий фокус. Во-первых, его выполнение позволяет изучить теоретические основы построения интерпретаторов, а также дает возможность познакомиться с практическими средствами их реализации. Во-вторых, работа по данному проекту позволяет проработать простой динамический анализ кода.

Проект ориентирован на студента или группу студентов второго курса и разрабатывался в качестве альтернативной формы отчетности по практическим занятиям курса «программирование». Первый тестовый запуск проекта был осуществлен в осеннем семестре 2016 года. Его результаты (код интерпретатора и анализатора) доступны на сайте github.com[1].

Данная работа структурирована следующим образом. В секции 2 подробно рассматривается задача. Далее, в секции 3 описывается процесс разработки и тестирования языка, лексического и синтаксического анализатора, интерпретатора с возможностью поиска утечек памяти. В секции 4 подводятся итоги первого запуска данного учебного проекта. В секции 5 проводится обзор существующих работ, относящихся к нашей задаче.

2. ПОСТАНОВКА ЗАДАЧИ

Цель нашего проекта — разработка простого детектора утечек памяти для учебного интерпретируемого языка. Для достижения этой цели были поставлены следующие задачи:

1. Разработка языка программирования, его формализация посредством грамматики. Язык должен быть минималистичным и позволять реализацию за один семестр.

[1] https://github.com/ortem/MiniValgrind

При этом он должен быть достаточно выразительным чтобы была необходимость в динамическом анализаторе. Например, в нем должна быть работа с динамической памятью, должны быть функции.
2. Разработка интерпретатора для созданного на первом шаге языка программирования. Предполагалось, что на данном шаге студенты познакомятся как с теоретическими основаниями [2], так и со средствами Bison и Flex [3].
3. Разработка модуля для простого динамического анализа. За образец функциональности брался детектор утечек памяти Valgrind [1] (уровень, разумеется, недостижимый в рамках односеместрового учебного проекта).

Данный проект задумывался как возможность для сильных студентов получить дополнительные знания в ходе занятий. Существует [4] множество вариантов обеспечения этого: соревнования в рамках учебного процесса [5] и вне его [6].

При решении данной задачи студент может применить на практике навыки, приобретенные на занятиях по программированию и дискретной математике. Например, применить базовые структуры данных (списки, деревья) и классические алгоритмы их обработки. Предполагается, что студенты прослушали два семестра дисциплин теории и практики по программированию.

Как и другие крупные (по сравнению со стандартными задачами) учебные проекты, данная задача предполагает использование некоторых индустриальных средств: системы контроля версий (Git), тестирования (разработка собственных тестов и, возможно, использование специальных средств для модульного тестирования), отладки (GNU Debugger, Valgrind).

Для ознакомления с новой предметной областью и технологиями студенту придется пользоваться различной специализированной литературой, и это развивает способность искать нужную информацию и самостоятельно разбираться в новой области.

Благодаря большому объему и сложности, решение данной задачи можно включить в портфолио студента для будущего трудоустройства.

3. РАЗРАБОТКА

В ходе разработки было принято решение разбить исходную задачу на следующие этапы:

1. Разработка простого языка программирования.
2. Создание лексического и синтаксического анализатора.
3. Создание интерпретатора.
4. Разработка классов исключений для поиска семантических ошибок.
5. Добавление методов для учета выделенной динамической памяти.

3.1. Описание Языка MiniC

В рамках проекта был разработан язык MiniC. Для простоты и удобства программирования MiniC наследует многие синтаксические особенности C. В исходной постановке задачи не было формального описания языка; студент самостоятельно продумывает грамматику и возможности языка. Основным требованием к языку являлось наличие указателей и возможность работы с динамической памятью. Возможности будущего языка расширялись и изменялись в процессе проектирования: например, в первом рабочем

прототипе языка не было функций и глобальных переменных. Рассмотрим этот язык подробнее.

Типы данных. MiniC поддерживает три типа данных: целое число (**int**), массив из целых чисел (**arr**) и указатель на целое число (**ptr**). Для поставленной задачи этих типов вполне достаточно, но при желании несложно будет добавить в язык поддержку чисел с плавающей точкой или строк. Данный язык не поддерживает тип boolean, а использует вместо него целочисленные значения аналогично языку C. Кроме того, не реализована арифметика указателей.

Операторы. В языке присутствуют арифметические операторы (+, −, *, /, %, ++, −−), логические операторы (!, &&, ||), операторы сравнения (<, <=, ==, >=, >, !=) и оператор присваивания (=).

Управляющие конструкции. В ходе обсуждения с преподавателем были приняты следующие стандартные конструкции: ветвление (**if**−**else**) и циклы (**for**, **while**). Конструкции **switch**−**case**, **break**, **continue**, **goto** были исключены для упрощения разработки.

Функции. В исходной версии задачи не требовалась поддержка функций в языке. Но в процессе разработки интерпретатора было решено расширить язык поддержкой функций, так как значительная часть ошибок при работе с динамической памятью связана с их использованием. Функции в MiniC могут возвращать значения всех трёх типов (**int**, **ptr**, **arr**), могут вызывать другие функции и быть рекурсивными.

Управление памятью. Как упоминалось выше, возможность динамического управления памятью являлась ключевой для языка. Поэтому MiniC поддерживает функции **malloc** и **free**, аналогичные по синтаксису и семантике стандартным функциям языка C.

Глобальные переменные. Возможность добавления глобальных переменных может существенно облегчить разработку программ, причём добавить поддержку глобальных переменных в язык несложно. Поэтому эта возможность была реализована в MiniC. Глобальные переменные создаются посредством добавления в программу специального блока **global**. Все инструкции, содержащиеся в этом блоке, выполняются перед входом в функцию main, а переменные, определенные в этом блоке, добавляются в область видимости всех функций программы.

Листинг 1. Заполнение массива заданным значением

```
#begin
arr fillArr(int size, int num) {
    ptr p = malloc(size);
    for (int i = 0; i < size; i++;) {
        p[i] = num;
    }
    return p;
}
int main() {
    int s = 5;
    int n = 3;
    arr a[5];
    a = fillArr(s, n);
    print(a); // "[3, 3, 3, 3, 3]"
    free(a);
    return 0;
}
#end
```

Программа, написанная на MiniC, начинается со служебного слова **#begin** и заканчивается словом **#end**. Программы состоят из функций и, опционально, блока глобальных переменных. Чтобы продемонстрировать общие черты языка, рассмотрим пример программы, выполняющей заполнение массива заданным значением, изображенный на Листинге 1. В данной программе внутри функции main происходит вызов функции fillArr и присваивание возвращаемого ею значения переменной типа **arr**. Следует обратить внимание на то, что функция fillArr должна возвращать значение типа **arr**, но возвращает **ptr**. Это не приведет к ошибке, так как массивы и указатели взаимозаменяемы, как и в языке C. Из-за того, что размер массива (**arr**) обязан быть статическим, в теле этой функции приходится пользоваться типом **ptr** и функцией **malloc**, а после использования этого массива вызывать **free**.

3.2. Лексический и Синтаксический Анализ

Вместо реализации собственного лексического и синтаксического анализатора решено было использовать их генераторы. Конечно, создание собственного анализатора — хорошее упражнение, но это отняло бы слишком много времени; кроме того, для первичного ознакомления с предметной областью полезно изучить готовые классические решения.

В качестве таких генераторов были выбраны Flex (генератор лексических анализаторов) и Bison (генератор синтаксических анализаторов). Flex принимает на вход правила выделения лексем и выдает код анализатора на языке C. Далее Bison может использовать этот анализатор и поданные на вход правила грамматики языка, чтобы сгенерировать синтаксический анализатор для этого языка.

Создание правил для Flex не вызвало трудностей, но составление грамматических правил языка для Bison оказалось сложной задачей. Для того чтобы изучить Bison на достаточном уровне, студенту потребовалось чтение [3].

3.3. Интерпретатор

Интерпретацию простейшего языка можно выполнять непосредственно во время работы синтаксического анализатора. Однако для дальнейшего анализа программы удобно сначала построить синтаксическое дерево, а потом производить необходимые проверки и запуск программы. Узлами этого дерева будут указатели типов, соответствующих различным языковым конструкциям.

Основными типами являются Statement (инструкция) и Expression (выражение). От Statement наследуются классы операторов ветвления, циклов и т.д. «Инструкции» содержат метод run (выполнение инструкции). От Expression наследуются унарное, бинарное выражения, вызов функции, обращение к массиву по индексу, числовое значение, имя переменной. «Выражения» содержат метод eval (вычисление выражения), результатом которого является объект типа Var («переменная»).

Класс Var служит для представления переменной из программы объектом, инкапсулирующим тип, значение и индикатор инициализации переменной. Он состоит из следующих полей:

- тип переменной (**int**, **ptr**, **arr**),
- три поля для значения (intVal, ptrVal, arrVal),
- размер массива (не используется для переменных типа **int**),

– индикатор (инициализирована ли переменная) и массив индикаторов (инициализирован ли элемент массива).

Для функций, определенных в программе, предназначен класс Function, содержащий сигнатуру и тело функции. Кроме того, определен класс-синглтон Program, состоящий из основных элементов программы: списка функций, глобальных переменных и списка указателей на блоки выделенной памяти (последнее нужно для поиска утечек памяти).

Теперь синтаксический анализатор, обрабатывая текст, создает объекты-операторы и объекты-выражения и, храня в стеке указатели на них, составляет «дерево разбора», в корне которого находится объект Program.

Для отслеживания состояния каждой переменной, объявленной в блоке программы (блок — это часть программы, заключенная в фигурные скобки), предназначен ассоциативный массив map<string, Var *> vars. При каждом использовании переменной в программе производится рекурсивный поиск этой переменной в блоках программы (при помощи vars.find()): сначала в текущем блоке, затем вверх по вложенности блоков.

Используя такой анализатор, несложно создать интерпретатор MiniC: необходимо зайти в функцию main и вызывать метод run для каждой инструкции.

3.4. Анализ Программ и Поиск Ошибок

Перейдем к описанию процесса проверки исходного кода на наличие ошибок. Одной из самых распространенных ошибок в программах является выход за границы массива. При описанной выше архитектуре искать такие ошибки довольно просто: при попытке чтения/записи переменной по индексу необходимо произвести проверку на соответствие индекса допустимому диапазону (этот диапазон хранится как поле в каждой переменной). Кроме того, несложно устанавливать и использование неинициализированных переменных: при попытке чтения переменной необходимо проверить индикатор инициализации (при чтении по индексу — соответствующий элемент массива индикаторов). Для уведомления пользователя об этих (и некоторых других) ошибках были разработаны соответствующие классы исключений.

Помимо поиска синтаксических и семантических ошибок, возможен также поиск утечек памяти в программе. За это ответственен модуль динамического анализа. Рассмотрим пример, изображенный на Листинге 2.

Листинг 2. Взаимозаменяемость типов **ptr** и **arr**

```
1 ptr p = malloc(3);
2 p[0] = 0; p[1] = 1; p[2] = 2;
3 arr A[3];
4 A = p;     // A = [0, 1, 2]
5 A[1] = 8; // p = [0, 8, 2]
6 p[2] = 5; // A = [0, 8, 5]
```

Как уже упоминалось, в MiniC массив ведет себя аналогично указателю. Поэтому память, которую используют переменные p и A, должна быть одной и той же. Для этого в реализации класса Var было решено использовать умные указатели, а именно shared pointer.

При вызове функции **malloc** указатель на выделенную память сохраняется в специальном списке внутри объекта Program, а при вызове **free** — удаляется из этого списка. По завершении исполнения входной программы MiniValgrind, используя этот список, выводит информацию о количестве утечек и размере неосвобожденной памяти.

3.5. Тестирование

Для тестирования MiniValgrind было создано около 40 небольших программ на MiniC, реализующих классические алгоритмы, в том числе: бинарный поиск, сортировка пузырьком, сортировка вставками, сортировка слиянием, алгоритм Евклида, решето Эратосфена. В некоторых программах были намеренно допущены различного рода ошибки, которые MiniValgrind успешно отыскал. Кроме того, код MiniValgrind был протестирован с помощью оригинального Valgrind. Memcheck обнаружил несколько утечек памяти, которые позже были устранены.

Листинг 3. Рекурсивная сортировка слиянием

```
#begin
global {
    arr A[5];
    A[0] = 2; A[1] = 5; A[2] = 6; A[3] = 3; A[4] = 1;
    arr B[5];
}

int merging(int low, int mid, int high) {
   int l1 = low;
   int l2 = mid + 1;
   int i;
   for (i = low; (l1 <= mid) && (l2 <= high); i++;) {
      if (A[l1] <= A[l2]) {
         B[i] = A[l1++];
      }
      else {
         B[i] = A[l2++];
      }
   }
   while (l1 <= mid) {
      B[i++] = A[l1++];
   }
   while (l2 <= high) {
      B[i++] = A[l2++];
   }
   for (i = low; i <= high; i++;) {
      A[i] = B[i];
   }
}
int sort(int low, int high) {
   int mid;
   if (low < high) {
      mid = (low + high) / 2;
      sort(low, mid);
      sort(mid + 1, high);
      merging(low, mid, high);
   }
}

int main() {
    print(A);
    sort(0, 4);
    print(B);
}
#end
```

Программа, приведённая в Листинге 3, реализует рекурсивную сортировку слиянием. Заметим несколько возможностей MiniC, использованных в этой программе. Во-первых, в блоке **global** можно не только объявлять переменные, но и инициализировать их. Во-вторых, можно не возвращать значения из функций (то есть не добавлять инструкцию **return**), и это не приведет к ошибке в случае, когда возвращаемое значение не используется. Эта возможность позволяет не добавлять в язык тип **void**, а обходиться типом **int**, подобно ранним версиям C.

4. ИТОГИ ПЕРВОГО ЗАПУСКА

В результате первого запуска данного учебного проекта был создан описанный выше язык (MiniC) и интерпретатор этого языка с возможностью поиска утечек памяти (MiniValgrind). Задача была выполнена студентом Математико-механического факультета СПбГУ в течение трёх месяцев третьего семестра (точнее, проект выполнялся по 3–4 часа в неделю в течение трёх месяцев). Раз в неделю в рамках занятий по программированию проводились встречи и обсуждение текущих результатов с преподавателем.

Исходный код правил и грамматики для Flex и Bison занимает около 170 строк. Исходный код интерпретатора, реализованного на языке C++, занимает около 1500 строк. Во время выполнения было сделано 36 коммитов в git-репозиторий. На проектирование языка и его грамматики было в сумме потрачено около месяца; остальные два месяца были потрачены на разработку и тестирование интерпретатора.

5. ОБЗОР СУЩЕСТВУЮЩИХ РАБОТ

Построение учебных компиляторов и интерпретаторов — популярный вариант как для проведения специализированных курсов по трансляции, так и для проведения вводных курсов по программированию. Кроме того, разработка и использование упрощенных языков программирования — также весьма часто используемый прием в преподавании. На эти две темы опубликовано множество работ. Наш подход совмещает оба аспекта, поэтому мы рассмотрим представителей обоих классов работ.

Автор статьи [7] описывает задачу для студентов, слушающих семестровый курс по построению компиляторов. Студентам предлагается разработать собственный интерпретатор языка XQuery. В процессе авторы сталкиваются с несколькими проблемами. Во-первых, теория построения синтаксических анализаторов и компиляторов слишком обширна, поэтому пришлось сократить изучение теории и разрешить студентам использовать средства для автоматической генерации лексического и синтаксического анализаторов (JFlex и CUP Parser Generator). Кроме того, для упрощения задачи было решено делать интерпретатор подмножества XQuery, не поддерживающий некоторые сложные конструкции этого языка.

В работе [8] студентам было предложено реализовать транслятор простого Java-подобного языка в JVM-ассемблер Jasmin. Основное отличие этого эксперимента от предыдущего заключается в том, что лексический и синтаксический анализатор необходимо было написать самому «с нуля». В качестве алгоритма синтаксического анализа был выбран метод рекурсивного спуска. В результате студенты потратили много времени на создание синтаксического анализатора, и лишь 40% из них закончили проект за один семестр.

Исследование [9] рассказывает о разработке простого языка для обучения студентов технике трансляции. Автор решил не прибегать к использованию генераторов лексических и синтаксических анализаторов, чтобы студенты могли изучить компилятор более детально. Для того чтобы код компилятора получился более коротким и понятным, было решено максимально упростить язык: использовать только один тип данных, не использовать указатели и массивы, исключить ручную работу с памятью. Опыт автора оказался позитивным: большинство студентов, изучивших учебный компилятор, смогли разработать собственные анализаторы и оптимизаторы кода для него.

Автор [10] считает, что задача создания компилятора для подмножества C «с нуля» слишком сложна для студентов. Поэтому он предложил студентам реализовать компилятор простого предметно-ориентированного языка для разработки небольших игр, используя уже упомянутые средства Flex и Bison.

CLIP [11] — интерпретатор простого C++-подобного языка. Новичкам тяжело изучать C++: помимо сложности самого языка, преградой является необходимость использования компилятора, недостаточно дружественного для пользователя. Поэтому автор решил создать интерпретатор для простого C++-подобного языка, который был бы прост в использовании и печатал бы понятные новичку сообщения об ошибках. Полученный учебный язык не содержит множества сложных в освоении возможностей C++: классы, исключения, работу с файлами, шаблоны и др. Автор считает, что такой язык и интерактивный интерпретатор для него упростят понимание базовых концепций программирования для студентов. Данный проект имеет много общего с нашим, так как MiniC и интерпретатор для него после некоторой доработки могут быть использованы для обучения новичков: благодаря тому, что интерпретатор включает в себя динамический анализатор кода, многие распространенные ошибки новичков (использование неинициализированных переменных, утечки памяти, выход за границы массива) будут обнаружены, и MiniValgrind напечатает простые и понятные сообщения об этих ошибках.

В статье [12] А.Н. Терехов описывает создание компилятора для подмножества C — РуСи. Этот язык спроектирован прежде всего для обучения школьников, поэтому он поддерживает написание ключевых слов и идентификаторов на кириллице. Компилятор специально создавался максимально дружественным к пользователю, чтобы школьники могли легко разобраться в ошибках в своём коде. Основное различие РуСи и MiniValgrind состоит в подходах (компилятор или интерпретатор). Конечно, интерпретаторы в основном проигрывают компиляторам, но, как упоминалось в предыдущей статье, применительно к обучению новичков интерпретатор может давать некоторые преимущества (например, возможность поиска простых утечек памяти).

Наш подход качественно отличается от вышеупомянутых прежде всего совмещением разработки как интерпретатора, так динамического анализатора. Кроме того, в некоторых рассмотренных работах упрощенный язык (и соответствующее программное обеспечение) разрабатывался преподавателями. В нашем же случае, язык и интерпретатор создавался именно студентом, хоть и под руководством преподавателя.

Кроме того, заметим, что в тех работах язык был создан с целью облегчить вхождение в предметную область начинающим студентам. Наш язык также может быть ограниченно использован в обучении. Основным препятствием на данный момент является отсутствие дружественных к пользователю сообщений об ошибках. Однако у нас имеется простой динамический анализатор. Он позволит с легкостью находить обычно трудно находимые ошибки работы с памятью. Улучшение же сообщений об ошибках может быть частью будущей работы.

Если же рассматривать нашу работу как задание в рамках курса, то необходимо заметить, что у нас несколько другая ориентация задания. Оно рассчитано на сильных студентов младших курсов и вводный курс программирования. В то же время, курсы по трансляции обычно рассчитаны на подготовленных старшекурсников. Поэтому наш подход более упрощенный и, например, не включает в себя разработку синтаксического анализатора методом рекурсивного спуска, а использует готовый инструмент.

6. ЗАКЛЮЧЕНИЕ

В данной работе мы описали альтернативное задание в рамках курса «практикум по программированию» для студентов второго курса. Мы обсудили плюсы и минусы такого подхода, а также разобрали набор выработанных решений. Кроме того, мы обсудили итоги первого запуска данного проекта.

Список литературы

1. Valgrind Home. http://valgrind.org/ дата обращения: (16.03.2017).
2. *Ахо А., Лам М., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструментарий. М.: Вильямс, 2015.
3. *John Levine. Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009.
4. *Чернышев Г.А.* Опыт проведения практикумов по программированию на математико-механическом факультете СПбГУ // Компьютерные инструменты в образовании. 2015. № 6. С. 43–66.
5. *Balakina E., Chernishev G., Novikov B., Mukhin A., Pervakov G., Smirnov K., Smirnova A., Zakharov G.* Using Programming Contest in Teaching Introductory Programming Course: an Experience // In Procedings of the 9th IEEE International Conference on UbiMedia Computing. 2016. P. 272–277.
6. *Смирнов К.К., Чернышев Г.А.* Участие в соревновании ACM SIGMOD как возможность для студентов углубленно изучить некоторые аспекты баз данных и программной инженерии // Компьютерные инструменты в образовании. 2012. № 5. С. 25–32.
7. *Miner S., Pevzner T., Deutsch A., Baden S., Kube P.* Building an XQuery Interpreter in a Compiler Construction Course // SIGCSE Bull. 2005. № 37(1) P. 2–6.
8. *Sattar A., Lorenzen T.* Develop a Compiler in Java for a Compiler Design Course // SIGCSE Bull. 2007. № 39(2). P. 80–82.
9. *Baldwin D.* A Compiler for Teaching About Compilers // SIGCSE Bull. 2003. № 35(1). P. 220–223.
10. *Henry T.R.* Teaching Compiler Construction Using a Domain Specific Language // SIGCSE Bull. 2005. № 37(1). P. 7–11.
11. *Luoma H., Lahtinen E., J'arvinen H.-M.* CLIP, a Command Line Interpreter for a Subset of C++ // In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research. Darlinghurst, Australia. 2007. Vol. 88. P. 199–202.
12. *Терехов А.Н.* Инструментальное средство обучения программированию и технике трансляции // Компьютерные инструменты в образовании. 2016. № 1. С. 36–47.

Поступила в редакцию 01.02.2017, окончательный вариант — 30.03.2017.

Computer tools in education, 2017
№ 2: 5–15
http://ipo.spb.ru/journal

MINIVALGRIND: SIMPLE MEMORY LEAK DETECTOR

Mukhin A.M.[1], Chernishev G.A.[1]

[1]SPbSU, Saint-Petersburg, Russia

Abstract

In this paper, we describe an educational project for creating a memory leak detector for an interpreted language. This project is intended for a student or a group of second-year students. It was developed as an alternative form of classroom assignment in an introductory programming course.

The statement of the problem, and the general outline of the project are described in the paper; the subtasks and methods for their solution are also discussed. All stages of the project are considered: language development, interpreter construction, implementation of the memory leak detector. In addition, the experience — obtained in the fall semester of 2016 — of applying this project in the educational process is described in the article. The project code is available from GitHub.

Keywords: *introductory programming course, interpreter, memory leak detection.*

Citation: A.M. Mukhin and G.A. Chernishev, “MiniValgrind: prostoi detektor utechek pamyati” [MiniValgrind: Simple Memory Leak Detector], *Computer tools in education*, no. 2, pp. 5–15, 2017 (in Russian).

Received 01.02.2017, the final version — 30.03.2017.

Artem M. Mukhin, second-year undergraduate student, Saint-Petersburg State University, artem.m.mukhin@gmail.com
George A. Chernishev, assistant professor of Department of Analytical Information Systems, Saint-Petersburg State University; 198504, Saint-Petersburg, Peterhof, Universitetsky pr. 28, Department of Analytical Information Systems, chernishev@gmail.com

**Мухин Артем Михайлович,
студент 2-го курса математико-механического факультета СПбГУ,
artem.m.mukhin@gmail.com**

**Чернышев Георгий Алексеевич,
ассистент кафедры информационно-аналитических систем СПбГУ;
198504, Санкт-Петербург, Петергоф,
Университетский пр. 28, кафедра информационно-аналитических систем,
chernishev@gmail.com**

© Наши авторы, 2017.
Our authors, 2017.