МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«Национальный исследовательский ядерный университет «МИФИ»
(НИЯУ МИФИ)
КАФЕДРА ИНФОРМАЦИОННЫХ СИСТЕМ И ТЕХНОЛОГИЙ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»
на тему «Расширение языков стекового калькулятора и интерпретатора арифметических выражений операцией побитового сдвига. Вычисление периметра и площади части выпуклой оболочки, расположенной в верхней полуплоскости. Нахождение суммы длин полностью видимых рёбер полиэдра»

Группа	К04-361
Студент	А.Ю. Шедько
Руководитель работы к.ф.-м.н., доцент	Е.А. Роганов

Москва 2016

Аннотация

Работа посвящена модификации проектов «Компилятор формул», «Интерпретатор арифметических выражений», «Выпуклая оболочка» и «Изображение проекции полиэдра». В первом из этих проектов решалась задача расширения языков стекового калькулятора и итерпретатора арифметических выражений опрацией побитового сдвига. Модификация второго пректа требовала ... В проекте «Выпуклая оболочка» вычислялась ... В последнем из проектов определялась ...

Содержание

1. Введение

В проектах «Компилятор формул» и «Интерпретатор арифметических выражений» были решены задачи расширения языков стекового калькулятора и интерпретатора арифметических выражений операциями побитового сдвига. Применена структура данных «хэш-таблица».Решение задачи требовало представления о формальных грамматиках, основы ООП и знания языка Ruby.

Проект «Выпуклая оболочка»[1] решает задачу индуктивного перевычисления выпуклой оболочки последовательно поступающих точек плоскости и таких её характеристик, как периметр и площадь. Целью данной работы является индуктивное вычисление расстояние до наперёд заданного стандартного прямоугольника и количества острых улов выпуклой оболочки. Применено два специфических алгоритма, значительно ускоряющих вычисление расстояние от прямоугольника до отрезка [2] и проверку пересечения прямоугольника и отрезка (алгоритм Лианга-Барски) [3]. Решение этой задачи требует знания теории индуктивных функций, основ аналитической геометрии и векторной алгебры и языка Ruby [4].

Проект «Изображение проекции полиэдра» [5] — пример классической задачи, для успешного решения которой необходимо знакомство с основами вычислительной геометрии. Задачей, решаемой в данной работе, является модификация эталонного проекта с целью определения суммы длин полностью видимых рёбер заданного полиэдра. Для этого необходимы хорошее понимание ряда разделов аналитической геометрии и векторной алгебры, основ объектно-ориентированного программирования и языка Ruby.

Для подготовки пояснительной записки необходимо знакомство с программой компьютерной вёрстки LaTeX [6], умение набирать математические формулы [7] и включать в документ графические изображения и исходные коды программ.

Общее количество строк в рассмотренных проектах составляет около 190, из которых 28 были изменены или добавлены автором в процессе работы над задачами модификации.

2. Модификация проекта «Компилятор формул»

Постановка задачи:

В предположении, что язык стекового калькулятора расширен операциями L (left) и R (right), реализующими побитовый сдвиг влево и вправо соответственно, компилировать формулы, содержащие операции << и >>.

Теоретические аспекты:

С формальной точки зрения компилятор представляет собой программную реализацию некоторой функции $\tau\colon L_1 \to L_2$, действующей из множества цепочек одного языка L_1 в множество цепочек другого L_2 таким образом, что $\forall\omega \in L_1$ *семантика цепочек* ω *и* $\tau(\omega) \in L_2$ *совпадает.*

Для решения задачи необходимо задать грамматики, описывающие языки стекового калькулятора и компилятора, соответственно G_0 и G_s:

G_0:

S →	F	|	SRF	|	SLF	
F →	T	|	F+T	|	F-T	
T →	M	|	T*M	|	T/M	
M →	(S)	|	V			
V →	a	|	b	|	...	| z

G_s:

$$e \to e\ e\ + \mid e\ e\ - \mid e\ e\ * \mid e\ e\ / \mid e\ e\ >> \mid e\ e\ << \mid a \mid b \mid \cdots \mid z$$

где L, R соответствуют `<<`, `>>`, т. е. сдвигу влево и вправо, а S — классу сдвигаемых выражений (ассоциативность аналогична `+` и `-`).

Нужный нам компилятор τ представляет собой программную реализацию отображения из множества цепочек языка $L(G_0)$ в множество цепочек языка $L(G_s)$ По этой причине его можно рассматривать, как функцию на пространстве последовательностей. Легко понять, что эта функция не индуктивна.

Построим индуктивное расширение функции, чтобы с его помощью реализовать однопроходный алгоритм, осуществляющий нужный нам перевод. Заметим, что любую правильную формулу можно откомпилировать с соблюдением следующих двух условий:

- переменные в выходной цепочке (программе для стекового калькулятора) будут идти в том же порядке, что и переменные в исходной формуле;
- все операции в выходной цепочке будут расположены позже соответствующих им операций в исходной формуле.

Любую формулу можно компилировать так: встретив имя переменной, немедленно записывать его в массив, где мы будем накапливать результат компиляции, а встретив знак операции или скобку, записывать в этот массив те из предыдущих, но ещё не обработанных операций (будем называть их *отложенными*), которые выполнимы в данный момент, после чего «откладывать» и новый знак.

В качестве контейнера для хранения отложенных операций можно использовать стек. Этот стек и будет содержать ту дополнительную информацию, которая необходима для индуктивного перевычисления функции T осуществляющей компиляцию исходной формулы. Основная проблема — понять, что надо делать, когда в исходной формуле встречается очередная операция или скобка.

Таким образом, встретив в исходной формуле очередной знак операции или скобку, нужно иногда просто положить её в стек отложенных операций, а иногда — извлечь предварительно одну или несколько ранее отложенных операций и добавить их в массив результата. С точки зрения теории индуктивных функций необходимо построить индуктивное расширение функции T осуществляющей компиляцию исходной формулы, и найти для неё отображение G обеспечивающее её перевычисление при удлинении входной формулы.

Для построения индуктивной функции компиляции цепочки ω необходимо разделить символы на категории: `SYM_LEFT, SYM_RIGHT, SYM_OPER` и `SYM_OTHER` соответственно. правила построения таковы:

- открывающую скобку всегда помещать в стек, имя переменной — всегда сразу добавлять в массив результата компиляции;
- когда в формуле встречается закрывающая скобка нужно все операции, появившиеся в ней *после соответствующей ей открывающей скобки*, извлечь из

стека отложенных операций и добавить в массив результата (не забыть также извлечь открывающую скобку);
- операции и скобки помещаются в стек в соответствии с приоритетом соответствующей операции или скобке. Открывающие скобки имеют наименьший приоритет, закрывающие — наибольший; для операций он будет описан в дальнейшем.

Более подробно см. [8]

Применяемые структуры данных

Стек, используемый в эталонном проекте, не требует пояснений. Помимо стека используется константная хэш-таблица (в качестве отображения между множествами операций двух грамматик). Подробно с работой хеш-таблицы в языке Ruby можно ознакомиться в документации к языку [9] и на соответствующей странице Википедии.

Детали реализации

Отображение между множествами операций языков калькулятора и компилятора описывается данной таблицей.

```
CONV_TABLE = {">>" => "R",
              "<<" => "L",
               "+" => "+",
               "-" => "-",
               "/" => "/",
               "*" => "*"}
```

Приоритет операций задаётся следующим образом:

```
def priority(c)
  (c == '+' or c == '-') ? 1 : ( c=='L' or c=='R' )? 0 : 2
end
```

Этот фрагмент кода устанавливает наименьший приоритет для сдвигов (0) и наивысший для деления и умножения (2).

Наиболее значительным является изменение, касающееся обработки символов операций, а именно обработка символов >> и << так как они занимают два строковых символа, являясь одним логическим символом. Идея проста: хранить первый символ в обрабатываемой последовательности, соответствующий одной из операций, в специальной переменной `@op`, а при поступлении следующего символа, если последовательность составлена верно и этот символ совпадает с < или >, записывать его в `@op`. После этого, вне зависимости от содержимого `@op`, происходит стандартная процедура обработки отложенных операций с использованием значения `CONV_TABLE[@op]`, которое затем помещается в стек. После успешной обработки символа `@op` очищается.

Код, реализующий обработку символов операций:

```
def process_symbol(c)
  ...
  when SYM_OPER
    if ((c=='>'||c=='<')&&@op=="")
      @op=c
    else
      @op+=c
      process_suspended_operators(CONV_TABLE[@op])
      push(CONV_TABLE[@op])
      @op=""
  end
  ...
end
```

Таким образом в язык стекового компилятора добавлена операция побитового сдвига.

Возможные обобщения

Имеет смысл ввести в язык стекового компилятора также остальные побитовые операции (&, |, ^, ~), их логические аналоги (&&, |||).

3. Модификация проекта «Интерпретатор арифметических выражений»

Постановка задачи

Вычисляются значения выражений, содержащих битовые операции << и >>, приоритет первой из которых является минимальным, а второй — максимальным.

Теоретические аспекты

Если компилятор осуществляет перевод с одного языка на другой, то интерпретатор *вычисляет* значение арифметической формулы, в которой вместо имён переменных содержатся записанные тем или иным способом числа.

Грамматика G_0 из предыдущего пункта должна быть изменена для учёта максимального приоритета операции сдвига вправо:

G_0:

```
S_0 →    F    |  S_0LF
  F →    T    |   F+T   |  F-T
  T →    M    |   T*M   |  T/M
S_1 →    M    |  S_1RF
  M →  (S_0)  |    V
  V →    a    |    b    |  ...  |  z
```

`S_0` — класс сдвигаемых влево выражений;
`S_1` — класс сдвигаемых вправо выражений.

G_s можно оставить без изменений.

Класс `Calc`, реализующий алгоритм интерпретации арифметических выражений может быть построен при помощи уже реализованного класса `Compf`.

Интерпретатор выражений отличается от компилятора тем, что во вместо идентификаторов переменных во входной формуле стоят *числа* и получаемую выходную формулу необходимо *выполнять* (вместо её печати). Для того чтобы реализовать стековый калькулятор, естественно, необходим стек.

При появлении на входе цифры, будем помещать её в стек калькулятора `@s`, а при появлении операции доставать из стека её аргументы и помещать туда результат её выполнения. На конечном этапе на вершине стека будет находиться результат выполнения арифметического выражения.

Детали реализации

Здесь также как и в первом проекте применена хеш-таблица для отображения множества операций стекового компилятора в множество операций языка Ruby.

```
CONV_TABLE = {"R" => ">>",
              "L" => "<<",
              "+" => "+",
              "-" => "-",
              "/" => "/",
              "*" => "*"}
```

В соответствии с задачей, приоритет операций был изменён следующим образом:

```
def priority(c)
  (c == '+' or c == '-') ? 1 : c=='L' ? 0 : (c=='R') ? 3 : 2
end
```

Таблица 1. Приоритет операций

Операция	Приоритет
L	0
+, -	1
*, /	2
R	3

4. Модификация проекта «Выпуклая оболочка»

...

5. Модификация проекта «Изображение проекции полиэдра»

...

Список литературы и интернет-ресурсов

[1] `https://home.mephi.ru/files/2373/material_ici_toc.zip/index.html` — Описание проекта «Выпуклая оболочка».

[2] *Advances in Spatial and Temporal Databases: 9th International Symposium.* — SSTD 2005, Angra Dos Reis Brazil, August 22-24, 2005, Proceedings, C 333.

[3] Liang, Y.D., and Barsky, B., *A New Concept and Method for Line Clipping.* — ACM Transactions on Graphics, 3(1):1-22, January 1984.

[4] `http://ru.wikipedia.org/wiki/Ruby` — Википедия (свободная энциклопедия) о языке Ruby.

[5] `???` — Описание проекта «Изображение проекции полиэдра».

[6] С.М. Львовский. *Набор и вёрстка в системе LaTeX, 3-е изд., испр. и доп.* — М., МЦНМО, 2003. Доступны исходные тексты этой книги.

[7] D. E. Knuth. *The TeXbook.* — Addison-Wesley, 1984. Русский перевод: Дональд Е. Кнут. *Все про TeX.* — Протвино, РДТеХ, 1993.

[8] `https://home.mephi.ru/files/2077/material_ici_toc.zip/index.html` — Описание проекта *«Стековый компилятор формул»* Е.А. Роганов

[9] `http://ruby-doc.org/` — Документация языка Ruby