УДК 004.43

Библиотечная реализация стековой вычислительной модели на примере языка Joy

© 2013 г. А. В. Мошкина

a.v.moshkina@gmail.com

Кафедра алгоритмических языков

1 Введение

Перед началом решения некоторой задачи, программист естественным образом сталкивается с вопросом выбора языка программирования, на котором будет написан проект. Часто этот вопрос состоит не столько в выборе языка, сколько в выборе парадигмы программирования, в рамках которой удобнее всего решать поставленную задачу.

Понятие парадигмы программирования не имеет однозначного определения. В рамках данной работы под парадигмой программирования понимается система идей и понятий, которые определяют ход мышления программиста и стиль написания программ. Язык программирования не всегда однозначно определяет парадигму, используемую при работе с этим языком. Согласно [1], язык программирования может иметь различное отношение к конкретной парадигме: от навязывания применения парадигмы в рамках языка, до полного запрещения применения парадигмы. Поэтому языку могут соответствовать сразу несколько парадигм, в разной степени характерных ему.

Однако, парадигма, которую удобно использовать в одной задаче, не всегда может подойти для других, в то время как в рамках одного проекта могут решаться множество разнородных задач. Например, любое полноценное приложение должно

иметь графический интерфейс пользователя, который удобно проектировать в рамках объектно-ориентированной парадигмы. Почти каждое приложение осуществляет взаимодействие с базами данных, запросы к которым носят декларативный характер — в запросе описываются свойства требуемых данных, и никак не оговариваются способы их получения. Кроме того, в процессе обработки и манипулирования этими данными может быть удобно применение императивной, функциональной или, например, сценарной парадигм программирования.

Таким образом, мы видим, что потребности программиста не всегда ограничиваются одной парадигмой и, как следствие, одним языком программирования, поскольку использование одной или нескольких из требуемых парадигм в рамках языка может оказаться неудобным или даже невозможным. Решение этой проблемы лежит в поиске способов сочетания парадигм.

Одним из подходов к объединению нескольких языков программирования в рамках одного проекта является метод непосредственной интеграции [2], который является частным случаем метода библиотечного расширения существующего языка программирования.

Метод непосредственной интеграции основывается на выборе основного языка, на котором будет написан весь проект, а конструкции вспомогательных языков моделируются с помощью средств основного языка. Эти конструкции должны быть семантически близкими и синтаксически эквивалентными средствам моделируемого языка. Таким образом, метод позволяет не выходя за рамки выбранного базового языка программирования использовать парадигмы альтернативных языков.

Метод непосредственной интеграции успешно применялся для расширения языка C++ возможностями различных альтернативных языков [3–5], что позволяет говорить о перспективности этого подхода.

Работа посвящена реализации метода непосредственной интеграции для стекового функционального языка Joy.

2 Стековые языки программирования

Существует множество классификаций языков программирования, и одна из них — разделение языков на конкатенативные и аппликативные. Идея аппликативных языков заключается в том, что вычисление программы производится посредством применений функций к их аргументам. К этой группе можно отнести большое число языков общего назначения, например, C, Java, Python, Haskell и другие. В конкатенативных языках программирования вычисление программы состоит в композиции нескольких функций (то есть последовательного применения одной функции к результату вычисления предыдущей функции), причем все они оперируют с одним и тем же объектом данных, который передается от функции к функции. Чаще всего этим объектом данных становится стек. Также важно заметить, что описанная композиция функций реализуется простой конкатенацией программного кода этих функций. К семейству конкатенативных языков можно отнести такие языки программирования, как Forth [6], Joy [7], PostScript [8], Cat [9], Factor [10] и другие. Все они имеют сильное семейное сходство, однако значительно различаются по дизайну, реализации и назначению.

Для описания конкатенативных языков программирования также используется термин «стековые языки программирования», эти термины эквивалентны в практическом аспекте. Понятие стека является одним из фундаментальных в программировании, многие языки программирования используют стек во внутренней реализации. Любой язык, который поддерживает рекурсивные определения функций, использует реализованный в некотором виде стек вызовов, в котором хранится адрес возврата в функцию и ее локальное состояние. В большинстве случаев этот стек недоступен программисту в явном виде и является лишь частью реализации языка.

Особенностью стековых языков программирования является то, что в них присутствуют сразу несколько стеков. Один

из них — это стек вызовов (стек возвратов или действий), который используется для реализации рекурсии, но кроме него существует также стек данных (стек значений), предназначенный для передачи данных между функциями. Стек данных используется при программировании чаще, поэтому в случаях, когда это не вызывает недоразумений, он называется просто стеком.

Большинство языков общего применения являются аппликативными: центральной сущностью языка является некоторая форма функционального вызова. В процессе вызова функция применяется к своим аргументам, каждый из которых, в свою очередь, также является результатом функционального вызова, либо значением переменной или константы. В стековых языках программирования функциональный вызов в программе записывается только именем функции, параметры же являются неявными, функция извлекает их из стека, в который они должны быть помещены до вызова функции. Если функция в качестве результата возвращает некоторые данные, то они помещаются на вершину стека и могут быть использованы другой функцией.

В рамках реализаторской семантики обычно говорят, что функции извлекают из стека свои аргументы и помещают в стек вычисленные значения. С другой стороны, можно рассматривать любую функцию как преобразование стека, т. е. считать, что любая функция имеет ровно один аргумент — состояние стека, — и возвращает в качестве значения новое состояние стека.

В стековых языках программирования проще осуществляется возврат нескольких значений из функции, чем в аппликативных языках. В последних это осуществляется посредством объединения отдельных элементов данных в кортежи (например, списки для языка Lisp или структуры для языка C). В то время как в стековых языках это осуществляется последовательным помещением на вершину стека нескольких элементов

данных. Это исключает накладные расходы по предварительной упаковке данных и последующей распаковке перед их использованием.

Для записи функционального вызова не требуется никакого дополнительного синтаксиса, поскольку функциональный вызов осуществляется записью только имени функции. Такая программа выглядит как последовательность слов, в связи с чем вместо термина «функция» в стековых языках часто используется термин «слово» (например, в языке Forth). В настоящей работе будут использоваться оба термина, в зависимости от контекста.

Конкатенативные языки программирования используют нотацию стекового эффекта слова. Она описывает, какие данные слово извлекает из стека, и какие данные оно оставляет на вершине стека в качестве результата вычисления. Чаще всего слова имеют устойчивый стековый эффект — число входных и число выходных аргументов являются постоянными величинами.

3 Язык Joy

Стековый язык программирования Joy был разработан в 2001 году Манфредом фон Таном в университете Мельбурна, Австралия. Joy представляет собой чисто функциональный язык программирования: в программе на Joy отсутствует понятие состояния, и, как следствие, отсутствуют переменные и присваивание.

Типы данных, поддерживаемые языком, можно разделить на две группы — атомарные и агрегатные. Атомарные типы включат символы, целые числа, числа с плавающей точкой, логические величины. К агрегатным типам относятся строки, множества, списки, файлы и квотированные программы. Поддерживаются только числовые множества, содержащие целые значения в диапазоне от 0 до 31. Список является гетерогенной структурой данных, его элементами могуть быть данные

любых типов, как атомарных, так и агрегатных, в том числе и другие списки. Квотированная программа — это расширение понятие списка: наряду с данными, квотированная программа может также содержать функциональные вызовы. Элементы списков и квотированных программ записываются в квадратных скобках и разделяются пробелами, например:

```
[2 ["string" literal] 'c' {7 5 3} []]
[[1 2 3 4] [dup *] map]
```

Программа записывается в обратной польской нотации, в которой знак операции записывается после ее операндов. Например, программа, вычисляющая арифметическое выражение $(2 + 3) \cdot 15$, будет выглядеть следующим образом: `2 3 + 15 *`.

В языке формально вводятся операция квотирования и обратная ей операция деквотирования, которые применимы к исполняемому коду программы и к квотированным программам соответственно. Квотированием программы является простое помещение текста программы в квадратные скобки, посредством чего исполняемая программа превращается в объект данных, который можно передать фунции как параметр. Деквотирование совершает обратное действие — активизирует квотированный программный код, делая его готовым к немедленному исполнению. Таким образом, выявляется концепция единства данных и программного кода, которая впервые была реализована Джоном Маккарти в языке Lisp [11].

4 Примеры некоторых комбинаторов языка Joy

Функции, осуществляющие в каком–либо виде деквотирование называются комбинаторами. С помощью комбинаторов осуществляется управление потоком программы (ветвление, рекурсия), а также с помощью комбинаторов реализуются функции высших порядков.

Для начала рассмотрим каким образом в Joy записываются определения функций.

```
DEFINE
    square == dup * ;
    cube   ==  dup dup * *
END
```

Здесь представлены определения двух функций. Функция `square` вычисляет квадрат своего числового аргумента. Для этого она сначала удваивает аргумент, лежащий на вершине стека функцией `dup`, а затем функция `*` достает с вершины стека два верхних элемента и возвращает на стек их произведение. Аналогично, функция `cube` возводит свой аргумент в третью степень.

Блок определений функций начинается с ключевого слова [1] `DEFINE` и заканчивается словом `END`. Внутри блока может находится одно или несколько определений. Между названием и телом функции ставится разделитель `==`. В случае нескольких определений, они отделяются друг от друга точкой с запятой.

Язык Joy не предоставляет специального синтаксиса управляющих конструкций программы. Как уже было сказано, управление программой производится с помощью комбинаторов. Например, комбинатор `ifte` реализует ветвление. Функция ожидает на вершине стека 3 параметра, каждый из которых является квотированной программой. Рассмотрим пример:

```
DEFINE
    factorial == [0 =]
                 [pop 1] [dup 1 - factorial *] ifte
END
```

Здесь представлено рекурсивное определение функции, считающей факториал целого числа. Третий аргумент комбинатора

[1] Здесь под «словом» понимается лексическая единица, а не специфичное для стековых языков название функции. Ключевые слова не являются функциями, это всего лишь метки.

(`if-part`) проверяет аргумент на равенство нулю. Если условие выполнено, то деквотируется второй агрумент (`then-part`) и на вершину стека попадает ответ ($0! = 1$). Если условие не выполнено, то деквотируется первый аргумент (`else-part`), в котором происходит рекурсивный вызов для аргумента, уменьшенного на единицу ($n! = (n-1)! \cdot n$).

Для реализации рекурсии Joy предоставляет большое число рекурсивных комбинаторов. Рассмотрим два из них — комбинатор линейной рекурсии `linrec` и комбинатор бинарной рекурсии `binrec`. Комбинатор `linrec` ожидает на вершине стека 4 параметра, каждый из них — квотированная программа: `[if-part] [then-part] [rec1-part] [rec2-part] linrec`. Четвертый параметр проверяет условие выхода из рекурсии и, если оно выполняется, то выполняется нерекурсивная часть определения. Если условие не выполняется, то сначала выполняется `rec1-part`, затем происходит рекурсивный вызов для только что вычисленного значения, после возврата из которого выполняется `rec2-part`. В следующем примере дается определение факториала с помощью `linrec`:

```
DEFINE
    factorial1 == [null] [succ] [dup pred] [*] linrec
END
```

Полиморфная функция `null` возвращает `true`, если ее аргументом является нуль (целого типа или с плавающей точкой) или пустой объект агрегатного типа (строка, список или множество), а каждая из функций `pred` и `succ` уменьшает или, соответственно, увеличивает значение целого аргумента на единицу.

Комбинатор бинарной рекурсии совершает два рекурсивных вызова. Комбинатор `binrec` ожидает на вершине стека 4 аргумента-квотированные программы. Почти все они имеют тот же смысл, что и для комбинатора `linrec` — кроме `rec1-part`. Теперь результатом выполнения этой квотированной программы должны быть два значения, для каждого из которых происходит рекурсивный вызов.

Комбинаторы можно использовать не только в определениях пользовательских функций. Следующий фрагмент программного кода вычисляет восьмой элемент последовательности Фибоначчи: `8 [small] [] [pred dup pred] [+] binrec`. Полиморфная функция `small` возвращает `true`, если ее целый числовой аргумент не больше единицы или, если объект агрегатного типа содержит не более одного элемента.

Рассмотрим несколько комбинаторов, которые будут использоваться в примерах (схема вида `A1 ...An -> B1 ...Bn` обозначает стековый эффект функции — `Ai` обозначают аргументы функции, а `Bi` обозначают ее возвращаемые значения):

- `uncons: list -> head tail` — разделяет список на голову и хвост, первым кладет в стек голову, поверх нее — хвост;
- `split: qprog list -> listX listY` — разбивает список на два, которые возвращает на стек. В первый (`listX`) попадают элементы, для которых выполнение квотированной программы (`qprog`) дает ненулевой результат (или `true`), из всех остальных формируется второй список (`listY`);
- `cons: elem list -> list1` — помещает элемент в начало списка;
- `dip: qprog param -> ...` — первым параметром является квотированнная программа. Второй параметр убирается со стека и сохраняется, затем выполняется квотированная программа, после чего сохраненный параметр помещается на вершину стека.

Более интересным примером использования комбинатора бинарной рекурсии является описание функции быстрой сортировки:

```
DEFINE
```

```
    qsort == [small] []
             [[uncons] [>] split]
             [[swap] dip cons concat]
             binrec
END
```

5 Библиотека InteLib и ее структура

Проект InteLib [12] представляет собой библиотеку, реализующую метод непосредственной интеграции, где в качестве базового выбран объектно–ориентированный язык C++ [13]. Язык C++ обладает механизмом переопределения символов стандартных операций для пользовательских типов данных, что является ключевым моментом в выборе базового языка программирования для реализации метода непосредственной интеграции.

Рассмотрим понятие символьного выражения (или S-выражения). S-выражение представляет собой либо атомарные данные (символ, числовая константа, строковый литерал и т. п.) либо точечную пару вида `(A . B)`, где `A` и `B` также являются S-выражениями. Наряду с понятием точечной пары вводятся понятия списка и пустого списка. Список — это точечная пара вида `(A . (B . (C . NIL)))`, где `NIL` — специальное обозначение для пустого списка. Таким образом, на основе типизированных атомарных S-выражений можно строить нетипизированные контейнеры, являющиеся составными S-выражениями, и содержащими в себе разнородные данные.

Над S-выражениями определены операции композиции (составление точечной пары из двух заданных S-выражений), декомпозиции (получение каждого из элементов точечной пары) и вычисления S-выражения (некоторое отображение множества S-выражений на само себя).

Библиотека InteLib реализована в виде нескольких логических слоев. Нижний слой — это S-выражения, не связанные ни с какими вычислительными моделями. Символьны-

ми выражениями в библиотеке InteLib представляются данные и программный код. S-выражения реализованны в виде иерархичного дерева наследования с общим предком. Корнем иерархии является класс `SExpession`, представляющий наиболее общие свойства всех возможных S-выражений, от него наследуются классы, представляющие конкретные S-выражения, такие как `SExpressionInt`, `SExpressionString`, `SExpressionChar`, `SExpressionCons` и другие.

Для удобства использования класса `SExpession` был разработан класс `SReference`, являющийся интерфейсом для эффективной работы с S-выражениями. Этот класс инкапсулирует S-выражение, его поведение аналогично поведению указателя на S-выражение, кроме того, `SReference` может реализовывать и некий дополнительный функционал (например, счетчик ссылок на S-выражение, позволяющий организовать сборку мусора).

Верхний слой библиотеки реализует вычислительные модели S-выражений для различных языков, таких как Lisp, Scheme [14], Refal [15]. Кроме того, в библиотеке представлен средний слой, предоставляющий некоторое инструментальные средства для работы с S-выражениями (средства ввода-вывода, синтаксический анализ текстового представления и т. п.), а так же инкапсулирующий некоторые общие части вычислительных моделей для S-выражений. В ходе реализации библиотеки активно используются возможности определения абстрактных типов данных в языке C++, наследования и полиморфности объектов.

Идея реализации общей части для вычислительных моделей в библиотеке InteLib была навеяна механизмом продолжений, используемым в языке Scheme. Продолжение представляет собой последовательность действий, которые необходимо выполнить до завершения вычисления [3]. Результатом выполнения этих действий являются некоторые значения, которые могут использоваться для совершения последующих действий.

Поэтому речь идет о двух основных сущностях: о совершаемых действиях и результатах их значений. При этом чем позднее было получено значение выражения, тем быстрее оно понадобится для вычислений, что естественно приводит нас к концепции двух стеков — стека значений и стека действий.

Библиотека InteLib предоставляет все необходимые инструменты для библиотечной реализации языка Joy, а именно S-выражения для реализации гетерогенных списков и вычислительную модель на основе продолжений для реализации конкатенативности.

6 Реализация вычислительной модели

Вычислительная модель языка очень проста. После разбиения на лексемы текст программы представляет собой последовательность слов-функций и значений произвольного типа (атомарного или агрегатного), разделенных пробелами. Текст программы просматривается последовательно. Если текущий элемент является значением, то это значение просто кладется в стек, а если на очередном шаге встречается имя функции, то из стека извлекаются аргументы функции (в соответствии с ее стековым эффектом), далее происходит вычисление функции, после чего результат попадает на вершину стека.

Единственная конструкция, вычисление которой не соответствует вышеописанному алгоритму, — это объявление пользовательской функции. При вычислении блока определений, имя и тело определяемой функции не вычисляются, а заносятся в специальную таблицу функций, откуда они будут доступны для дальнейшего использования. Такая таблица реализуется по аналогии со словарем языка Forth.

Таким образом, вычисление программы не всегда происходит по одному сценарию. Вычислитель может находиться в двух состояниях: определение пользовательской функции и интерпретация программного кода. Реализуется такое поведе-

ние с помощью конечного автомата. Для удобства определению пользовательской функции соответствует не одно, а два состояния. В первом из них происходит считывание и запоминание имени функции, а во втором — последовательное считывание и запись элементов тела функции. Диаграмма состояний вычислителя представлена на рис. 1. Переход по дугам, помеченным звездочками, происходит по имени функции (стандартной или уже определенной пользовательской) или по значению. Если в некотором состоянии вычислителю встречается лексема, по которой нет переходов в другие состояния, то такая ситуация является исключительной.

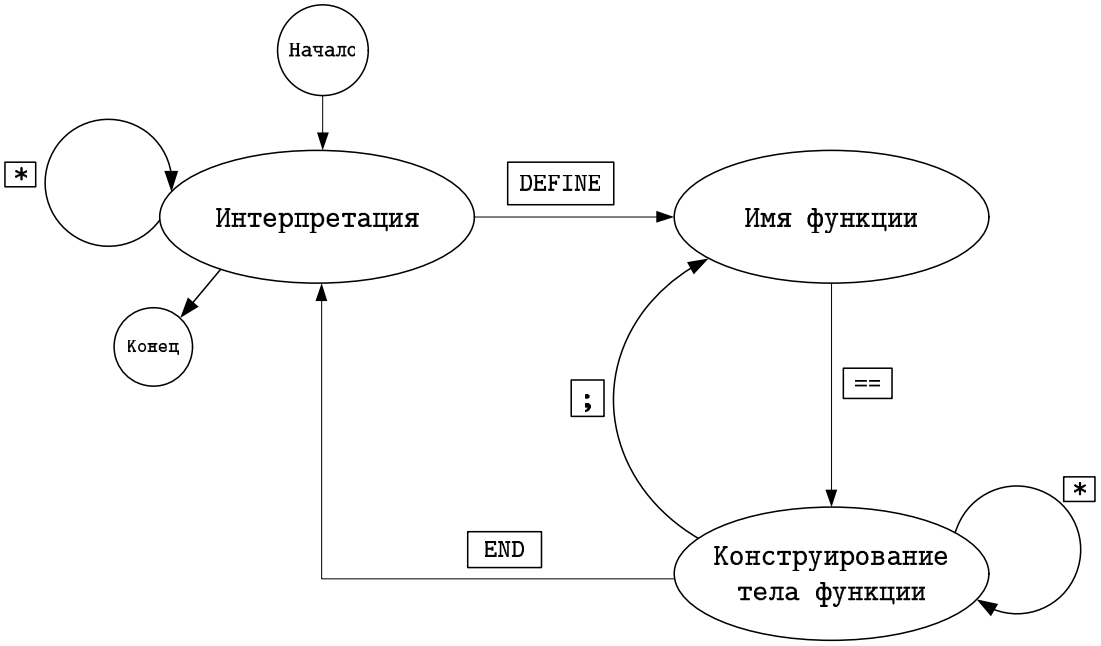

Рис. 1. Диаграмма состояний вычислителя

Для реализации вычислительной модели был использован класс `IntelibContinuation` библиотеки InteLib. Этот класс реализует коцепцию продолжений (континуаций) и инкапсулирует два стека — стек значений и стек действий — и предоставляет инструменты для работы с ними. На очередном шаге работы континуации из стека действий извлекается верхний элемент — инструкция, предписывающая континуации выполнение опре-

деленного функционала. Данные, получаемые в ходе этих действий, помещаются в стек результатов.

Класс `IntelibContinuation` предоставляет досточно объемный набор инструкций, однако в производном классе этот набор может быть расширен.

Вычислительная модель языка Joy реализуется классом `JoyContinuation`, являющимся наследником класса `IntelibContinuation`. В континуацию были добавлены две инструкции: `joy_evaluate` и `joy_dequote`. По инструкции `joy_evaluate`, примененной к очередной встретившейся лексеме, континуация производит вычисление этой лексемы в соответствии с предложенной выше вычислительной моделью. Инструкция `joy_dequote` активизирует квотированный программный код, такая иструкция может быть применена только к квотированной программе. Технически эта инструкция разбирает список на элементы и для каждого элемента списка применяет инструкцию `joy_evaluate`.

При обсуждении символьных выражений отмечалось, что список является точечной парой специального вида: `(A . (B . (C . NIL)))`, причем пустой список не является точечной парой, а представляет собой атомарное S-выражение. Таким образом, списки и квотированные программы языка Joy реализуются классом `JExpressionQProgram`, являющимся наследником класса `SExpressionCons`, а пустой список реализуется отдельным объектом типа `SExpressionLabel`. Указатель на квотированную программу реализуется специальным классом `JReferenceQProgram`, унаследованным от класса `SReference`. Этот класс реализует операцию деквотирования списка (другими словами, операцию его вычисления), которая представлена методом `Evaluate`. Метод `Evaluate` производит вычисление списка с помощью континуации, используя описанную выше инструкцию `joy_dequote`. Операция деквотирования списков используется при реализации библиотечных комбинаторов, а также при написании кода на языке Joy непосредственно в ко-

де C++.

Далее вводится класс `JoyQProgramConstructor`. В нем перегружается операция побитового или `|`, которая конструирует и возвращает указатель на список из одного элемента (то есть, объект класса `JReferenceQProgram`), этот элемент — единственный аргумент перегруженной внутри класса операции `|`. Для создания пустого списка перегружается операция побитового отрицания `~`. Для добавления нового элемента в уже существующий список, в классе `JReferenceQProgram` переопределяется операция запятая, позволяющая максимально приблизить запись списка целевого языка констукциями языка C++. Ниже представлен пример создания списка `[1 2 3 4]` и пустого списка `[]`:

```
JoyQProgramConstructor J;
(J| 1, 2, 3, 4);
~J;
```

Множества языка Joy могут содержать только целые числовые значения в диапазоне от 0 до 31. Множество реализуется классом `JExpressionSet`, наследником `SExpression`. Для конструирования множеств, так же, как и для списков вводятся классы `JoySetConstructor` и `JReferenceSet`. Ниже приводятся примеры создания множеств `{1 2 3 4}` и `{}`:

```
JoySetConstructor S;
(S| 1, 2, 3, 4);
~S;
```

Для реализации специальных лексем в библиотеке InteLib существует класс `SExpressionSpecialToken`, являющийся наследником класса `SExpressionLabel`. Этот класс имеет абстрактный метод `Evaluate`, который переопределяется в наследнике класса, задавая специальную логику вычисления реализуемой лексемы.

Класс `SExpressionSpecialToken` используется для реализации библиотечных функций языка Joy. От него наследуется класс `JFunctionExpression`, являющийся абстрактным представлением функции. В нем реализуется некоторый дополнительный функционал, однако метод `Evaluate` в нем остается

абстрактным, он переопределяется в наследниках класса, реализующих конкретные библиотечные функции.

Поскольку при программировании непосредственно S-выражения не используются, по аналогии со списками и множествами, для использования функций вводится класс `JReferenceFunction`, представляющий указатель на функцию. Конструктор класса принимает в качестве аргумента указатель на экземпляр класса функции. Например, следующий код определяет указатель на функцию `MUL`, реализуемую классом `JFunctionMul`:

```
JFunctionReference MUL(new JFunctionMul());
```

Для записи ключевых слов (`DEFINE`, `END`) и разделителей (`;`, `==`) используется класс `JToken`, конструктор которого принимает на вход текстовое представление слова (разделителя). Из-за требований к идентификаторам языка C++, а также для единообразия, вместо разделителей `;` и `==` используются соответственно ключевые слова `SEMICOLON` и `EQUALS`. Для записи имен функций при определении (и при последующем вызове) используется класс `JLabel`, конструктор которого так же принимает на вход строку с именем функции.

Вспомним уже встречавшийся пример определения функций `square` и `cube` на языке Joy и посмотрим как аналогичный код будет выглядеть на языке C++.

```
// Required declarations
    JoyQProgramConstructor J;

// Declarations of keywords
    JToken DEFINE("DEFINE");
    JToken EQUALS("EQUALS");
    JToken SEMICOLON("SEMICOLON");
    JToken END("END");

// Declarations of names of user's functions
    JLabel square("square");
    JLabel cube("cube");
```

```
// Declarations of library functions
    JFunctionReference DUP(new JFunctionDup());
    JFunctionReference MUL(new JFunctionMul());

// Code of interest
    (J|
        DEFINE,
            square, EQUALS, DUP, MUL, SEMICOLON,
            cube, EQUALS, DUP, DUP, MUL,
        END
    ).Evaluate();
```

В коде на языке C++ сначала объявляются все используемые имена, после чего следует код, аналогичный коду на языке Joy с той разницей, что он записывается в виде квотированной программы и деквотируется (иначе пришлось бы использовать оператор `Evaluate` для каждой лексемы, что сделало бы код запутанным). Код на языке C++, моделирующий код языка Joy, семантически эквивалентен и приближен синтаксически (насколько это возможно) к оригинальному коду Joy.

7 Пример реализации функции сопоставления строки с образцом

Ниже приводится пример полной программы на языке C++, определяющей в синтаксисе языка Joy функцию `match` сопоставления строки с образцом. Знак `*` в образце обозначает произвольную подстроку, а знак `?` — один произвольный символ. В примере используется комбинатор `cond`, семантика которого унаследована из языка Lisp. Функция ожидает на стеке 2 параметра: первый — шаблон, второй — сопоставляемая строка. После определения функции, средствами языка C запрашиваются у пользователя шаблон и строка, которую нужно сопоставить. И, наконец, определенная функция применяется к введенным параметрам. Функция возвращает 1, если строка

соответствует шаблону, и 0 в случае неуспеха. Чтобы не загромождать пример, ответ функции не анализируется, а просто выводится на экран (функцией `print` языка Joy).

```
#include <stdio.h>
#include "joycont.hpp"
#include "joynames.hpp"
#include "joyqprogram.hpp"
#include "jfunction.hpp"

int main() {
    JoyQProgramConstructor J;
    JoyContinuation stacks;

    JFunctionReference PRINT(new JFunctionPrint());
    JFunctionReference EQ(new JFunctionEq());
    JFunctionReference IFTE(new JFunctionIfTE());
    JFunctionReference FIRST(new JFunctionFirst());
    JFunctionReference REST(new JFunctionRest());
    JFunctionReference DIP(new JFunctionDip());
    JFunctionReference COND(new JFunctionCond());
    JFunctionReference DROP(new JFunctionDrop());

    JToken DEFINE("DEFINE");
    JToken EQUALS("EQUALS");
    JToken SEMICOLON("SEMICOLON");
    JToken END("END");

    JLabel match("match");
    JLabel starmatch("starmatch");
    JLabel drop2("drop2");

    (J|
      DEFINE,
        match, EQUALS,
        (J|
            (J| (J| FIRST, '\0', EQ),
                  DROP, FIRST, '\0', EQ),
            (J| (J| FIRST, '?', EQ),
                  (J| DROP, "", EQ),
```

```
                (J| drop2, false), (J| (J| REST),
                DIP, REST, match), IFTE),
            (J| (J| FIRST, '*', EQ),
                REST, starmatch),
            (J| (J| true),
                (J| (J| FIRST), DIP, FIRST, EQ),
                (J| (J| REST), DIP, REST, match),
                (J| drop2, 0), IFTE)
        ), COND,
        SEMICOLON,
        starmatch, EQUALS,
        (J|
            (J| (J| match), drop2, true),
            (J| (J| DROP, "", EQ), drop2, false),
            (J| (J| true), (J| REST), DIP, starmatch)
        ), COND,
        SEMICOLON,
        drop2, EQUALS, DROP, DROP,
      END
    ).Evaluate(stacks);

    char pattern[200];
    char string[200];

    printf("%s", "Please, enter the pattern: ");
    scanf("%s", pattern);

    printf("%s", "Please, enter the string: ");
    scanf("%s", string);

    (J| string, pattern, match, PRINT).Evaluate(stacks);

    return 0;
}
```

8 Заключение

Целью данной работы являлась реализация библиотечного расширения языка C++ для работы со стековым функциональным языком Joy. В ходе работы применялся метод непосредственной интеграции и использовалась библиотека InteLib. В результате работы была реализована вычислительная модель языка Joy и библиотека стандартных функций языка.

Несмотря на то, что код, моделируемый конструкциями C++, похож по синтаксису на оригинальный код Joy, он остается достаточно громоздким и требует написания дополнительных инициализирующих конструкций. Поэтому реализация была дополнена транслятором, переводящим модули в оригинальном синтаксисе Joy, в модули на языке C++, которые можно непосредственно включить в проект. Это позволяет не только избавить разработчика от необходимости написания кода Joy в синтаксисе C++, но и дает возможность без больших трудозатрат преобразовать уже существующий код на языке Joy для использования в проекте на C++.

Список литературы

[1] А. В. Столяров. *Об одном подходе к построению универсальных языков программирования.* Сборник статей молодых учёных факультета ВМиК МГУ, N 4. М.: Издательский отдел факультета ВМиК МГУ, 2007, стр. 135–146.

[2] И. Г. Головин, А. В. Столяров. *Объектно-ориентированный подход к мультипарадигмальному программированию.* Вестник Московского Университета, сер. 15 вычисл. матем. и киберн., 2002, № 1, стр. 46–50.

[3] А. В. Столяров. *Импорт вычислительной модели языка Scheme в объектно–ориенторованное окружение.* Сборник статей молодых учёных факультета ВМиК МГУ, № 5. М.:

Издательский отдел факультета ВМиК МГУ, 2008, стр. 119–130.

[4] И. Е. Бронштейн, А. В. Столяров. Библиотечная поддержка вычислительной модели языка Рефал. Сборник статей молодых учёных факультета ВМиК МГУ, № 6. М.: Издательский отдел факультета ВМиК МГУ, 2009, стр. 36–46.

[5] И. В. Струков. Библиотечная реализация Пролог–решателя. Сборник статей молодых ученых факультета ВМК МГУ, № 9, 2012. М.: Издательский отдел факультета ВМиК МГУ, 2012, стр. 235–250.

[6] Leo Brodie. *Thinking FORTH. A Language and Philosophy for Solving Problems.* Englewood Cliffs, N. J., Prentice–Hall, Inc., 1984.

[7] Manfred von Thun. *Joy: Forth's Functional Cousin.* Proceedings of the 17th EuroForth Conference, 9 October 2001.

[8] Glenn C. Reid. *Thinking in PostScript.* Addison–Wesley, 1990.

[9] Christopher Diggins. *Cat: A Functional Stack-Based Little Language.* Doctor Dobbs Journal, April 15th, 2008.

[10] Slava Pestov, Daniel Ehrenberg, Joe Groff. *Factor: a dynamic stack-based programming language.* Dynamic Languages Symposium 2010.

[11] Э. Хювёнен, И. Сеппянен. *Мир Лиспа. В 2-х т.* Мир, 1990.

[12] Официальный сайт проекта InteLib. http://www.intelib.org

[13] Бьерн Страуструп. *Дизайн и эволюция языка C++.* ДМК Пресс, Питер, 2006.

[14] R. Kesley, W. Clinger and J. Rees. *Revised^5 report on Algorithmic Language Scheme.* ACM SIGPLAN Notices, 1998.

[15] В. Ф. Турчин *Алгоритмический язык рекурсивных функций (РЕФАЛ).* — М.: ИПМ АН СССР, 1968.