УДК 004.434 **DOI: 10.18413/2518-1092-2017-2-2-21-33**

Седых А.
Рязанов Ю.Д.

СКРИПТОВЫЙ ЯЗЫК «LINK» ДЛЯ ПРОГРАММИРОВАНИЯ СЦЕНАРИЕВ ИСПОЛНЯЕМЫХ СОБЫТИЙНЫХ СИСТЕМ

Белгородский государственный технологический университет им. В.Г. Шухова,
ул. Костюкова, 46, г. Белгород 308012, Россия

e-mail: chronoexp@gmail.com, ryazanov.iurij@yandex.ru

Аннотация
Языки описания сценариев (скриптовые языки) – высокоуровневые языки программирования, которые интерпретируются некоторой программой во время её выполнения. Скриптовые языки предоставляют довольно привлекательные возможности, обладают более сложным инструментарием и поддерживают более прогрессивные техники программирования, чем, например, компилируемые языки. Они, как правило, позволяют простым и компактным способом описывать управление большим количеством объектов предметной области. Однако многие языки базируются на некоторых шаблонных подходах, которые могут создавать проблемы во время использования и породить множество ошибок. В статье представлен разработанный авторами специализированный язык Link для описания сценариев асинхронных событийных систем. В языке Link решены такие известные проблемы языков программирования, как использование значения null, тесная взаимосвязь данных с кодом, большая сложность разработки и чтения кода, глобальные области видимости переменных. Некоторые его возможности, такие как линки, разделение функций на типы, операции применения функций, классы функций, частичная настройка и наследование функций обеспечивают отличающийся от большинства языков подход в разработке сценариев исполняемых событийных систем.
Ключевые слова: скриптовые языки программирования; описание сценариев; асинхронные событийные системы; теория языков программирования.

UDK 004.434

Sedykh A.
Ryazanov Y.D.

SCRIPTING PROGRAMMING LANGUAGE “LINK” FOR RUNNABLE EVENT-DRIVEN SYSTEMS

Belgorod State Technological University named after V.G. Shoukhov, 46 Kostyukova St., Belgorod, 308012, Russia

e-mail: chronoexp@gmail.com, ryazanov.iurij@yandex.ru

Abstract
Script description languages (scripting languages) are high-level programming languages that are interpreted by some program during its execution. Script languages provide quite attractive features, has more complex tools and supports more advanced programming techniques, in comparison to compiled languages. Scripting languages allow a simple and compact way to describe the control flow of a large number of objects in the subject area. However, many languages are based on some template approaches that can create problems during use and cause a lot of errors. The article presents a specialized language developed by the authors for describing scenarios of asynchronous event-driven systems. The Link language solved such well-known problems of programming languages as the use of null, close correlation of data with the code, great difficulty in developing and reading code, global scope of variables. Some of its features, such as links, the separation of functions into types, operation of application functions, function classes, partial tuning and inheritance of functions offer an approach different from most languages in the development of scenarios of runnable event-driven systems.
Keywords: scripting programming languages; script composition; asynchronous event-driven systems; theory of programming languages.

Введение

На текущее время существует множество языков программирования, созданных под самые различные нужды. Помимо универсальных языков программирования, таких как C, Java и C# также разрабатываются специфичные определенной среде выполнения сценарные языки, такие как sh, Perl, Elixir и Erlang [1]. Необходимость в создании нового языка или улучшении существующего обуславливается как появлением новых окружающих условий или систем для программирования, так и возникновением идей для усовершенствования существующей концепции написания программ на определенном языке. К примеру, недавно был разработан новый проблемно-ориентированный язык для удаленного программирования контроллеров, в котором была применена идея удаленного программирования с помощью модуля GSM [2].

Языки описания сценариев (скриптовые языки) – высокоуровневые языки программирования, которые интерпретируются некоторой программой во время её выполнения [3]. Это отличает их от компилируемых языков, таких как C, C++ или Go. В общем плане, языки подобного рода предназначены для более удобного управления объектами в сценариях какой-то уже существующей системы [4]. Создание новых и усовершенствование существующих скриптовых языков также является актуальной темой для проведения научной работы [5].

Языки описания сценариев предоставляют довольно привлекательные возможности, обладают более сложным инструментарием и поддерживают более прогрессивные техники программирования. Они, как правило, помогают простым и компактным способом обрабатывать и управлять большим количеством объектов в применяемой предметной области. Однако многие языки базируются на некоторых шаблонных подходах, которые могут создавать проблемы во время использования и породить множество ошибок. Но существуют языки программирования, в которых удалось избавиться от некоторых недостатков такого рода.

Целью данной работы является разработка нового скриптового языка для управления множеством объектов асинхронных событийных систем различной структуры и поведения, позволяющего упростить программирование, чтение и отладку сценариев, а также минимизировать возникающие в процессе программирования проблемы.

В первой части статьи приведены некоторые существующие актуальные проблемы современных языков программирования. Описываются различные способы решения каждой из проблем в различных языках программирования на данный момент. Во второй части статьи приводится неформальное описание принципа работы и особенностей разработанного авторами скриптового языка программирования, который включает в себя композицию хорошо зарекомендовавших себя и совершенно новых методов решения описанных проблем.

1. НЕКОТОРЫЕ ПРОБЛЕМЫ СОВРЕМЕННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Среди наиболее известных проблем популярных языков программирования присутствуют такие как наличие неопределенности пустоты переменной (использование значения null), тесная взаимосвязь данных с кодом в императивных языках программирования, большая сложность разработки и чтения кода. В сценарной среде исполнения поднимается вопрос расположения точки входа и привязки необходимого кода к определенной ситуации в системе [6]. Базовым решением, например, как в языке JavaScript, является предоставление объекту или прототипу требуемой функции обратного вызова. Однако реальные системы нередко требуют более детального описания ситуации, при которой нужно запускать код. Ещё одной серьёзной проблемой является отведение специального значения для обозначения отсутствия чего-либо – так называемое null-значение. Возможность каждой переменной иметь ссылку не только на реально существующий объект, но и на отсутствие этого объекта значительно усложняет процесс разработки и отлаживания кода. Рассмотрим эти и другие проблемы более подробно.

1.1 Глобальная видимость объектов, замыкания и callback hell

В скриптовых языках зачастую в функциях обратного вызова требуется доступ к внешнему окружению объекта, что реализуется системой замыканий. В результате может возникнуть избыточная загруженность большим количеством переменных и замыканий, разобраться в которой затруднительно. В проектах с небольшим количеством разработчиков это может не являться сильной проблемой, но в крупных корпоративных разработках такая сложность разработки превращается в ад. Так и зародился термин «callback hell» – «ад коллбэков», который часто применяют, критикуя JavaScript [7].

Чем меньше будет область видимости функции, тем проще её реализовывать и поддерживать. Нет необходимости нагружать текущий контекст всем вышестоящим содержимым, как это сделано в случае замыканий в JavaScript. Примером идеального в этом плане языка можно привести Haskell – в этом языке отсутствуют глобальные области видимости переменных. Функция зависит только от переданных ей параметров, и вся разработка сводится к описанию зависимостей выходных данных от входных. Такие функции называются чистыми, их проще разрабатывать, отлаживать и поддерживать [8]. Как правило, код написанный на Haskell, работает правильно с первой же компиляции. Во многом это заслуга именно отсутствия глобальных переменных и различных замыканий. Такой код хорошо структурируется и его легче «разложить по полочкам». В некоторых языках программирования разработчики пытаются хоть и не избавиться, но наиболее безопасно использовать замыкания. Например, в php для передачи определенных переменных в контекст вложенной функции используется служебное слово use. Таким образом, внутри функции доступны только переменные, которые туда нужно передать. Тем самым фильтруется набор доступной изнутри функции информации. Такой подход уменьшает сложность написания и поддержки программ с использованием замыканий по сравнению с JavaScript [9].

1.2 Ошибка несоответствия типов

В динамических языках программирования вроде Python ошибка несоответствия типов переменных во время выполнения – довольно частое явление. Возможность не указывать тип данных в совокупности с тем, что тип данных конкретной переменной может меняться со временем стала одной из основных проблем динамических языков. Вместе с этой возможностью писать код стало легче, однако чтение и отладка того же кода представляет собой довольно сложный процесс [10]. Более свободная и менее ограничивающая динамическая типизация накладывает на разработчика более строгие требования по документированию кода. Чтобы читатели кода не догадывались, как могут повести себя функции, что они принимают и что возвращают, разработчик должен всегда следить за актуальностью информации, которая в статических языках частично прошита на уровне системы типов и автоматически обновляется при рефакторинге [11]. Использование же строго типизированного языка сильно увеличивает объем кода и порой ограничивает разработчика в гибкости его кода.

В качестве компромисса возникает идея создания скриптового языка с универсальной динамической типизацией, в котором все значения будут обрабатываться одним и тем же образом, без возможности изменить значение идентификатора. Исходя из этого, можно одновременно использовать удобство автоматического вывода типов и не испытывать проблем с возникновением ошибок разной типизации, ведь большинство значений будет являться одним типом и при необходимости будет преобразовываться в нужный применяемый тип.

1.3 Ошибка пустого указателя (NullPointerException)

В марте 2009-го года сэр Тони Хоар выступил на конференции Qcon в Лондоне с докладом на тему "Нулевые ссылки: ошибка на миллиард долларов" (Null References: The Billion Dollar Mistake), в котором признался, что считает изобретение нулевых указателей одной из главных своих ошибок, стоившей индустрии миллиарды долларов [12]. Попытка доступа к элементу объекта, который является нулевым указателем (null), вызывает ошибку времени выполнения. В языке Java это называется NullPointerException (NPE). В языке JavaScript присутствуют целых два пустых состояния – null и undefined [13]. Наличие таких специальных значений в языках программирования очень сильно увеличивает число потенциальных ошибок. В результате программистам требуется проверять переменные на наличие требуемых объектов каждый раз перед совершением какого-либо действия. На данный момент существует масса способов избежать такой ошибки [14]. Некоторые способы, вроде повсеместного указания условных операторов слишком нагружают код, прямолинейны и требуют дополнительных вычислительных действий. Другие, как специальные атрибуты NotNull, Nullable, Optional, метод orNull и операторы coalesce, Элвиса и тернарные – описывают соглашения по спецификации конкретных переменных, а также их использование [15]. Среди подобных решений основная проблема – унаследованный код и сторонние библиотеки, которые по-прежнему имеют дело с нулевыми указателями. Также, ввиду своей необязательности, они позволяют разработчику создавать как безопасный, так и небезопасный в плане ошибки NPE код. Такая двойственность заставляет разработчиков по-прежнему заботиться по поводу ошибки нулевого указателя, хоть и в более мягкой форме. В языке Haskell для обработки результата функции, которая может возвратить пустое значение используется специальная монада Maybe [16]. Остальные же функции *обязаны* вернуть

значимый результат. Также одним из вариантов решения этой проблемы является разработка нового языка, без установленного изъяна в процессе проектирования. К примеру, в языке описания шаблонов Freemarker было решено не включать null в поддерживаемые значения [17]. Однако, в процессе взаимодействия с другими языками и системами, работающими на других языках – по-прежнему остается вероятность получения подобной ошибки. Если в генератор файлов языка Freemarker передать значение null, то возникнет ошибка во время использования шаблона. Требуется спроектировать такой язык, который не будет давать возможности возникновения таких ошибок. Само по себе отсутствие чего-либо не должно вызывать никаких исключительных ситуаций. В большинстве случаев, если в программе встречается значение отсутствия объекта, дальнейший код просто не должен выполняться. Таким образом, даже в результате взаимодействия с системами на других языках, ошибка NPE будет отсутствовать, потому что интерпретатор самостоятельно позаботится об остановке выполнения инструкции в нужный момент.

2. ПРИНЦИП РАБОТЫ И ОСНОВНЫЕ ОСОБЕННОСТИ СКРИПТОВОГО ЯЗЫКА LINK

Разработанный авторами язык **Link** – интерпретируемый внедряемый скриптовый язык программирования. Он исполняется в среде виртуальной машины JVM во время выполнения программы, написанной на языке Java [18]. Внутри языка полностью отсутствует хранимое состояние переменных – все данные считываются и записываются в интерпретирующую его программу, которая использует систему идентификации объектов языка Link. Программы, использующие Link должны выполняться с поддержкой Link-машины, которая предоставляет каркас для создания и управления всеми объектами системы. Link-машина содержит в себе подсистемы управления событиями и объектами, а также – подсистему интерпретатора, позволяющую исполнять код на этом языке.

В рамках языка введено понятие «типа» функции. Функции могут быть различных типов, где каждый тип означает то, как эта функция будет обрабатываться в системе. Программа на языке Link представляет собой набор функций различных типов. Точку входа определяют функции типа «линк» – они связываются с определенными условиями и событиями, произошедшими в событийной и объектной подсистемах. Таким образом, в Link-машине присутствует собственная подсистема мониторинга, которая контролирует все события и запускает по мере необходимости нужные линки.

Основной упор при разработке языка сделан на удобство и скорость программирования для создания высокоуровневых функций обработки и управления большими наборами объектов различных сущностей системы. Скрипт общается с внешним миром только с помощью своей машины и вызова «нативных» функций на пользовательской платформе. Нативными функциями называются такие функции, которые были реализованы в основной системе [19]. Таким образом, Link-машина встраивается в исполняемое на JVM приложение и вызывает любые методы классов, которые были зарегистрированы в нём.

Подсистема объектов представляет собой гибкий контейнер для хранения и поиска по всем зарегистрированным объектам. Объект в подсистеме обладает своим уникальным идентификатором, типом, а также атрибутами-классами. В любое время можно получить все необходимые объекты с помощью строки-селектора объектов на языке Link. Результатом действия селектора является список подошедших под условие объектов, с которыми можно производить любые доступные действия. Принцип работы селектора напоминает функцию $ в библиотеке jQuery для JavaScript [20].

Далее опишем основные особенности разработанного языка. Описание будет сопровождаться примерами кода на языке Link для решения задач из следующей предметной области. Разрабатывается моделируемая на 3D-движке демонстрационная интерактивная сцена офиса [21]. Каждый тип предмета или персонажа сцены в исполняемом приложении описан на языке Java как класс ООП и зарегистрирован в подсистеме объектов Link-машины. Для данной работы в офисе представлен следующий список классов: Guard – охранник, Document – документ, Computer – компьютер, Mouse – мышь, Chair – кресло. Также присутствует общий родитель для нескольких классов – LivingCreature – живое существо. Каждый объект этих классов имеет характерные для объекта свойства и методы поведения. Поставлена задача реализовать различные сценарии поведения некоторых типов объектов в зависимости от происходящих в офисе действий. Перейдем к описанию выделяющихся возможностей языка Link.

2.1 Линки

Линк – специальный тип функции, представляющий собой точку входа для сценария. Линк позволяет «линковать» – привязывать сценарий к определенным объектам, событиям, времени ожидания и

дополнительным произвольным условиям. Линк состоит из секции описания ситуации вызова сценария, и, собственно, самого сценария. Пример:

```
// после того как в офисе был прочитан документ, вывести в
// консоль имя этого документа и количество страниц
link documentProcessed
| object: [/Document/]^
| event: FinishedReading
| code: {
    print(“Просмотрен документ “ + object:name + “. Обработано страниц:  “ +
object:lines)
}
```

В представленном коде описана функция вывода в консоль имени документа и числа страниц документа, находящегося в офисе. Этот сценарий запускается всякий раз при совершении события `FinishedReading` объекта типа `Document`.

Приведем другой пример. Допустим, требуется написать код, который будет реализовывать следующий сценарий: если человек с классом охранника заходит в офис и наблюдает отсутствие всех компьютеров, то он через 5 секунд запускает сигнал тревоги. Для начала, попробуем написать этот сценарий на нативном для Link языке проекта – Java. Для реализации сценария добавим в класс охранника Guard метод `guardComputersStolen:`

```
class Guard extends NPC {
  // методы и свойства охранника
  // включая callOverallAlarm – запуск тревоги
  …

  // необходимо вызывать в конструкторе, т.к. идет привязка
  // к событийной подсистеме
  void guardComputersStolen() {
    final Guard guard = this;
    EventManager.link(this, "EnteredOffice",  new Callable {
      void exec(EventParams params) {
        try {
          List<Computer> computers =ObjectMgr.findByClass("Computer");
          if (computers.isEmpty()) {
            Thread.Sleep(5000);
            guard.callOverallAlarm();
          }
        } catch (InterruptedException ie) { /**/ }
      }
    });
  }
}
```

При разработке сценариев на том же языке, на котором написано основное ПО, требуется прописывать все необходимые для работы сценария инструкции. В данном примере большую часть сценария занимает именно такой код: связывание функции обратного вызова с событием; присвоение статичной ссылки на требуемый объект охранника для использования замыкания; обработка исключения, которое может возникнуть при отсчете времени [22]; громоздкое обращение к объектной подсистеме для выбора списка компьютеров; проверка данного списка на пустоту – всё это значительно усложняет написание, чтение и отладку исходного кода. На первый взгляд простая задача описания сценария реакции объекта на событие превращается в достаточно сложную реализацию. Если в разрабатываемом проекте запланировано создать

множество таких или, как правило, более сложных сценариев – исходный код проекта будет разрастаться высоконагруженными цепочками кода, обрабатывающего сценарии с различным поведением объектов. Однако, бо̀льшая часть этого кода будет заниматься лишь обслуживанием и связыванием самих сценариев, проверкой условий выполнения сценариев, в то время как сама логика действия будет занимать гораздо меньший объём.

Далее приводится реализация того же сценария с использованием языка Link:

```
link guardComputersStolen
| object: [/Guard/]^
| event: EnteredOffice
| condition: ![/Computer/]
| time: 5
| code: {
    object~Guard.callOverallAlarm$
}
```

Преимущество в краткости и лаконичности кода по сравнению с использованием нативного решения очевидно. Проведем детальный обзор происходящего в скриптовом коде. На второй строчке с пометкой `object` указывается объект, который выступает в роли инициатора события `event`. В данном случае это объект типа `Guard`. После того, как охранник заходит в офис, вызывается событие с именем `EnteredOffice` и проверяется условие, помеченное как `condition` – в нашем случае это обычный селектор всех объектов с типом `Computer` (подробнее механизм селектора описывается в пункте 2.2 этой статьи). Если данный селектор возвратит пустой список, отрицание пустого списка будет являться истинным выражением и сценарий запустится. Пометка `time` указывает количество секунд, которое требуется подождать перед началом выполнения сценария. Внутри секции `code` доступны все объекты линка, включая `object`, который будет обозначать инициатора данного события. Символ `~` обозначает операцию применения функции к левому операнду – охраннику, которая также будет описана позднее. Справа от этого символа указывается нативная функция `Guard.callOverallAlarm`, которая принадлежит исполняемой программе сцены офиса. Таким образом, разработчик скрипта знает, что в сцене у класса `Guard` реализован метод с названием `callOverallAlarm`, который позволяет охраннику поднимать тревогу.

2.2 Селектор объектов

Селектор объектов предоставляет специальный синтаксис для выборки объектов из подсистемы управления объектов. Оператор выборки является строкой, которая определяет условия для выборки. Условия могут быть самыми разнообразными: от классов, идентификаторов и типов объектов до выборки и фильтрации дочерних/родительских элементов. Схема использования селектора похожа на CSS-выборку в WEB – в строке указывается набор атрибутов объекта, по которым система выполняет поиск среди объектов исполняемой среды [23]. В селекторе языка Link можно использовать следующие атрибуты для выборки объектов:

1) Идентификатор. Идентификатор является уникальным значением для каждого объекта системы. В результате выборки по идентификатору селектор возвращает объект с заданным идентификатором, если он зарегистрирован в системе. Пример: `[/#Guard_32/]`.

2) Тип объекта. Типом объекта в Link-машине называется его класс в исполняемой среде JVM. При помещении контейнера объекта в подсистему объектов машины название класса регистрируется как атрибут типа для поиска и использования его в качестве селектора. Для запроса по типу объекта нужно просто указать в селекторе его название, к примеру - `[/Computer/].`

3) Класс объекта. Каждый объект в подсистеме объектов Link-машины может иметь набор классов, его описывающих. Выборка по классам представляет собой фильтрацию всех доступных объектов на соответствие заданным классам, подобно CSS-селектору. К примеру, описываемый офис может содержать белые и черные стулья с мягкой и жесткой спинкой. Допустим, что эти свойства реализованы в подсистеме объектов как классы `black`, `white`, `soft` и `hard` у объектов типа `Chair`. Чтобы произвести выборку в селекторе по классу, необходимо поставить символ “.” перед названием класса. В нашем случае, для

выборки всех черных стульев с мягкой спинкой мы можем использовать селектор `[/Chair .black .soft/]`.

4) Методы манипуляции с объектами иерархической структуры. При регистрации объекта в объектной подсистеме можно указать его родительский объект, который будет выступать в качестве контейнера новому объекту. Чтобы выбрать дочерние элементы контейнеров, нужно записать в селектор специальный символ `‘>’`, который возьмёт из текущих отфильтрованных объектов все дочерние объекты, и в дальнейшем фильтрация будет производиться именно по ним. Если контейнер не содержит дочерних объектов, он автоматически отклоняется. Для выборки родительских объектов существует специальный символ `‘<’`. К примеру, в интерактивной сцене существуют компьютеры типа `Computer` с классом `IBM`, внутри каждого из которых могут быть зарегистрированы платы оперативной памяти `RAM` класса `DIMM3`. Для выборки всей доступной памяти этого класса среди компьютеров IBM можно использовать следующий селектор: `[/Computer .IBM > RAM .DIMM3/]`.

Подобный синтаксис очень удобен для выборки целевых объектов сценариев. Система использования селекторов существенно облегчает написание и использование сценариев.

2.3 Вызов нативных методов

Связь языка Link с исполняемой на JVM программой реализуется с помощью объектной подсистемы. В этой подсистеме объекты Java содержатся внутри специальных контейнеров, называемых нативными. Нативный контейнер – это специальный объект, который содержит ссылку на настоящий объект Java, базовую информацию о хранимом объекте – его тип, персональный идентификатор в подсистеме, а также набор атрибутов, называемых классами. По этим параметрам и реализуется выборка значений с помощью селектора в объектной подсистеме.

В исходном коде Link разрешено вызывать нативные JVM методы, описанные в классах исходного проекта на языке Java. Для вызова нативного метода в сценарии достаточно сделать выборку контейнера требуемого объекта, и указать знак `$` в конце названия функции в случае применения метода над контейнером. Например, для выполнения нативного метода `callOverallAlarm` объекта с классом Guard достаточно написать `[/Guard/]~callOverallAlarm$`. Статические методы без привязки к конкретным объектам вызываются с помощью указания класса через символ точки: `System.println$("Привет!”)`. Применение нативных функций ничем не отличаются от применения функций самого языка Link. Благодаря рефлексии JVM, разработчику сценариев доступен весь набор классов и методов, который присутствует в исполняемом приложении [24]. Таким образом достигается прозрачность использования нативного кода приложения.

2.4 Расширенные операции, типы и их приведение

Автоматическая конвертация типов в случае использования литералов в Link позволяет в большинстве случаев не задумываться об используемых динамических типах. Все типы выводятся самостоятельно за счёт четкой логики выведения из исходного или нативного кода. Всего в языке существует 10 типов литералов:

Таблица 1

Литералы языка программирования Link

Table 1

Literals of Link programming language

№	Название	Пример	Описание
1	Integer	`2, 4, 5, 42`	Целое число
2	Float	`3.14, -5.24, 0.001`	Число с плавающей точкой
3	String	`"Hello", ""`	Строка из символов
4	Boolean	`false, true`	Логический тип
5	Identifier	`a, parseValue`	Идентификатор переменной
6	List	`[2, 3, "123"], []`	Набор значений, список
7	Map	`{literal: true, name: “map”}, {}`	Словарь

8	Selector	`[/Chair .black .soft/], [//]`	Селектор объектной подсистемы
9	Function	`Guard.alarm$, random`	Функция
10	Container	`[/Chair .black .soft/]^`	Контейнер объектной подсистемы

Как можно заметить, функция в Link также является литералом. Функции здесь не исполняются, а применяются к левому операнду с помощью операций применения функций (см. пункт 2.8). Специальный тип Selector отличается от списка лишь формой записи и поведением – когда потребуется вычислить значение этого литерала, выполнится запрос к объектной подсистеме и результат преобразится в список контейнеров. Запись конкретного контейнера в коде невозможна, контейнер объекта можно получить только с помощью селектора или вызова нативного метода.

Помимо привычных операций над обычными литералами, таких как вычитание и умножение чисел, в язык встроены дополнительные полезные операции над списками и объектами. Также расширен доступный набор действий при операциях со строками. К примеру, арифметические операции над списками и словарями реализуют базовые алгоритмы дискретной математики над упорядоченными множествами – + => объединение, - => разница, * => пересечение и / => исключающая разница [25]. Для списков добавлены операции взятия головы и хвоста, для словарей – выборка значения [26]. Всё это существенно облегчает работу по выборке, отсеиванию, выделению нужных параметров и изменению объектов сценария, ведь результатом работы селектора является список контейнеров.

Среди типов и операций отсутствует значение null. Необходимость в null исключается за счёт того, что результатом запросов селектора или вызова нативной функции всегда будет являться список. С точки зрения программиста видимо будет лишь одно значение – список объектов, который может быть пустой. Пустой же список объектов будет значить, что требуемые объекты не были найдены. В этом случае код над пустым списком не будет выполняться. Это позволит максимально упростить написание и отладку кода, достигнув логической простоты сценарных функций. Благодаря этому нововведению, при написании сценария можно будет не указывать повсеместно проверки на null или на пустоту списка там, где это на самом деле не требуется.

2.5 Встроенные типы функций

Все функции в языке Link классифицированы по типу функции. Для функций определенных типов применяется специальный синтаксис. На данный момент существуют такие типы как map, link, filter, reduce, transform. Некоторые типы функций созданы как безопасная замена циклов в императивных языках программирования. Идеология этих типов взята из основных функций высшего порядка в функциональных языках программирования, где они с успехом применяются [27]. Предопределенные ссылки идентификаторов вроде x, y, z – уже встроены в язык в пределах функции. Эти ссылки позволяют оперировать с входными и выходными значениями. Применение функции к объекту будет значить проход по одному элементу, если это один элемент, проход по списку – если это список, и отсутствие действий в случае пустого списка. В следующей таблице отражены все существующие типы функций.

Таблица 2

Типы функций языка программирования Link

Table 2

Function types of Link programming language

Название	Описание	Синтаксис
map	Отображает один набор данных в другой. **x** – элемент входного списка **y** – элемент выходного списка	map <name> | params: <arguments> | code: { <code>}
link	Связывает ситуацию в системе с исполняемым кодом сценария. **object** – связуемый объект	link <name> | object: <selector> | event: <name>

	event – связуемое событие **time** – связуемое время **condition** – связуемое условие	| time: <time> | condition: <expression> | code: { <code> }
filter	Фильтрует входной список. **x** – элемент входного списка **y** – логическое значение	filter <name> | params: <arguments> | code: { <code> }
reduce	Склеивает элементы списка в 1 элемент. **x** – входной элемент **y** – аккумулятор склеивания **z** – выходной элемент **initial** – начальное значение аккумулятора	reduce <name> | params: <arguments> | initial: <expression> | code: { <code> }
function	Стандартная функция в обычном понимании. **xs** – входной элемент **ys** – выходной элемент	function <name> | params: <arguments> | code: { <code> }
transform	Последовательное применение функции к контексту, пока не будет достигнуто условие. **x** – входной элемент **until** – условие завершения **code** – код преобразования	transform <name> | params: <arguments> | until: <expression> | code: { <code> }

2.6 Параметры функций и их наследование

Любая функция в Link состоит из параметров. Обязательными параметрами являются имя и тип функции. Обычно функции содержат и другие параметры: код и список аргументов. В случае нестандартных типов функций – например, линков – условия привязки. Идея разбиения функции на параметры заключается в том, что эти параметры можно будет указывать частично – например, указать для функции только аргументы, а реализацию описать в другом месте. В совокупности с возможностью наследования функций это даёт гибкую возможность описать шаблон функции, а позже лишь менять базовую часть. Далее описывается новая уникальная возможность языка – настройка функций.

Настройка функции – возможность описать все необходимые для работы функции параметры. По своему принципу работы она напоминает каррирование в языке Haskell, – функция агрегируется со своими аргументами по очереди, и лишь когда все параметры собраны функция может быть вычислена [28]. Однако, в Link функции собирают свои параметры, а не аргументы. Поэтому агрегировать их можно совершенно в любом порядке.

Наследование функций – возможность представленного языка повторно использовать один и тот же код. В совокупности с частичной настройкой, предоставляет разработчику гибкую возможность расширять уже существующий код. Подобно ООП, в наследовании функции участвуют два объекта: наследник и родитель [29]. Для наследования существует ограничение по типу функции – функция-наследник должна быть такого же типа, как и функция-родитель. В отличие от наследования классов, наследовать функции безопаснее, ведь отсутствует изменяемое состояние в рамках наследования.

Сущность наследования функции в том, что функция-наследник перенимает у функции-родителя все её параметры. Также, в теле наследника можно использовать специальную функцию parent, которая означает вызов функции родителя с указанными аргументами. Наследование функции, как и наследование классов в Java, записывается с помощью ключевого слова extends, после чего следует название функции-родителя.

2.7 Операции применения функций

В объектно-ориентированных языках программирования встречается следующий подход вызова методов у объектов:

Объект.метод1(аргументы);
Объект.метод2(аргументы);

В языке Link было решено пойти другим путём. Выделяются две операции применения функций: операция синхронного применения и асинхронного. Операция применения функций работает следующим образом: она помечает левый операнд как контекст, связывает его с функцией – правом операндом, и вызывает эту функцию выбранным способом с поддержкой связанного контекста. Контекст доступен внутри функции обычно по ссылке **x** или **xs**. После выполнения функции, контекст является измененным, и он меняется каждый раз, когда далее по коду происходит ещё применение функций над этим контекстом. Таким образом получаются цепочки применения функций на контексте.

Объект~функция1(аргументы)~функция2(аргументы);

Стоит выделить, что те же самые методы можно применить асинхронно, если они не связаны друг с другом:

Объект|функция1(агрументы)|функция2(аргументы);

Таким образом, применение функций становится частью выражения. Подобный синтаксис позволяет максимально прозрачно для разработчика использовать как синхронные, так и асинхронные вызовы [30]. Во время разработки сценария очень важно, чтобы сценарист не задумывался об организационных вопросах вроде распараллеливания подзадач, общих переменных и синхронизации. Это всё должно происходить автоматически, оставив для управления исполняемым потоком выбор из двух вариантов применения конкретной функции.

2.8 Классы функций

Классы функций являются уникальной возможностью привязывать дополнительный шаблонный код к функциям. Классом функции называется определенный именованный блок кода, который расположен вне всякой функции. Этим классом можно пометить любую функцию, тогда каждый раз при вызове функции будет выполняться указанный в блоке класса код. На данный момент существует три типа классов, которые отличаются временем выполнения: класс before запускается до выполнения основного кода привязанной функции, класс after – после выполнения. Третий тип класса wrap предназначен для оборачивания подписываемых функций в дополнительный код. Для класса wrap существует ключевое слово execute, которое производит выполнение основного кода функции. Приведем простой пример с использованием wrap:

```
// класс логирования в консоль любых исполняемых линков
// вместе с данными объекта и события
class wrap loggingLink {
  consoleLog(“Произошел вызов линка ” + this:name + “!”)
  consoleLog(“Данные объекта: “ + object)
  consoleLog(“Данные события: “ + event)
  execute
  consoleLog(“Закончился линк “ + this:name)
}

// опишем несколько линков

// каждое существо при входе в комнату начинает искать своих друзей
link creatureFindFriendsOnEntering using loggingLink
| object: [/LivingCreature/]^
| event: EnteredRoom
| code: {
  object~findFriends$
}

// при выходе одного из существ из комнаты все остальные существа
производят прыжок
link allCreaturesJumpOnLeaving using loggingLink
| object [/LivingCreature/]^
| event LeavingRoom
| code {
```

```
  // все кроме самого объекта совершают прыжок
  ([/LivingCreature/] - [object])~jump$
}

// и так далее…
```

Данный функционал напоминает аспектно-ориентированное программирование, где подобно классам функции можно описать общий логический слой для разных мест в исходном коде [31]. Например, при разработке финансовых операции мы можем определить класс транзакционных функций, который будет оборачивать функции с этим классом в транзакции. Также в Link разрешено комбинировать несколько классов для функции. Таким образом можно организовать многоуровневую логику для определенного вида функций.

ЗАКЛЮЧЕНИЕ

Разработанный скриптовый язык программирования Link предназначен для описания сценариев в системах с большим количеством разнотипных объектов, пользовательских и сторонних событий, требующих гибкое и удобное решение внедрения сценариев. Он подходит любым приложениям на любой платформе, так как Link является универсальным внедряемым языком программирования, и выполняется в мультиплатформенной исполняемой среде Java Virtual Machine [32]. В языке Link решены такие известные проблемы языков программирования, как использование значения null, тесная взаимосвязь данных с кодом, большая сложность разработки и чтения кода, глобальные области видимости переменных. Некоторые его возможности, такие как линки разделение функций на типы, операции применения функций, классы функций, частичная настройка и наследование функций обеспечивают отличающийся от большинства языков подход в разработке сценариев исполняемых событийных систем.

Дальнейшим расширением языка может быть введение модульности и системы экспорта и импорта функций между модулями, введение возможности выделить действующее лицо сценария и других возможностей, направленных на упрощение разработки сценариев.

Список литературы

1. Computer Languages History. URL: https://www.levenez.com/lang/ (дата обращения: 04.05.2017).

2. Муромцев В.В., Ломакин В.В., Цоцорина Н.В. Разработка специализированного языка для удаленного программирования микроконтроллеров // Научные ведомости Белгородского государственного университета. Серия: Экономика. Информатика. 2011. Т. 19. № 13-1 (108). С. 180-185.

3. Сценарные языки программирования на сайте Игоря Гаршина. Скрипты. URL: http://www.garshin.ru/it/computer-languages/scripts.html (дата обращения: 04.05.2017).

4. Седых А. Обзор языков описания сценариев // Образование, наука, производство Белгородский государственный технологический университет им. В.Г. Шухова. 2015. С. 2714-2719.

5. Гевлич М.В., Рязанов Ю.Д. Интерпретатор скриптов графической системы // Информационные технологии в управлении и моделировании: Сб. докл. Международной науч.-технич. интернет-конф. – Белгород: Изд-во БГТУ им. В.Г. Шухова, 2005. – С.54–56.

6. Непейвода. Н. Н. 13. Лекция: Событийное программирование // Стили и методы программирования. курс лекций. учебное пособие. – М.: Интернет-университет информационных технологий, 2005. – С. 213-222. – 316 с. – ISBN 5-9556-0023-X.

7. Callback Hell. URL: http://callbackhell.com/ (дата обращения: 04.05.2017).

8. Неизменность и чистота <- О Haskell по-человечески. URL: https://www.ohaskell.guide/immutability-n-purity.html (дата обращения: 04.05.2017).

9. Замыкания в php / Блог компании Mail.Ru Group / Хабрахабр. URL: https://habrahabr.ru/company/mailru/blog/103983/ (дата обращения: 04.05.2017).

10. Ликбез по типизации в языках программирования / Хабрахабр. URL: https://habrahabr.ru/post/161205/ (дата обращения: 04.05.2017).

11. SoftKey.info: Статьи - Рефакторинг в Visual Studio. URL: http://www.softkey.info/reviews/review5449.php (дата обращения: 04.05.2017).

12. Null References: The Billion Dollar Mistake. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare (дата обращения: 04.05.2017).

13. Exploring the Abyss of Null and Undefined in JavaScript - Modern Web. URL: https://modernweb.com/exploring-the-abyss-of-null-and-undefined-in-javascript/ (дата обращения: 04.05.2017).
14. Null, великий и ужасный / Хабрахабр. URL: https://habrahabr.ru/post/309462/ (дата обращения: 04.05.2017).
15. Значение Null. Nullable-типы. Оператор ??. URL: http://mycsharp.ru/post/47/2014_09_30_znachenie_null_nullable-tipy_operator___.html (дата обращения: 04.05.2017).
16. Haskell/Understanding monads/Maybe - Wikibooks, open books for an open world. URL: https://en.wikibooks.org/wiki/Haskell/Understanding_monads/Maybe (дата обращения: 04.05.2017).
17. Expressions - Apache FreeMarker Manual. URL: http://freemarker.org/docs/dgui_template_exp.html#dgui_template_exp_missing (дата обращения: 04.05.2017).
18. Charlie Hunt, Binu John. Java Performance. Prentice Hall, 2012, 693 P.
19. Нативные методы и вызовы нативного интерфейса Java (JNI) в Android приложении | Src-CODE.Net. URL: http://src-code.net/nativnye-metody-i-vyzovy-nativnogo-interfejsa-java-jni-v-android-prilozhenii/ (дата обращения: 04.05.2017).
20. Selectors | jQuery API Documentation. URL: https://api.jquery.com/category/selectors/ (дата обращения: 04.05.2017).
21. Анатомия игровых движков / Игры. URL: https://3dnews.ru/games/engines/ (дата обращения: 04.05.2017).
22. Interrupted Exception. URL: http://bazhenov.me/blog/2009/09/04/interrupted-exception.html (дата обращения: 04.05.2017).
23. CSS Selectors Reference. URL: https://www.w3schools.com/cssref/css_selectors.asp (дата обращения: 04.05.2017).
24. Введение в Java Reflection API. URL: http://www.quizful.net/post/java-reflection-api (дата обращения: 04.05.2017).
25. Операции над множествами - MathHelpPlanet. URL: http://mathhelpplanet.com/static.php?p=operatsii-nad-mnozhestvami (дата обращения: 04.05.2017).
26. Parlante, Nick. Linked List Basics. Stanford CS Education Library (2001). 26 P.
27. Функции высшего порядка | Выразительный Javascript. URL: https://karmazzin.gitbooks.io/eloquentjavascript_ru/content/chapters/chapter5.html (дата обращения: 04.05.2017).
28. Каррирование и частичное применение | Src-CODE.Net. URL: http://src-code.net/karrirovanie-i-chastichnoe-primenenie/ (дата обращения: 04.05.2017).
29. Васильев А. Н. Java: объектно-ориентированное программирование : для магистров и бакалавров : базовый курс по объектно-ориентированному программированию. СПб: Издательский дом "Питер", 2011. 395 с.
30. Асинхронное программирование. URL: http://javascript.ru/unsorted/async (дата обращения: 04.05.2017).
31. Аспектно-ориентированное программирование (АОП): Для чего его лучше использовать? URL: https://www.ibm.com/developerworks/ru/library/pollice/ (дата обращения: 04.05.2017).
32. Java Programming Environment and the Java Runtime Environment (JRE) (JDK 1.1 for Solaris Developer's Guide). URL: https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html (дата обращения: 04.05.2017).

References

1. "Computer Languages History", last modified May 04, 2017, https://www.levenez.com/lang/.
2. Muromcev V.V., Lomakin V.V., Tsotsoryna N.V. Developing of specified language for microcontroller remote programming. Scientific bulletins of the Belgorod State University. Series: Economics. Informatics. T. 19. № 13-1 (2011): p. 180-185.
3. "Scenario programming languages on the site of Igor Garshin. Scripts", last modified May 04, 2017, http://www.garshin.ru/it/computer-languages/scripts.html.
4. Sedykh A. Overview of scripting languages. Education, science, and production of Belgorod State Technological University. named after V.G. Shoukhov (2015): p. 2714-2719.
5. Gievlich M.V., Ryazanov U.D. Script interpreter for the graphics system. Information technologies in management and modeling. The collection of reports of the scientific and technical conference. Belgorod: publishing house of BSTU named after V.G. Shoukhov (2005): p.54–56.
6. Nepeyvoda N. N. 13. Lection: Event-driven programming. Programming styles and methods. Lecture course. Tutorial. Moscow: Internet University of Information Technologies (2005): p. 213—222. ISBN 5-9556-0023-X.
7. "Callback Hell", last modified May 04, 2017, http://callbackhell.com/.
8. "Immutability and purity <- simply about Haskell", last modified May 04, 2017, https://www.ohaskell.guide/immutability-n-purity.html.
9. "Php closures / Mail.Ru Group company blog / Habrahabr", last modified May 04, 2017, https://habrahabr.ru/company/mailru/blog/103983/.
10. "Educational program about programming languages type systems / Habrahabr", last modified May 04, 2017, https://habrahabr.ru/post/161205/.

11. "SoftKey.info: Articles - Refactoring in Visual Studio", last modified May 04, 2017, http://www.softkey.info/reviews/review5449.php.
12. "Null References: The Billion Dollar Mistake", last modified May 04, 2017, https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.
13. "Exploring the Abyss of Null and Undefined in JavaScript - Modern Web", last modified May 04, 2017, https://modernweb.com/exploring-the-abyss-of-null-and-undefined-in-javascript/.
14. "Null, the great and terrible / Habrahabr", last modified May 04, 2017, https://habrahabr.ru/post/309462/.
15. "Null value. Nullable types. ?? operator", last modified May 04, 2017, http://mycsharp.ru/post/47/2014_09_30_znachenie_null_nullable-tipy_operator___.html.
16. "Haskell/Understanding monads/Maybe - Wikibooks, open books for an open world", last modified May 04, 2017, https://en.wikibooks.org/wiki/Haskell/Understanding_monads/Maybe.
17. "Expressions - Apache FreeMarker Manual", last modified May 04, 2017, http://freemarker.org/docs/dgui_template_exp.html#dgui_template_exp_missing.
18. Charlie Hunt, Binu John. Java Performance. Prentice Hall (2012). 693 P.
19. "Native methods and call of native Java interfaces with JNI in Android application | Src-CODE.Net", last modified May 04, 2017, http://src-code.net/nativnye-metody-i-vyzovy-nativnogo-interfejsa-java-jni-v-android-prilozhenii/.
20. "Selectors | jQuery API Documentation", last modified May 04, 2017, https://api.jquery.com/category/selectors/.
21. "Game engine anatomy / Games", last modified May 04, 2017, https://3dnews.ru/games/engines/.
22. "Interrupted Exception", last modified May 04, 2017, http://bazhenov.me/blog/2009/09/04/interrupted-exception.html.
23. "CSS Selectors Reference", last modified May 04, 2017, https://www.w3schools.com/cssref/css_selectors.asp.
24. "Introduction into Java Reflection API", last modified May 04, 2017, http://www.quizful.net/post/java-reflection-api.
25. "Operations under the mathematical sets - MathHelpPlanet", last modified May 04, 2017, http://mathhelpplanet.com/static.php?p=operatsii-nad-mnozhestvami.
26. Parlante, Nick. Linked List Basics. Stanford CS Education Library (2001). 26 P.
27. "Higher order functions | Eloquently Javascript", last modified May 04, 2017, https://karmazzin.gitbooks.io/eloquentjavascript_ru/content/chapters/chapter5.html.
28. "Currying and partial application | Src-CODE.Net", last modified May 04, 2017, http://src-code.net/karrirovanie-i-chastichnoe-primenenie/.
29. Vasyl'ev A.N. Java: object-oriented programming: for master and bachelor degrees : basic course of object-oriented programming. St. Peterburg: Publisching house "Piter", 2011. p. 395.
30. "Asynchronious programming", last modified May 04, 2017, http://javascript.ru/unsorted/async.
31. "Aspect-oriented programming (AOP): Best use cases", last modified May 04, 2017, https://www.ibm.com/developerworks/ru/library/pollice/.
32. "Java Programming Environment and the Java Runtime Environment (JRE) (JDK 1.1 for Solaris Developer's Guide)", last modified May 04, 2017, https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html.

Седых Артём, магистрант кафедры программного обеспечения вычислительной техники и автоматизированных систем
Рязанов Юрий Дмитриевич, доцент, доцент кафедры программного обеспечения вычислительной техники и автоматизированных систем,

Sedykh Arteom, Master Degree Student
Ryazanov Yury Dmitrievich, Associate Professor, Department of Software Computer Technology and Automated Systems