

ИНФОРМАТИКА,  
ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА  
И УПРАВЛЕНИЕ

INFORMATICS,  
COMPUTER ENGINEERING  
AND CONTROL

УДК 004.43

DOI: 10.17212/1814-1196-2018-1-117-136

## Функционально-императивный язык программирования EI\*

А.А. МАЛЯВКО

630073, РФ, г. Новосибирск, пр. Карла Маркса, 20, Новосибирский государственный технический университет, кандидат технических наук, доцент. E-mail: [a.malyavko@corp.nstu.ru](mailto:a.malyavko@corp.nstu.ru)

В работе обсуждается тенденция взаимопроникновения идей и технологий функциональной и императивной парадигм программирования в современные их реализации. Предлагается новый функционально-императивный язык программирования EI, во многом похожий на функциональный язык Erlang, но отличающийся от него рядом особенностей. Приводится краткое описание лексики, синтаксиса и семантики языка EI. Выделяются его основные отличительные черты и характеристики: чистые функции, функции первого и высшего порядка, анонимные функции, замыкания, перегрузка сигнатур функций, локальность сбора мусора, императивность выполнения операций в теле функции, высокоуровневые типы данных и операции над ними, вариативность статической и динамической типизации, а также иммутабельности переменных по выбору программиста, наличие примитивных и высокоуровневых типов данных и операций над ними, немногословность, простота и удобство управляющих конструкций, возможность явного определения циклов вместо более дорогостоящих рекурсивных вызовов функций, которые, впрочем, тоже можно использовать. Описывается типовая структура файла, содержащего программный модуль; приводится перечень и назначение его секций, определения функций, все виды управляющих операторов языка, сводка существенных отличительных характеристик языка. Описывается текущее состояние разработки и реализации транслятора программ с языка EI для разных целевых платформ с использованием инфраструктуры компиляторов LLVM.

**Ключевые слова:** язык программирования, функциональная парадигма, императивная парадигма, лексика, синтаксис, семантика, выражение, оператор, компилятор

## ВВЕДЕНИЕ

В сосуществовании императивной и функциональной парадигм программирования [1], начавшегося с момента появления двух самых первых алгоритмических языков: императивного FORTRAN и функционального LISP, сложилась ситуация противостояния [2, 3]. Эти парадигмы и реализующие их языки часто противопоставляются друг другу, и делается это обыч-

---

\* Статья получена 20 августа 2017 г.

но с целью возвысить преимущества одной парадигмы на фоне недостатков второй [4]. В то же время наблюдается «взаимопроникновение» идей и технологий противопоставляемых парадигм как в императивные, так и в функциональные языки [5]. К таким явлениям можно относить, например, замыкания и лямбда-функции в императивных языках [6], монады и продолжения для выполнения некоторых операций строго в требуемой последовательности или различные способы создания побочных эффектов в функциональных языках [7]. Добавление все новых и новых расширенных возможностей на изначально заложенный «узкий» фундамент того или иного языка постепенно приводит к его росту до необозримости и трудностям в практическом использовании. Иногда декларируемые при этом цели «теряются» или искажаются по мере реализации средств их достижения.

В связи с этим представляет интерес создание языка, изначально обладающего достоинствами (хотя бы некоторыми) обеих парадигм и, соответственно, отсутствием хотя бы некоторых их недостатков. В этой работе описывается язык программирования El, возникший как результат анализа известных достоинств и недостатков функционального языка Erlang [8, 9], называемого далее прототипом. При описании тех конструкций предлагаемого языка, которые существенно отличаются от конструкций прототипа, обосновываются причины и описываются ожидаемые следствия. В предлагаемом языке предпринята попытка совместить в разумной степени отличительные черты функциональной и императивной парадигмы. От функциональной парадигмы взяты чистые функции, функции первого и высшего порядка, возможность работать с иммутабельными переменными; от императивной – строго последовательное исполнение операторов тела функции, возможность организовывать циклические вычисления без использования рекурсивных вызовов функций, возможность оперировать с мутабельными (но локальными в теле функции) переменными, вариативность типизации. Некоторые лексические и синтаксические конструкции языка El похожи на свои аналоги в языках Java, C/C++, C# и ряда других.

## 1. ЛЕКСИКА ЯЗЫКА EL

Ниже приведены лексические единицы (слова) языка.

1. Идентификаторы: последовательности букв, цифр или символа подчеркивания «\_», начинающиеся с буквы (неважно, прописной или строчной) или с подчеркивания. В отличие от Erlang, идентификаторы могут начинаться и со строчных букв, а символ «\_» никакой специальной роли не играет.

2. Ключевые слова: выделенные идентификаторы, играющие особую роль в тексте программы, например *when*, *by*, *of*, *int* и ряд других. Ключевые слова не могут использоваться, например, для именования переменных или функций.

3. Числа: вещественные в обычной или научной форме, целые десятичные и целые в заданной системе счисления вида *<основание> \$ <значение>*. Эта форма представления численных литералов унаследована из прототипа, только вместо символа «#» используется символ «\$».

4. Атомы: непустые последовательности произвольных UNICODE-символов, обязательно заключенные в одинарные апострофы. Внешний вид ато-

мов несколько отличается от Erlang, в котором атомы – это просто последовательности букв, начинающиеся со строчной буквы, или другие последовательности, заключаемые в одиночные апострофы. Некоторые атомы, например *'true'*, *'false'*, *'number'*, *'string'* и ряд других, играют специальные роли в программах на языке El, для двух атомов даже введены ключевые слова – аналоги. Это логические значения *true* и *false*.

5. Строки: последовательности произвольных UNICODE-символов, заключенные в двойные апострофы, например: "string", "это\u0020строка\u0020символов" или "это тоже строка символов: !'№;%:?'\*(".

6. Отдельные символы и последовательности символов, используемые в качестве знаков операций, скобок, разделителей и ограничителей. Некоторые допустимые слова этой группы демонстрируются далее при описании синтаксических конструкций языка.

Между словами в тексте программы могут располагаться однострочные или блочные комментарии. Однострочный комментарий начинается с двух символов *«//»*, может содержать произвольные символы и заканчивается переводом строки. Блочный комментарий начинается с двух символов *«/\*»*, заканчивается символами *«\*/»*, может содержать произвольные символы, в том числе переводы строки, строчные комментарии и, в отличие от многих других языков, – вложенные блочные комментарии.

Синтаксическими единицами языка являются выражения, операторы, блоки операторов и функции.

## 2. СТРУКТУРА ПРОГРАММ НА ЯЗЫКЕ EL

Компилируемой единицей является модуль, текст которого должен целиком содержаться в одном файле (имена модуля и файла не обязательно должны совпадать). Модуль в определенном смысле представляет собой класс, хотя и может использоваться без явного создания экземпляра. Явное создание экземпляра модуля тоже возможно, при этом явно и неявно созданные экземпляры могут (но не обязаны) различаться состояниями (значениями неизменяемых переменных, которые доступны по чтению во всех функциях модуля, но невидимы извне).

В файле текста программы на языке El составляющие части (разделы) модуля должны следовать друг за другом строго в указанном порядке:

*<имя модуля><опции>@<импорт>@<инициализаторы>@<функции>*

Здесь и далее символ *@* в качестве верхнего индекса обозначает необязательно присутствующий раздел. Раздел *<опции>* в этой работе не рассматривается, он предназначен для передачи компилятору параметров трансляции и сборки.

Именованное и импортное модулей необходимо для организации межмодульных связей при компиляции и сборке программы. Соответствующие предложения в тексте могут выглядеть, например, так:

```
module neuroNet;  
import neurons, neurons.synapses;  
import neurons.training;
```

«Вложенности подмодулей» в модули, которую можно «увидеть» в приведенных предложениях импорта, на самом деле не существует. Другими

словами, модули *neurons* и *neurons.synapses* во всех смыслах равноправны, второй не является частью первого. Имена модулей с использованием точки могут использоваться исключительно для удобства разбиения большого количества функций и программных единиц на разумно организованные совокупности. Объявления импорта модулей нужны компилятору и сборщику в случае, если из функций данного модуля вызываются функции, входящие в другие модули. В то же время существует возможность вызова функций из другого модуля без его импортирования, она описана далее.

Между произвольными двумя строками текста модуля могут находиться макроопределения, записываемые каждое в отдельной строке (или в нескольких последовательных строках). Механизм макросов очень похож на свой аналог в C/C++, но имеется несколько отличий. Во-первых, для создания макроопределения используется ключевое слово *macro* (не начинающееся с символа «#» как *#define*), которое должно находиться в самом начале строки (первые 5 символов). Во-вторых, в списке аргументов макроопределения допустимы табуляции и пробелы. В третьих, при попытке использования рекурсивных макровыводов первый вложенный макрос не заменяется макрорасширением. Это может, но не обязательно повлечь выдачу синтаксических или семантических ошибок в период компиляции для предложения, в котором содержится макровывод рекурсивно определенного макроса. Сообщений об ошибках или предупреждений, связанных с каким-либо некорректным использованием механизма макросов, компилятор не выдает.

В любой точке текста модуля в качестве отдельной строки может появиться: *include <путь\_и\_имя\_файла> //слово include без #, как и macro*

Текст указанного файла вставляется вместо этой строки, после чего обрабатывается как часть транслируемого модуля. Включаемый файл может включать другие файлы, глубина вложенности ничем не ограничена, но попытка рекурсивного включения тоже блокируется с выдачей предупреждения. При обнаружении каких либо ошибок во включенном тексте компилятор формирует диагностическое сообщение, содержащее указанные в предложениях *include* имена всех включенных файлов (вплоть до точки возникновения ошибки) и номера строк. Отсутствие включаемого файла по указанному пути не рассматривается как ошибка, но компилятор выдает предупреждение.

Раздел *<инициализаторы>* по назначению очень похож на объектные инициализаторы в языке Java. Он содержит анонимные функции (см. ниже), исполняемые при создании экземпляра модуля. Нужная анонимная функция выбирается сопоставлением списков формальных и фактических аргументов (фактические аргументы указываются при создании экземпляра модуля путем вызова встроенной функции *new*). Иммутабельные переменные, получающие значения в инициализаторах, видимы во всех функциях модуля и могут рассматриваться как константные поля модуля.

### 3. СИНТАКСИС И СЕМАНТИКА ЯЗЫКА

Синтаксическими единицами языка являются функции, блоки операторов, выражения и операторы.

### 3.1. ОПРЕДЕЛЕНИЯ ФУНКЦИЙ МОДУЛЯ

Основным в тексте модуля является раздел *<функции>*, в котором могут быть определены одна или несколько функций. Функции объявляются так:

*<модификатор доступа>@ <заголовок> <тело функции>*

В качестве модификатора доступа используется одно из ключевых слов – *public* или *private*. Функции, объявленные как *public*, могут быть вызваны из любого другого модуля, а как *private* – только из данного модуля. Отсутствие модификатора трактуется как объявление со словом *private*.

Заголовок функции представляет собой идентификатор, за которым в круглых скобках следует возможно пустой кортеж формальных аргументов – последовательность выражений языка через запятую. Модуль может содержать любое количество определений функций с одним и тем же идентификатором и даже с одним и тем же набором формальных аргументов. В процессе исполнения программы при вызове функции с этим идентификатором выполняется последовательное сопоставление кортежа фактических аргументов с образцами – кортежами формальных аргументов разных определений функции. Сопоставление осуществляется в порядке следования объявлений функции с этим именем в файле. Это позволяет иметь в модуле несколько версий функции с одним и тем же образцом формальных аргументов, размещая нужную версию перед другими – полезная возможность, облегчающая получение метрик или отладку программы. В этом язык E1 отличается и от прототипа, и от большинства других языков, в которых обычно не допускается перегрузка функций с одинаковыми сигнатурами. При первом положительном результате сопоставления выполняется непосредственно следующее за этим заголовком тело функции, которое в общем случае представляет собой цепочку блоков условного выполнения.

Каждый такой блок выглядит следующим образом:

*<охранник> { <объявления переменных>@ <последовательность операторов> }*

Охранник (охранное выражение) имеет вид:

*when <выражение> :*

Для того чтобы охраняемый блок был выполнен, необходимо, чтобы выражение в охраннике имело значение *'true'* (или просто *true*). Любые другие значения считаются эквивалентными значению *'false'* (*false*).

Выполнение выбранного по подходящим аргументам тела функции состоит в последовательном (в текстуальном порядке) вычислении значений охранных выражений до тех пор, пока одно из них не окажется истинным. Тогда выполняется непосредственно следующая за охранником последовательность операторов, заключенная в фигурные скобки (блок), выполнение функции на этом завершается, а в вызывающую функцию возвращается значение, выработанное последним выполненным оператором блока.

Охранник последнего блока в цепочке может быть пустым, это эквивалентно охранному выражению *when true* :. Если такого блока нет и охранники всех блоков тела выбранной по аргументам функции ложны, то в вызывающую функцию возвращается значение *nothing*.

Вот простейший пример функции с несколькими охраняемыми блоками:

```
threeLevel(?(n) == 'number') //Выражение ?(n) возвращает тип переменной n
  when n < 0 : { -1; }        //Охранник этого блока: when n < 0 :
                              //Этот блок возвращает значение выражения -1.
  when n == 0 : { 0; }        //Этот блок возвращает значение 0.
  { 1; }                      //Это неохранный блок, возвращающий значение 1.
```

Если функция *threeLevel* вызывается с единственным аргументом, тип которого – целое или вещественное число (только в этом случае сопоставление аргумента с образцом *?(n) == 'number'* будет успешным), то начнется выполнение тела, содержащего два охраняемых и один неохранный блок. Нужный блок будет выбран путем вычисления значений охранников, сравнивающих фактический аргумент с нулем. В результате будет возвращено одно из трех значений -1, 0 или 1. Заметим, что в модуле могут существовать другие определения функции *threeLevel*: без аргументов с двумя или более аргументами или с одним аргументом, но не числовым. При вызове любой из них сопоставление фактических аргументов с образцом *?(n) == 'number'* не будет успешным.

Точно такого же результата можно было бы добиться и путем написания единственного неохранный блока в качестве тела этой функции, выполнив внутри блока проверку значения полученного аргумента. Однако использование охранных выражений позволяет писать более короткие тела функций.

### 3.2. ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННЫХ В ФУНКЦИЯХ

Функции в языке El оперируют только с локальными переменными (фактические аргументы тоже локальны, потому что передаются исключительно по значению) и именованными константными полями экземпляра модуля. Таким образом, функции (за исключением некоторых встроенных, осуществляющих, например, ввод-вывод информации) в принципе не могут порождать побочного эффекта. В момент выхода из функции, сопровождаемого возвратом некоторого значения (которое далее используется как локальное в вызывающей функции), все ее локальные переменные становятся ненужными и занятая ими память освобождается. В этом и состоит сбор мусора в программах на языке El. Глобальных переменных в языке El нет вообще, поэтому нет необходимости в других видах сбора мусора.

Первое появление имени переменной начинает (открывает) область ее видимости, заканчивающуюся в конце блока и, возможно, прерывающуюся во вложенных блоках, содержащих явное объявление с этим же именем.

Необъявленные переменные блока по умолчанию являются неизменяемыми (иммутабельными). Этого же эффекта можно добиться, явно объявляя переменную в секции *<объявления переменных>* блока с использованием ключевого слова *def*, например, так:

```
def myVariable;
```

При необходимости с помощью ключевого слова *var* для какой-либо переменной можно указать, что в блоке допускается переприсваивание ее значения. Это выглядит так:

```
var eventClass;
```

Локальные переменные функции, имеющие любой, явно или неявно объявляемый тип, как изменяемые так и иммутабельные, могут быть объявлены с ключевым словом *static*. Значения таких переменных будут сохраняться в промежутке времени между разными вызовами данной функции. Такие переменные локальны, т. е. недоступны для использования в любых других функциях.

### 3.3. ТИПИЗАЦИЯ В ЯЗЫКЕ EL

В языке El существуют данные следующих типов: числа, атомы, строки, векторы (массивы однотипных элементов), списки, кортежи, бинарники, функции, модули. Тип значения переменной может быть не задан явно. Тогда при первом присваивании она получает и значение, и его тип. Иммутабельные переменные после этого не могут получить ни другое значение, ни другой тип. Мутабельные переменные, определенные без указания типа, могут получать и другое значение, и другой тип.

Явно можно задавать типы *int*, *long*, *bigint*, *float*, *double*, *bigfloat*, *string*, *atom*, *list*, *tuple*, *binary*, *function*, *module*; способы записи объявления переменной с указанием ее типа:

```
var long eventCount;  
def tuple vendorData, customerData;
```

Тип данных вектор задается не ключевым словом *vector*, а специальной нотацией, определяемой при рассмотрении данных этого типа.

Явно заданные переменные численных типов *int*, *long*, *float*, *double* называются примитивными, все остальные – объектными.

Если тип переменной задан явно, то присваиваемое ей значение должно иметь точно такой же тип (или неявно преобразуемый к нему, например *int* -> *long* или *int* -> *double*), иначе выбрасывается исключение времени выполнения. Исключения можно перехватывать и обрабатывать с использованием оператора *try-catch-finally*. В то же время для предотвращения возникновения таких исключений присваиваемые значения можно явно приводить к нужному типу. Для этого нужно использовать постфиксный знак операции «!*'<имя\_типа>'*», например:

```
eventCount += anyValue!'int';
```

Значение переменной *anyValue*, каким бы ни был его тип, перед прибавлением его к *eventCount* преобразуется в целое. Явные преобразования, так же как в С-подобных языках, могут сопровождаться потерей данных или точности. Неявные преобразования возможны, но только в пределах типа *number*, к которому принадлежат все его численные подтипы *int*, *long*, ....

Явное задание типа элементов массивов (векторов) преследует цель уменьшение объемов памяти на их хранение и затрат времени на обработку. В том случае, если в выражениях, значения которых используются для формирования значений элементов вектора, используются только неявно преобразуемые друг в друга типы, компилятор может построить соответствующие фрагменты целевой программы без проверок возможности возникновения исключений (естественно, кроме непроверяемых во время компиляции, например деление на ноль).

Явно можно также задавать, что переменная имеет тип записи (полный аналог конструкции *record* в языке Erlang и близкий к понятию *struct* в C/C++). Объявления записей (это один из инициализаторов модуля) могут появляться в начале файла до объявления первой функции. Пусть запись объявлена так:

```
record authorData(firstName, lastName, birthDay = (day, month, year), authorId);
```

Тогда переменная, с использованием которой обрабатываются данные об авторах книг, может быть объявлена так:

```
var authorData author; //или так:  
def authorData author; //если ее не нужно переприсваивать
```

Теперь внутри блока к значениям полей записи *author* можно обращаться по имени как к полям объектной переменной:

```
author.firstName (или author.birthDay.year).
```

Тип значения любого выражения (в частности, переменной или константы) во время исполнения программы можно получить с использованием знака операции «*?(expression)*». Эта операция возвращает один из атомов '*number*', '*atom*', '*string*', '*vector*', '*list*', '*tuple*', '*binary*', '*function*', '*module*', '*nothing*'. Например, использование этого знака операции для вышеопределенной переменной типа *authorData* в виде *?(author)* вернет атом '*tuple*'. Тип значения '*nothing*' соответствует отсутствию значения, т. е. несвязанной переменной в терминах функционального программирования. В языке Erlang этому соответствует понятие *unbound*.

### 3.4. ТИПЫ ДАННЫХ

В описании типов данных будем использовать понятия выражения и оператора присваивания, в том числе присваивания с вычислениями.

1. Числа относятся к обобщенному типу *number* и могут быть целыми (одного из трех подтипов *int*, *long*, *bigint*) или вещественными (трех подтипов *float*, *double*, *bigfloat*). Подтипы различаются диапазонами (и для вещественных – точностью) представления значений. Подтип *int* соответствует 32-битному представлению целых чисел, *long* – 64-битному, *bigint* – представлению, занимающему столько памяти, сколько необходимо для данного числа. Аналогично, подтип *float* предполагает 32-битное представление (согласно стандарту IEEE 754), *double* – 64-битное и *bigfloat* – представление с требуемой точностью. Все числа хранятся и обрабатываются со знаком. Если подтип не определен явно, то для целых чисел используется формат *bigint*, а для вещественных – *bigfloat*. Обработка чисел в форматах *bigint* и *bigfloat* требует значительно больших расходов как по памяти, так и по времени, поэтому во всех тех случаях, когда диапазон значений при написании программы может быть оценен как ограниченный сверху, имеет смысл явно определять подходящий подтип для численных переменных. Формат численных литералов (констант) определяется компилятором.

2. Атомы – неизменяемые элементы данных, представляющие любые последовательности символов, заключенные в одинарные апострофы. К атомам применимы только операции сравнения и простого присваивания их переменным. В выражениях присваивания атом может находиться только спра-

ва от знака операции «=». Например, верно: *month* = 'сентябрь', неверно: *month* += 'сентябрь' или 'сентябрь' = 'октябрь'. Операции сравнения (применяемые как к атомам, например 'сентябрь' > 'октябрь', так и к значениям других типов) вырабатывают атомарные значения 'true' или 'false'. Выражение 'сентябрь' > 'октябрь' имеет значение 'true', поскольку атомы сравниваются лексикографически. Для удобства можно считать, что в любом месте доступны две имутабельные переменные: *true* со значением 'true' и *false* со значением 'false'.

3. Тип данных 'string' (строка) отличается от типа «атом», во-первых, способом записи литералов; во-вторых, тем, что над строками кроме операций сравнения и присваивания определена еще операция сцепления (конкатенации), обозначаемая символом «+», и, в третьих, тем, что любая строка является одновременно вектором, содержащим символы – элементы целочисленного подтипа *int*. В силу этого к строкам применимы все операции и встроенные методы, которые могут выполняться над векторами, рассматриваемыми ниже. Строковые литералы – это последовательности любых символов Юникода, заключаемые в двойные кавычки, например "this is string", правила их оформления заимствованы из языка Java.

4. Данные типа 'vector' представляют собой массивы однотипных элементов. Многомерные массивы могут быть как ступенчатыми, так и прямоугольными. Ступенчатый вектор можно рассматривать как массив векторов на единицу меньшей размерности. Размерность вектора фиксируется в момент его создания и не может быть изменена. Имутабельность всех элементов вектора определяется тем, описан он с использованием ключевого слова *var* или нет. Векторы объявляются путем указания количества размерностей и, возможно, количества элементов по размерностям следующим образом.

Объявления ступенчатых векторов:

```
var@ min@ [<размер>@][<размер>@]...идентификатор<, идентификатор ...>;
```

Объявления прямоугольных векторов:

```
var@ min@ [<размер>@, <размер>@, ]...идентификатор<, идентификатор ...>;
```

Примеры объявлений векторов:

```
var int[10, 11] matrix; //целочисленный изменяемый прямоугольный массив
                        //размерами 10*11 элементов
def authorData[] authors; //одномерный массив неизменяемых записей,
//его размерность будет определена при создании с помощью функции new
var float[numSensors] sumIndication, currIndication; //два вектора, видимо
//предназначенные для хранения показаний датчиков
def [sizeX][sizeY][sizeZ] volume; //трехмерный ступенчатый массив
//неизменяемых элементов, тип которых будет
//фиксирован при первой записи значения в любой элемент
```

Создание векторов выполняется неявно, если все размерности и размеры по ним заданы в объявлении (так создаются все массивы из примеров, кроме *authors*). Если же размеры по размерностям явно не заданы, то вектор может быть создан явно путем вызова встроенной функции *new*, например:

```
authors = new( authorData[123] ); //создается //одномерный массив из 123 записей
```

К вектору в целом могут применяться все знаки операций, если они применимы к его элементам и при условии совпадения размерностей и раз-

меров в том случае, если операция применяется к двум массивам. Например, оператор:

```
matrix -= I;
```

вычитет единицу из каждого элемента объявленного выше массива (это значит, что не нужно писать два вложенных цикла, перебирающих строки и столбцы вектора).

В языке El для облегчения программирования операций с массивами предусмотрены еще и следующие возможности:

- использование обозначений нижней «*\_*» и верхней «*^*» границ индексов внутри индексных выражений (например, *autors*[*^*] определяет обращение к последнему элементу вектора *autors*, а *matrix*[*\_*][*^*] обозначает последний элемент самой первой строки двумерного массива *matrix*);

- получение нижней «*lo()*» и верхней «*hi()*» границ изменения индексов вектора, а также количества элементов в нем «*size()*» для какой-либо обработки (например, оператор *autorsCount* = *autors.size()*, который занесет количество записей в массиве *autors* в переменную *autorsCount*);

- задание диапазонов изменения индексов обрабатываемых элементов с помощью обозначения «*..*», используемого между начальной и конечной границами (включительно). Границы могут задаваться произвольными целочисленными выражениями. Например, оператор сложения двух векторов можно записать так:

```
sumIndication[_..^] += currIndication[_..^];
```

В индексном выражении может быть указан шаг перебора. Например, так: [*^*-1 .. *\_*+1 *step* 2]. Здесь указано, что индексы обрабатываемых элементов должны изменяться с шагом 2 начиная с предпоследнего (*^*-1) и до второго (*\_*+1). Направление перебора элементов определяется компилятором по соотношению заданных нижней и верхней границы. Если нижняя граница меньше верхней, то при переборе индекс увеличивается на значение шага, иначе – уменьшается.

- использование ссылок вида «*#*» («*##*», «*###*», ...) на ранее заданные в текущем операторе индексные выражения. Количество символов *#* в ссылке означает порядковый номер индексного выражения в операторе. Тот же оператор сложения векторов с использованием ссылок можно переписать так:

```
sumIndication[_..^] += currIndication[#];
```

Более сложный пример использования индексных ссылок – запись на языке El так называемого прямого хода метода Гаусса решения системы линейных алгебраических уравнений. Пусть *matrix* – двумерный изменяемый вектор коэффициентов и свободных членов решаемой системы размерностью *N* строк на (*N*+1) столбцов. Тогда приведение этого вектора к нижнему треугольному виду (ниже главной диагонали – все нули) можно задать одним оператором:

```
matrix[_+1 .. ^] -= matrix[_..#-1] * matrix[#][##] / matrix[##][##];
```

В этом операторе определено следующее:

- перечисляются все строки, кроме первой (имеющей индекс *matrix*[*\_*]);
- для вычисления нового значения каждой строки используются все предыдущие строки, поскольку индексное выражение *matrix*[*\_*..*#*-1] в правой части оператора присваивания не совпадает с выражением в левой части,

но тоже определяет перебор и ссылается на самый первый индекс оператора как на верхнюю границу;

- перебор предыдущих строк выполняется в порядке возрастания их индексов от начального до индекса перевычисляемой строки без единицы ( $\_.. \#-1$ );

- каждая перебираемая строка вычитается поэлементно из данной перевычисляемой строки;

- перед этим каждый элемент вычитаемой строки умножается на элемент, находящийся в перевычисляемой строке в столбце с номером перебираемой строки ( $matrix[ \# ][ \# \# ]$ ), и делится на элемент главной диагонали вычитаемой строки ( $matrix[ \# \# ][ \# \# ]$ ).

Аналог этого оператора на С-подобном языке значительно более многословен.

5. Тип данных «список» (*'list'*). Список – это одномерный массив изменяемого в процессе выполнения программы размера, содержащий не обязательно однородные значения. Список может содержать одновременно несколько элементов любых типов, в том числе списки, векторы, кортежи, функции и т. д. Это типичные свойства списков для функциональных языков. Список можно сконструировать явно, например, так:

```
anyList = [1, 'text', ("first", "second", []), 12.345, [a, b, c], (n){n+1; /*it's function*/};
```

Здесь первый элемент – это целый литерал, второй – атом, третий – кортеж, содержащий две строки и пустой список (о кортежах – в следующем пункте описания типов данных), далее – вещественный литерал, затем – список из трех переменных, и последний элемент – анонимная функция от одного аргумента. Это так называемый нерегулярный, редко используемый вид списка. Ниже будут обсуждаться в основном регулярные списки, содержащие наборы однотипных значений, поскольку именно такие списки легко формируются с помощью эффективных средств генерации. При работе со списком можно извлечь несколько элементов (например, три первых) и записать их значения в обычные переменные (пусть это будут несвязанные переменные  $x$  и  $y$ ), поместив остаток (в виде списка) в третью несвязанную переменную  $z$ :

```
(x, 'text', y, z) = anyList;
```

Поскольку переменные  $x$ ,  $y$  и  $z$  несвязанные, сопоставление завершается успешно (вторыми элементами и кортежа, и списка является атом *'text'*). Поэтому переменная  $x$  получает значение 1, переменная  $y$  – значение *("first", "second", [])* типа кортеж и переменная  $z$  – хвост исходного списка, т. е. *[12.345, ...]*.

В языке E1 списки обладают дополнительной функциональностью: они двусвязные, допускают доступ как с головы, так и с хвоста и имеют соответствующие средства генерации (*list comprehension*). Обычное разбиение списка на голову (первый элемент) и остаток выглядит так:  $[ \textit{head} <| \textit{tail} ]$ . Разбиение на начало и хвост (последний элемент) – так:  $[ \textit{head} |> \textit{tail} ]$ . Одновременный доступ к голове, средней части (телу) и хвосту – так:  $[ \textit{head} <| \textit{body} |> \textit{tail} ]$ . Обозначения головы и хвоста могут содержать по несколько элементов, перечисляемых через запятую, как в предыдущем примере. Если голову или хвост не предполагается извлекать, то соответствующий знак операции ( $<|$  или  $<|$ ) опускается.

Для реальной эффективной работы со списками необходимо владеть средствами их генерации. В языке E1 генерация списков возможна как путем

серии добавления элементов в конец списка (образования нового хвоста на каждом шаге, знак операции «<?»), так и путем добавления элементов в начало (образования новой головы, знак операции «?»>»). Синтаксис этих способов генерации:

```
[<новая_голова> <? <источник, способ и условия формирования>]
[<источник, способ и условия формирования> ?> <новый_хвост>]
```

Полный синтаксис и семантика средств генерации списков позволяют формировать произвольные структуры данных в качестве элементов генерируемого списка и использовать функции, в том числе анонимные, в качестве источников и произвольные логические выражения в качестве условий формирования. Функции в генераторах списков дают возможность организовывать ленивые вычисления, как и в Erlang.

Над списками в целом можно выполнять операции слияния (знак операции «+») и удаления всех элементов, входящих во второй список, из первого (знак операции «-»).

6. Тип данных кортеж ('*tuple*') представляет собой совокупность как правило разнородных элементов данных. Кортежи-литералы записываются в программе в виде последовательности элементов, заключенных в круглые скобки (в отличие от Erlang, использующего фигурные скобки). Примеры кортежей-литералов приводились ранее при рассмотрении списков. Кортежи в некотором смысле аналогичны структурам в C/C++ с тем отличием, что их элементы не имеют имен. Однако использование записей и соответствующих объявлений переменных позволяет обращаться к элементам кортежей по именам. Для кортежей существует ряд методов, позволяющих узнавать количество элементов, извлекать и заменять (при условии, что переменная, хранящая кортеж, мутабельна) элементы данных по номеру.

7. Тип данных «бинарник» ('*binary*') практически полностью позаимствован из языка Erlang, некоторые отличия можно будет найти в документации по языку.

8. Функции ('*function*') представляют собой особый тип данных, с которым можно выполнять только создание, присваивание переменной, передачу в качестве аргумента в другую функцию, возврат из другой функции и, наконец, вызов/выполнения. Любая функция, определенная в модуле (и как *public*, и как *private*), может быть присвоена как значение переменной в любой другой функции этого же модуля. Кроме того, существует возможность создавать и использовать анонимные функции. Вот довольно условный пример, в котором анонимная функция возвращается в качестве результата:

```
sampleFunctionReturnsAnonimFunction ( ?( arg ) == 'number' ) {
  ( x ) when ?(x) == 'number' : { x * arg; } { 0; }; //это выражение – анонимная
    // функция, тело которой содержит два блока, один – охраняемый
}
```

И теперь пример ее использования:

```
anyFunction = sampleFunctionReturnsAnonimFunction ( 3 );
x = anyFunction( 5 ); //x получит значение 15
```

9. Тип данных '*module*' предназначен для хранения экземпляра модуля, создаваемого с помощью встроенной функции *new*. Переменная такого типа может использоваться для вызова функций именно этого экземпляра модуля.

### 3.5. ОПЕРАТОРЫ ЯЗЫКА

Секция *<последовательность операторов>* любого блока содержит именно последовательность выполняемых операторов. Перечислим и охарактеризуем операторы языка.

1. Пустой оператор. Для его обозначения используется точка с запятой «;», обычно завершающая другие операторы, если их запись согласно синтаксическим правилам не заканчивается фигурной закрывающей скобкой. Пустой оператор ничего не делает и не изменяет последнего вычисленного значения.

2. Выражение. Выражения в языке E1 конструируются обычным образом с использованием термов, круглых скобок и знаков операций. Завершается выражение-оператор точкой с запятой. Термами могут быть атомы, литералы численные, литералы строковые, индексные ссылки и границы, имена переменных (в том числе формальных аргументов и именованных констант модуля), вызовы функций (возможно, определенных в других модулях), элементы векторов, списков, кортежей, бинарников, анонимные функции. При выполнении оператора «выражение» вычисляется, если это возможно, его значение и запоминается в качестве последнего для возврата из функции. В силу того, что отнюдь не все знаки операций применимы к любому виду термов, вычисление значения выражения может вызывать исключение времени выполнения. Для перехвата и обработки исключений следует использовать оператор *«try – catch – finally»*, описываемый ниже. Знаками операций являются:

- присваивания «=», «+=», «-=», «\*=», «/=», «\=» (разделить нацело и присвоить), «%=» (вычислить остаток по модулю и присвоить), «|=», «\$=», «\$|=», «\$&=», «\$^=», «\$<=», «\$>=», «\$~» (знаки операций, начинающиеся с символа «\$», оперируют с битами);
- логические «|» и «&», «~»;
- битовые «\$|», «\$&», «\$^», «\$~», «\$<», «\$>»;
- сравнения «==», «<>», «===» (идентично, т. е. совпадают и типы, и значения), «<=>» (не идентично), «>», «>=», «<», «<=»;
- арифметические «+», «-», «\*», «/», «\», «%»;
- операции инкремента «++» и декремента «--»;
- доступ к элементам или встроенным методам списков, кортежей, векторов, бинарников и модулей «.» или «\$».

Приоритеты знаков операций в основном возрастают по мере этого перечисления, однако все унарные операции (логическое отрицание «~», битовая инверсия «\$~», унарные «+» и «-») имеют приоритет выше, чем все бинарные, но ниже приоритета операций инкремента и декремента. Операции присваивания правоассоциативны, а все остальные, для которых это понятие имеет смысл, – левоассоциативны.

3. Оператор присваивания / сопоставления с образцом. Выражение присваивания, завершающееся точкой с запятой, является оператором присваивания. В зависимости от того, могут ли переприсваиваться значения элементов выражения, находящегося слева от знака операции, и сочетания типов выражений, связанных этим знаком, перед собственно присвоением значения может выполняться сопоставление с образцом. Если в левой части есть имутабельные элементы, уже получившие какое-либо значение, то соответ-

ствующие им элементы должны иметь точно такой же тип и точно такое же значение, в противном случае выбрасывается исключение. Если в левой части есть элементы, тип которых объявлен явно, то соответствующие им значения из правой части должны иметь точно такой же или неявно приводимый к нему тип, иначе выбрасывается исключение. После успешного выполнения всех проверок новые значения записываются в элементы левой части выражения.

4. Оператор атомарной перестановки значений переменных. Операция перестановки значений, обозначаемая последовательностью литер «`:=`», выделяется среди всех остальных операций. Она введена в расчете на возможное параллельное исполнение тела функции, применима только к двум выражениям и выполняется атомарно, т. е. не может быть прервана после того, как начала исполняться. Ее выполнение можно описать как три присваивания, выполняющиеся последовательно и неразрывно друг за другом с блокировкой доступа к используемым выражениям из других активаций функции, созданием временной переменной и соблюдением всех правил сопоставления с образцом:

- временной переменной присваивается значение (и тип) левого выражения;
- левому выражению присваивается значение правого выражения;
- правому выражению присваивается значение временной переменной.

5. Оператор возврата значения из функции. Записывается так:

`return <выражение>@;`

Обеспечивает возможность управления выходом из функции с возвратом значения указанного выражения. При отсутствии выражения возвращает «*nothing*» – начальное значение любой переменной до первого присваивания.

6. Оператор ветвления вычислений. Имеются две разные формы этого оператора.

Форма 1:

```
by <::метка>@ <выражение> {
  of <выражение> : <последовательность_операторов> // ветка оператора by
  of ...           // другие ветки
  else :           // ветка else, обязательно
} // закрывающая скобка оператора by
```

При выполнении этого оператора вначале вычисляется значение выражения, указанного после ключевого слова *by*. Затем оно последовательно сопоставляется с образцами, следующими после ключевых слов *of*, обозначающих начала веток оператора, до тех пор, пока в какой-либо из них сопоставление с образцом не окажется успешным. В этом случае выполняется следующая за двоеточием после этого выражения *<последовательность\_операторов>* (вплоть до ключевого слова *of* другой ветки, ключевого слова *else* при его использовании или до фигурной скобки, закрывающей этот оператор). Если при выполнении этой последовательности не встретится один из операторов *again*, *break* или *next*, то выполняется выход из оператора *by*, т. е. начинает выполняться оператор, следующий за закрывающей фигурной скобкой. С помощью операторов *again*, *break* и *next* можно изменять это поведение. Ветка *else* должна быть последней в операторе и выполняется только в том случае, если ни одна из предшествующих веток не выполнялась.

Оператор *again* обеспечивает возврат на начало выполнения оператора *by* (т. е. на вычисление выражения, указанного после слова *by*). Такой оператор имеет смысл применять в тех случаях, когда значение этого выражения или выражений в предшествующих ветках после слов *of* могло измениться в результате выполнения *<последовательности\_операторов>*.

– *again* *<метка>* – возврат на начало управляющего оператора, помеченного указанной меткой. Таким оператором может быть текущий оператор *by* или охватывающий его оператор более высокого уровня. При отсутствии в тексте помеченного охватывающего оператора компилятор выдает синтаксическую ошибку.

– *break* – немедленный выход из данного управляющего оператора.

– *break* *<метка>* – выход из этого или охватывающего оператора, помеченного меткой.

– *next* – переход к выполнению *<последовательности\_операторов>* следующего по тексту ключевого слова *of* без проверки его образца.

Любые операторы, следующие после слов *again*, *break* и *next* вплоть до следующего слова *of* или до фигурной закрывающей скобки, закрывающей оператор *by* (что встретится раньше), игнорируются с выдачей предупреждения.

Форма 2:

```
by <::метка>@ <выражение>@ {
  when <выражение> : <последовательность_операторов> //первая ветка
  when ...           // другие ветки, необязательно
  else : <последовательность_операторов> // ветка else, необязательно
}
```

Вычисляется *<выражение>*, следующее за словом *by*, если таковое присутствует. Возвращаемое значение не запоминается и не используется для возврата из функции, в этом выражении могут быть присвоены нужные начальные значения каким-либо переменным, используемым в теле управляющего оператора. Затем вычисляется значение выражения, следующего за первым ключевым словом *when*. Если вычисленное значение не совпадает с логическим значением *true* (атом *'true'*), то выполняется переход к следующей ветке *when*, и так далее, вплоть до ветки *else* или закрывающей скобки. Если выражение в какой-либо ветке имеет значение *true*, то выполняется следующая за двоеточием *<последовательность\_операторов>* этой ветки (вплоть до следующего слова *when*, слова *else*, или до закрывающей фигурной скобки), после чего (при отсутствии операторов *again*, *break* и *next*) осуществляется выход из тела оператора *by*. Последовательность операторов ветки *else* выполняется только в том случае, если ни одно из выражений веток *when* не оказалось истинным. Как и в форме 1, *<последовательность\_операторов>* может завершаться одним из операторов *again*, *again* *<метка>*, *break*, *break* *<метка>*, *next*. Во второй форме выполнение *again* (*again* *<метка>*) приводит к переходу на вычисление выражения первой ветки *when* в соответствующем управляющем операторе, *<выражение>* после его ключевого слова *by* заново не вычисляется.

Управляющий оператор *by* позволяет конструировать аналоги условных операторов, операторов цикла и переключателей императивных языков программирования. Например, операторы

*by x <= y { of true : z = x; of false : z = y; }* //форма 1

*by { when x <= y : z = x; else z = y; }* //форма 2

строго эквивалентны условному оператору языка C/C++:

*if ( x <= y ) z = x; else z = y;*

Оператор

*by i = 0; sum = 0; { when i < 10 : sum += array[ i ]; i += 1; again; }*

выполняет ту же работу, что и цикл языка C/C++:

*for( i = 0, sum = 0; i < 10; i++) sum += array[ i ];*

Значением условного оператора в целом является значение, выработанное последним оператором, выполненным в какой-либо ветке (операторы *again*, *break* и *next* значений не вырабатывают) к моменту выхода из оператора *by*. Если не была выполнена ни одна ветка оператора *by*, то значение, хранящееся для возврата из функции, не изменяется.

7. Блок операторов (*block*). Это один оператор или последовательность операторов, заключенные в фигурные скобки и, возможно, предваряемые объявлениями переменных. Блок рассматривается как одиночный оператор языка и может использоваться везде, где может быть записан одиночный оператор. Использование блоков предоставляет возможность ограничения видимости переменных. Переменные, объявленные внутри блока, не видны вне его. Объявление переменной внутри блока скрывает переменные с таким же именем, объявленные / используемые в любом из охватывающих блоков.

8. Оператор перехвата и обработки исключений времени выполнения.

Синтаксис:

*try block*

*catch {*

*of <выражение> : <последовательность\_операторов>*

*of ...*

*}*

*finally block*

Этот оператор содержит три части: *try*, *catch* и *finally*. Его семантика практически полностью совпадает с семантикой оператора перехвата исключений в Java, отличия можно найти в руководстве по языку.

#### 4. ОСНОВНЫЕ ХАРАКТЕРИСТИКИ ЯЗЫКА EL

Представленный в работе язык программирования EL сочетает признаки функциональной и императивной парадигм, развивает некоторые привлекательные черты языка-прототипа Erlang [8] и других функциональных языков и не имеет меньшего количества недостатков. Среди достоинств языка можно отметить отсутствие сборщика мусора и в то же время — явного захвата и освобождения памяти. Конечно, платой за это являются некоторые затраты времени на захват памяти для переменных в момент входа в блок и на ее возврат при выходе из блока, но эти затраты распределены по всему времени выполнения функций программы и не будут вызывать эффекта «stop the world» [10].

Ожидаемая простота тестирования и отладки, обусловленная тем, что нет глобальных переменных и указателей, нет побочного эффекта у функций (за исключением некоторых встроенных и ряда функций стандартных модулей, осуществляющих ввод-вывод), все функции чистые. Любое значение любой переменной устанавливается только внутри данной функции. Если функция отлажена, то на ее работоспособность ничто извне повлиять не может. За это, конечно, тоже есть плата – необходимость копирования значений при передаче их в функции и возврате.

Доступны функции высшего порядка, анонимные функции, замыкания, т. е. общеизвестные достоинства функциональной парадигмы.

Реализован полиморфизм на уровне сигнатур, а не имен функций.

Реализована возможность создавать и использовать объекты, но нет ни интерфейсов, ни наследования. Борьба с дублирующимися кодами может вестись другими способами, в частности, разумной организацией совокупности модулей.

Возможность явного задания циклических вычислений вместо более дорогостоящих рекурсивных вызовов, которые тоже можно использовать.

Немногословность и простота управляющих конструкций, возможность описывать в едином стиле условные вычисления, циклические вычисления и многоветочные разветвления.

Возможность использовать статическую типизацию, в частности, с целью организации эффективной обработки данных примитивных типов.

Возможности использования встроенных высокоуровневых типов данных (списки, кортежи, бинарники, векторы, функции) и наличие высокоуровневых операций над ними, в том числе операции над векторами в целом, упрощающие их распараллеливание компилятором.

На языке *EI* возможно программирование и в чисто функциональном стиле (иммутабельные переменные, рекурсия вместо циклов, использование функций высшего порядка...), и в чисто императивном стиле (все переменные мутабельны, организация циклических вычислений без рекурсии и т. д.), и, наконец, в смешанном, императивно-функциональном стиле.

## ЗАКЛЮЧЕНИЕ

На кафедре вычислительной техники Новосибирского государственного технического университета в настоящее время ведется разработка компилятора с предложенного в данной работе функционально-императивного языка программирования *EI* для различных платформ – Linux и Windows. Лексический и синтаксический анализаторы компилятора были разработаны с использованием пакета программ автоматизации разработки трансляторов Вебтранслаб [11], также созданного автором и используемого в учебном процессе [12]. С целью обеспечения возможности генерации объектного кода для различных целевых платформ в качестве исполняющей системы объектного кода выбрана инфраструктура компиляторов Low Level Virtual Machine (LLVM) [13]. В соответствии с технологией использования LLVM, подробно описанной в работах [14, 15], разработаны все необходимые структуры данных и реализована часть функций построителя псевдокода и генератора объектного кода для некоторых типов данных и исполняемых конструкций языка.

## СПИСОК ЛИТЕРАТУРЫ

1. Себеста Р. Основные концепции языков программирования. – М.: Вильямс, 2001.
2. Кауфман В.Ш. Языки программирования. Концепции и принципы. – М.: ДМК-Пресс, 2011. – 464 с.
3. Филд А., Харрисон П. Функциональное программирование. – М.: Мир, 1993. – 637 с.
4. Wadler P. Why no one uses functional languages // ACM SIGPLAN Notices. – 1998. – Vol. 33 (8). – P. 23–27.
5. Городняя Л.В. Парадигмальная декомпозиция определения языка программирования // Научный сервис в сети Интернет: труды XVIII Всероссийской научной конференции, (19–24 сентября 2016 г., г. Новороссийск). – М.: ИПМ им. М.В. Келдыша, 2016. – С. 115–127.
6. Davis A. What's new in Java 8: an unofficial guide [Electronic resource]. – May 7, 2014. – URL: <https://leanpub.com/whatsnewinjava8/read> (accessed: 21.03.2018).
7. Ахмечет В. Функциональное программирование для всех [Электронный ресурс] // RSDN Magazine. – 2006. – N 2. – URL: <http://rsdn.org/article/funcprog/fp.xml> (дата обращения: 21.03.2018).
8. Armstrong J. Programming Erlang: software for a concurrent world. – 2nd ed. – Dallas, Texas: The Pragmatic Bookshelf, 2013.
9. Чезарини Ф., Томпсон С. Программирование в Erlang. – М.: ДМК Пресс, 2012. – 487 с.
10. Десять решений проблемы stop the world при использовании автоматической сборки мусора [Электронный ресурс]. – URL: <https://eax.me/stop-the-world/> (дата обращения: 21.03.2018).
11. Малявко А.А. Формальные языки и компиляторы: учебное пособие для вузов. – М.: Юрайт, 2017. – 429 с.
12. Малявко А.А. Использование веб-приложений и веб-технологий при разработке учебного программного обеспечения для изучения методов трансляции // Современное образование: технические университеты в модернизации экономики России: материалы Международной научно-методической конференции. – Томск: Изд-во ТУСУР, 2011. – С. 45–47.
13. Лонес Б, Аулер Р. LLVM: инфраструктура для разработки компиляторов. – М.: ДМК Пресс, 2015. – 342 с.
14. Сен А. Создание действующего компилятора с помощью инфраструктуры LLVM. Ч. 1 [Электронный ресурс]. – URL: <https://www.ibm.com/developerworks/ru/library/os-createcompilerllvm1/index.html> (дата обращения: 21.03.2018).
15. Сен А. Создание действующего компилятора с помощью инфраструктуры LLVM. Ч. 2 [Электронный ресурс]. – URL: <https://www.ibm.com/developerworks/ru/library/os-createcompilerllvm2/index.html> (дата обращения: 21.03.2018).

*Малявко Александр Антонович*, кандидат технических наук, доцент кафедры вычислительной техники факультета автоматики и вычислительной техники Новосибирского государственного технического университета. Основные направления научных исследований: языки программирования и теория трансляции, нейронные сети, параллельные вычисления. Имеет более 40 публикаций. E-mail: [a.malyavko@corp.nstu.ru](mailto:a.malyavko@corp.nstu.ru)

DOI: 10.17212/1814-1196-2018-1-117-136

### *The El functional-imperative programming language\**

*A.A. MALYAVKO*

*Novosibirsk State Technical University, 20, K. Marx Prospekt, Novosibirsk, 630073, Russian Federation, PhD (Eng.), associate professor. E-mail: [a.malyavko@corp.nstu.ru](mailto:a.malyavko@corp.nstu.ru)*

The paper discusses the tendency of interpenetration of ideas and technologies of functional and imperative paradigms of programming in their modern implementation. A new functional-imperative programming language El is proposed, which in many respects resembles the

---

\* Received 20 August 2017.

functional Erlang language, but differs from it in some features. A brief introduction to the El lexis, syntax and semantics is given. Its main distinguishing features and characteristics are described such as pure functions, first and higher order functions, anonymous functions, closures, function signature overload, garbage collection at the moment of return from the function, imperative principle of operation execution in the body of the function, high-level data types and operations with them, variability of static and dynamic typing as well as the immutability of variables by the choice of the programmer, the availability of primitive and object types of data, brevity, simplicity and convenience of control structures, a possibility of an explicit cycle programming instead of more expensive recursive function calls, which, however, can also be used. A typical structure of a file containing a program module is described; a list and purpose of its sections, definitions of functions, and all kinds of control language operators are given. A summary of significant distinctive characteristics of the language is presented. The current state of the development and implementation of the program translator from the El language for different target platforms using the LLVM compiler infrastructure is described.

**Keywords:** Programming language, functional paradigm, imperative paradigm, lexis, syntax, semantics, expression, operator, compiler

## REFERENCES

1. Sebesta R. *Concepts of programming languages*. Boston, Addison Wesley, 2001 (Russ. ed.: Sebesta P. *Osnovnye kontseptsii yazykov programmirovaniya*. Moscow, Vil'yams Publ., 2001).
2. Kaufman V.Sh. *Yazyki programmirovaniya. Kontseptsii i printsipy* [Programming languages. Concepts and principles]. Moscow, DMK-Press Publ., 2011. 464 p.
3. Field A., Harrison P. *Functional programming*. Wokingham, England, Addison-Wesley, 1988 (Russ. ed.: Fild A., Kharrison P. *Funktsional'noe programmirovanie*. Moscow, Mir Publ., 1993. 637 p.).
4. Wadler P. Why no one uses functional languages. *ACM SIGPLAN Notices*, 1998, vol. 33 (8), pp. 23–27.
5. Gorodnyaya L.V. [Paradigmatic decomposition of the definition of a programming language]. *Nauchnyi servis v seti Internet: trudy XVIII Vserossiiskoi nauchnoi konferentsii* [Scientific service in network Internet: proceedings of the XVIII International scientific conference], Novorossiisk, September 19–24, 2016, pp. 115–127. (In Russian).
6. Davis A. *What's new in Java 8: an unofficial guide*. May 7, 2014. Available at: <https://leanpub.com/whatsnewinjava8/read/> (accessed 21.03.2018).
7. Akhmechet V. *Functional programming for the rest of us*. (In Russian). Available at: <http://www.defmacro.org/2006/06/19/fp.html> (accessed 21.03.2018).
8. Armstrong J. *Programming Erlang: software for a concurrent world*. 2nd ed. Dallas, Texas, The Pragmatic Bookshelf, 2013.
9. Cesarini F., Thompson S. *Erlang programming*. Beijing; Cambridge, MA, O'Reilly Media, 2009 (Russ. ed.: Chezarini F., Tompson S. *Programmirovanie v Erlang*. Moscow, DMK Press Publ., 2012. 487 p.).
10. *Desyat' reshenii problemy stop the world pri ispol'zovanii avtomaticheskoi sborki musora* [Ten solutions to the problem of stop the world when using automatic garbage collection]. Available at: <https://eax.me/stop-the-world/> (accessed 21.03.2018).
11. Maliavko A.A. *Formal'nye yazyki i kompilyatory* [Formal languages and compilers]. Moscow, Yurait Publ., 2017. 429 p.
12. Maliavko A.A. [Using web applications and web technologies in the development of educational software for studying methods of translation]. *Sovremennoe obrazovanie: tekhnicheskie universitety v modernizatsii ekonomiki Rossii: materialy Mezhdunarodnoi nauchno-metodicheskoi konferentsii* [Modern education: technical universities in the modernization of the economy Russia: materials International scientific-methodical conference]. Tomsk, TUSUR Publ., 2011, pp. 45–47. (In Russian).
13. Lopes B, Auler R. *Getting started with LLVM core libraries*. Birmingham, Packt Publishing, 2014 (Russ. ed.: Lopes B, Auler R. *LLVM: infrastruktura dlya razrabotki kompilyatorov*. Moscow, DMK Press Publ., 2015. 342 p.).

14. Sen A. *Create a working compiler with the LLVM framework, pt. 1.* (In Russian). Available at: <https://www.ibm.com/developerworks/library/os-createcompilerllvm1>. (accessed 21.03.2018).

15. Sen A. *Create a working compiler with the LLVM framework, pt. 2.* (In Russian). Available at: <https://www.ibm.com/developerworks/library/os-createcompilerllvm2> (accessed 21.03.2018).

Для цитирования:

Малиавко А.А. Функционально-императивный язык программирования EI // Научный вестник НГТУ. – 2018. – № 1 (70). – С. 117–136. – doi: 10.17212/1814-1196-2018-1-117-136.

For citation:

Maliavko A.A. Funktsional'no-imperativnyi yazyk programmirovaniya EI [The EI functional-imperative programming language]. *Nauchnyi vestnik Novosibirskogo gosudarstvennogo tekhnicheskogo universiteta – Science bulletin of the Novosibirsk state technical university*, 2018, no. 1 (70), pp. 117–136. doi: 10.17212/1814-1196-2018-1-117-136.