костыль 1

костыль 2

костыль 3

костыль 4

АННОТАЦИЯ

Работа посвящена разработке функционального языка программирования с динамической типизацией и ML-подобным синтаксисом и компилятора этого языка.

Проведен обзор ряда существующих языков программирования. Разработан синтаксис функционального языка программирования. Реализован компилятор из этого языка в язык программирования Scheme.

В организационно-экономической части представлено технико-экономическое обоснование разработки, проанализирована структура затрат проекта и построен календарный план-график проекта.

Пояснительная записка к работе содержит текст на 90 листах формата А4, 9 таблиц, 8 рисунков, список литературы из 14 библиографических ссылок.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Ранее, язык программирования был воплощением какой-либо парадигмы, концепции. Сейчас — при разработке языка стремятся добиться удобства. Особое внимание уделяется краткости, удобству написания и чтения кода.

К настоящему времени разработано несколько тысяч языков программирования, из которых около двадцати являются широко распространёнными [1]. У всех есть свои сильные стороны и под конкретные задачи выбирается конкретный язык программирования, наиболее подходящий для этих целей. Так, например, для написания кросс-платформенных приложений используются Java или C++ [2]. Для браузерных расширений и сайтов — JavaScript.

Однако, для большинства языков, выигрыш от их выбора не так очевиден. Пользователи языка Python считают код на своём языке простым для понимания, потому что чтение программы на Python напоминает чтение текста на английском языке. Это позволяет сосредоточится на решении задачи, а не на самом языке. Lisp является программируемым языком программирования, что позволяет изменять и дополнять его под конкретные задачи. Язык OCaml благодаря системе вывода типов позволяет писать высокоэффективные и безопасные приложения.

Анализ текущего состояния в разработке языков программирования показал, что существует необходимость в языке программирования, ориентированном на быструю разработку сценариев (скриптов), первоначальном обучении программированию и исследовательском программировании (когда изначально неизвестно как программа должна работать).

Для этих целей, на наш взгляд, должен существовать функциональный язык с «дружелюбным» синтаксисом, который подразумевает инфиксную нотацию в записи арифметических выражений.

Целью данной работы является разработка функционального языка программирования с динамической типизацией и ML-подобным синтаксисом и компилятора этого языка.

1 Языки-прототипы

1.1 Семейство языков Lisp, язык Scheme

Lisp (LISt Processing language) — функциональный язык программирования с динамической типизацией. Он был создан для символьной обработки данных [2]. *Scheme* — диалект Lisp'a, использующий хвостовую рекурсию и статические области видимости переменных [3]. Scheme — высокоуровевый язык общего назначения, поддерживающие операции над такими структурами данных как строки, списки и векторы.

Язык Scheme был создан как учебный, однако, в последнее время используется для написания текстовых редакторов, оптимизирующих компиляторов, операционных систем, графических пакетов и экспертных систем, вычислительных приложений и систем виртуальной реальности. Язык оказал сильное влияние на многие современные языки программирования, такие как Haskell (и языки семейства ML в целом), Rust, Python, JavaScript.

В Scheme реализована *сборка мусора* — одна из форм управления памятью, удаляющая оттуда объекты, которые уже не будут востребованы.

Все объекты, в том числе функции, являются *объектами первого класса*, что позволяет присваивать функции переменным, возвращать функции и принимать их в качестве аргументов. Все переменные и ключевые слова объедены в области видимости, а программы на языке имеют блочную структуру.

Как и во многих других языках программирования процедуры на языке Scheme могут быть рекурсивными. Все хвостовые рекурсии оптимизируются.

Scheme поддерживает определение произвольных структур с помощью *продолжений*. Продолжения сохраняют текущее состояние программы и позволяют продолжить выполнение с этого момента из любой точки программы. Этот механизм удобен для перебора с возвратом, многопоточности и сопрограмм [4].

Scheme также позволяет создавать синтаксические расширения для написания процедур трансформации новых синтаксических форм в существу-

ющие [4].

Функции на языке Scheme могут иметь произвольное число аргументов. Тем переменным, наличие которых необходимо, присваиваются имена, а остальные можно получить из списка оставшихся.

В Scheme есть *статические переменные* — долговременные переменные, существующие на протяжении функции. Они отличаются от глобальных переменных тем, что существуют только внутри функции, могут хранить свои значения между вызовами, но при этом не доступы извне. Это позволяет организовывать *мемоизацию* (сохранение результатов выполнения функций для предотвращения повторных вычислений). Это является одним из способов оптимизации. При использовании оптимизации перед началом вычислений проверяется вызывалась ли ранее эта функция с этим набором аргументов. Если не вызывалась, то результат вычисляется, сохраняется в статической переменной и возвращается. Если функция уже вызывалась с данным набором аргументов, то возвращается сохранённый ранее результат.

Язык Scheme обладает полноценными средствами символьной обработки.

Как и в большинстве языков семейства Lisp, в языке Scheme как код, так и данные представлены в виде упорядоченных последовательностей значений — списков. Таким образом, способ представления кода не отличается от способа представления данных. Это позволяет работать с кодом как с данными, в том числе разрабатывать самомодифицирующиеся программы.

В Scheme используется префиксная нотация. Выражения являются списками с оператором во главе этого списка. Такой способ записи позволяет, например, сложить сразу несколько значений. Но выражения с несколькими операторами трудны для восприятия из-за большого количества скобок. Это требует от разработчика использовать специализированные текстовые редакторы, поддерживающие контроль парности скобок и принятый стиль форматирования кода. Таких редакторов немного, и они либо недружелюбны к начинающему программисту (Emacs, vim), либо ориентированы на определённый диалект языка (Racket).

Пример функции вычисления факториала числа, написанный на языке Scheme:

```
(define (factorial n)
  (if (zero? n)
      1
      (* n (factorial (- n 1)))))
```

1.2 Язык Prolog

Prolog (PROgramming in LOGic) — язык, объединяющий в себе логическое и алгоритмическое программирование. Этот язык специально разрабатывался для систем обработки естественных языков, исследований искуственного интеллекта и экспертных систем. В настоящее время этот язык применяется не очень широко [1], хотя современный Prolog во многом является языком программирования общего назначения.

Язык использует *сопоставление с образцом* (в том числе повторное использование). Сопоставление с образцом в языке Prolog базируется на унификации, в частности используя алгоритм Мартелли-Монтэнери [6]. Этот алгоритм сопоставляет значения двух элементов выражения, находящихся на одинаковой позиции и в случае успеха приравнивает значения на других позициях.

Ключевыми особенностями языка Prolog являются унификация и перебор с возвратом. Унификация показывает как две произвольные структуры могут быть равными. Процессоры Prolog'а используют стратегию поиска, которая пытается найти решение проблемы перебором с возвратом по всем возможным путям, пока один из них не приведёт к решению [5]. Программа на языке Prolog представляет собой набор фактов и правил.

Входной точкой в программу на языке Prolog является запрос. Ответом на него может быть либо список подошедших шаблонов, либо `false`, означающий, что совпадений не найдено.

Рассмотрим пример программы на языке Prolog.

```
street( saint-petersburg, nevskii ).
street( moscow, arbat ).
street( berlin, arbat ).
```

Эта программа состоит из трёх предложений, каждое из которых объявляет один факт об отношении `street`. Например, факт `street( moscow, arbat )` описывает, что улица `arbat` относится к `moscow`. После передачи соответствующей программы в систему Prolog последней можно задать некоторые вопросы об отношении `street`. Например, можно узнать все города, в которых есть улица `arbat`. Сделать это можно введя в терминал следующий запрос:

```
?- street( X, arbat ).
```

После этого Prolog начнёт отыскивать все пары город-улица, где улица — `arbat`. Решения отображаются на дисплее по одному до тех пор, пока Prolog получает указание найти следующее решение (в виде точки с запятой) или пока не будут найдены все решения [7]. Ответы выводятся следующим образом:

```
X = moscow;
X = berlin;
true.
```

1.3 Семейство языков ML, Язык OCaml

ML (Meta Language) — семейство языков с полиморфным выводом типов и обработкой исключений. Языки ML не являются чистыми функциональными языками.

OCaml — самый распространённый в практической работе диалект ML.

Он включает в себя сборку мусора для автоматического управления памятью. Как и в Lisp, функции являются объектами первого класса. Это значит, что функции можно передавать в качестве аргумента, возвращать в качестве значения и присваивать их переменным.

OCaml использует статическую проверкую типов, что позволяет увеличить скорость сократить количество ошибок во время исполнения. При этом поддерживается *параметрический полиморфизм* — свойство семантики системы типов, позволяющее обрабатывать данные разных типов идентичным образом. Это даёт возможность создавать абстракции и работать с разными типами данных.

Механизм автоматического вывода типов позволяет избегать тщательного определения типа каждой переменной, вычисляя его по тому как она используется. Кроме того, в OCaml, как и в Scheme, есть поддержка неизменяемых данных.

В языке OCaml, поддерживаются *алгебраические типы данных* (составные типы, имеющие набор конструкторов, каждый из которых принимает на вход значения определённых типов и возвращает значение конструируемого типа) и сопоставление с образцом, чтобы определять и управлять сложными структурами данных [9].

Пример вычисления факториала числа на языке OCaml, используя сопоставление с образцом:

```
let rec factorial = function
    | 0 -> 1
    | n -> n * factorial (n - 1)
```

1.4 Язык Haskell

Haskell — чистый функциональный язык программирования общего назначения. *Чистота* означает, что результат вычислений, производимых функциями не зависит от состояния программы. Он зависит только от набора входных параметров, а значит, уже вычисленные значения не изменятся. А значит, при следующем вызове функции с таким же набором фактических аргументов это значение можно уже не вычислять.

Кроме того, Haskell является ленивым, что позволяет не вычислять значения, пока это не нужно. *Ленивые вычисления* ускоряют работу программы, однако позволяют допускать ошибки, так как выражение, содержащее ошиб-

ку может быть невыполнено.

Haskell обладает статической сильной полной типизацией с атоматическим выводом типов.

Также, Haskell поддерживает функции высшего порядка и частичное применение (*каррирование* — преобразование функции от многих аргументов в набор функций, каждая из которых принимает один аргумент).

Haskell ограниченно используется для написания сценариев. Дело в том, что Haskell удлиняет код для коротких скриптов за счёт статической типизации и необходимости использования монад, для организации императивного кода и функций с побочными эффектами. Это приводит также к усложнению кода, к введению лишних сущностей. Монады и необходимость следить за типами означают, что у языка высокий порог вхождения, что делает его неподходящим для первоначального обучения. Статическая проверка типов сокращает время выполнения программы, однако увеличивает затраты времени на проверку типов при загрузке скрипта интерпретатору. Это также делает Haskell менее удобным для написания скриптов, по скравнению с языками с динамической типизацией.

Haskell — краткий и элегантный язык [10]. В нём используется минимум ключевых слов. Например, для определения функции во многих языках необходимо использование ключевых слов (например, `define` в Scheme или `function` в OCaml). Нотация в целом похожа на математическую. Также, как в Python вместо скобок, ограничивающих блоки кода (Scheme, C, JavaScript) и разделительных знаков (C, OCaml) в нём используются отступы. Отступы в Haskell являются *синтаксическим сахаром* (синтаксической возможностью, не влияющей на выполнение программы, но упрощающей написание кода). Фактически препроцессор компилятора заменяет их на ограничивающий блок (`{}`) и разделители (`;`). Это делает код легче для восприятия, так как сразу видна вложенность. Однако, в отличие от Python, где отступы обязаны быть одинаковыми, в Haskell важно лишь, чтобы отступ вложенной операции был больше родительской.

Пример вычисления факториала числа на языке Haskell:

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

1.5 Концепция языка

Язык разрабатывается для написания скриптов, первоначального обучения программированию и исследовательского программирования.

Для этих целей больше подходит динамическая типизация, как в Scheme для большей гибкости и краткости кода.

Чтобы снизить порог вхождения Язык должен быть содержать минимум ключевых слов и разделяющих знаков, как в Haskell.

Для лучшего понимания программы запись функций и результат их вычислений должен быть приближен к математической. Для работы с математическими формулами необходимы символьные вычисления (Scheme, Prolog).

Короткие программы легче поддерживать. И, как правило, в них меньше возможности допустить ошибку. Для краткости кода в нём должно быть реализовано сопоставление с образцом как в OCaml и повторное использование переменных как в Prolog.

Для оптимизации рекурсивных функций (таких как функция вычисления факториала и чисел фибоначчи) необходима возможность мемоизации функций (как в языке Haskell).

Функция вычисления факториала числа на Языке должна выглядеть следующим образом:

```
n! 0 <- 1
n! n <- n * n! (n - 1)
```

1.6 Выбор языка реализации и целевого языка для компилятора

Для реализации динамической типизации удобнее всего выбрать язык с таким типом типизации. Таковым является язык Scheme.

Символьные вычисления также реализованы в Scheme.

Таким образом, язык Scheme был выбран и для реализации компилятора, и как язык, в который будет осуществляться компиляция. Для реализации была выбрана спецификация языка R5RS, так как она поддерживается всеми известными компиляторами (GNU Guile, Chicken Scheme, Racket).

2 Разработка языка

2.1 Критерии эффективности языка

Эффективность определяет степень соответствия языка программирования своему назначению. Она измеряется либо количеством затрат, необходимых для получения определённого результата, либо результатом, полученным при определённых затратах. Определить эффективность конкретного языка позволяют *критерии эффективности* (правила, служащие для сравнительной оценки качества различных языков программирования) [11].

Основные критерии эффективности для разрабатываемого языка:

- читабельность (легкость чтения и понятность программы);
- лёгкость написания программ;
- надёжность (обработка исключительных ситуаций);
- стоимость (суммарная стоимость всего жизненного цикла языка);
- переносимость (легкость переноса программ на языке из одной операционной системы в другую).

2.1.1 Читабельность

Одним из важнейших критериев качества языка является легкость чтения и понимания программ, написанных на нём. По современным представленнием, самым длительным этапом времени жизни программы является её сопровождение, в ходе которого неё вносятся какие-то изменения. Так как читабельность программы влияет на лёгкость сопровождения, то её считают существенной характеристикой качества программ и языков программирования [12].

На читабельность кода оказывают влияние следующие факторы:

- простота;
- ортогональность;
- структурированность потока управления в программе.

Язык должен предоставить простой набор конструкций, которые могут

быть использованы в качестве базисных элементов при создании программы. Желательно обеспечить минимальное количество различных понятий с простыми правилами их комбинирования. Этому мешает наличие в языке нескольких способов описания одного действия. На простоту влияет синтаксис языка. Он должен максимально прозрачно отражать семантику языка и исключать двусмысленность толкования [12].

Ортогональность означает, что любые возможные комбинации различных языковых конструкций будут осмысленными без непредвиденных ограничений или неожиданного поведения, возникающих в результате взаимодействия конструкций или контекста использования [12]. Например, язык содержит следующие элементарные типы данных: число, строка. Также в языке присутствуют конструкции: список и функция. Возможность применения этих конструкций к самим себе говорят об их ортогональности, обеспечивающей большое количество структур данных.

Когда конструкции языка ортогональны, язык легче выучить и использовать для чтения и написания программ, так как в нём меньше исключительных и специальных случаев, которые необходимо запомнить. Однако, излишняя ортогональность может стать источником проблем. Например, в ходе компиляции не генерируются ошибки, даже при наличии комбинаций, которые лишены логического смысла или крайне неэффективны при исполнении [12]. Таким образом, простота языка достигается комбинированием небольшого числа элементарных конструкций языка и ограниченного применения ортогональности.

Структурированность потока управления в программе означает, что порядок передачи управления между операторами программы должен быть удобен для чтения и понимания программы.

2.1.2 Лёгкость написания программ

Лёгкость написания программ, отражает удобство использования языка для написания программ в конкретной предметной области. Характеристики лёгкости написания программ во многом пересекаются с характеристи-

ками читабельности.

Концептуальная целостность включает в себя следующие взаимосвязанные компоненты:

- простота;
- ортогональность;
- единообразие понятий.

Простота предполагает использование минимального числа понятий. *Ортогональность* позволяет комбинировать любые языковые конструкции. *Единообразие понятий* требует единого подхода к описанию и использованию всех понятий [12].

Простота языка достигается за счёт уменьшения количества случайных ограничений на использование сложных языковых конструкций. Например, при реализации массивов следует разрешить их создание для любого типа данных. Какие-либо ограничения будут усложнением языка. Это также относится и к типам аргументов функций и возвращаемых значений. В частности, функции должны уметь принимать и возвращать функции.

Простота уменьшает затраты на обучение программистов и вероятность ошибок, возникающих из-за неправильной интерпретации программистом языковых конструкций [12]. Если язык содержит слишком большой набор конструкций, то программист может не знать каждую из них. Это приводит к неправильному или неэффективному использованию одних и игнорированию других конструкций. Можно сделать вывод о преимуществе использования небольшого набора элементарных конструкций и их комбинаций, над применением большого числа конструкций.

Выразительность языка характеризует следущее:

- наличие мощных средств для создания структур и описания действий, позволяющих описать большой объём вычислений в виде относительно маленькой программы;
- возможность записи вычислений в компактной и удобной форме.

2.1.3 Надёжность

Надёжность — способность программы выполнять требуемые функции при заданных условиях и в течение определённого периода времени [12]. Как правило уровень надёжности зависит от степени автоматического обнаружения ошибок. Надёжный язык позволяет выявить большое количество ошибок во время компиляции программы, а не во время выполнения, так как это минимизирует стоимость ошибок.

Высокую надёжность обеспечивает проверка типов. Она характерна для языков со статической типизацией. Языки с динамической типизацией осуществляют эту проверку только во время выполнения программы. Тем не менее, динамическая типизация часто используется в скриптовых языках для большей гибкости и сокращения объема кода (в данном случае — за счёт отсутствия объявлений типов переменных и функций).

Важным критерием надёжности является качество механизма *обработки исключений* — большинство ошибок должно быть найдено и, по возможности, устранено для последующей работы программы.

Читабельность и лёгкость написания программ на языке увеличивают надёжность программ, написанных на нём [12].

2.1.4 Стоимость

Стоимость складывается из нескольких составляющих:

- стоимость разработки, тестирования и использования программы;
- стоимость компиляции программы;
- стоимость выполнения программы;
- стоимость сопровождения программы.

Стоимость разработки, тестирования, использования программы во многом зависит от читабельности и лёгкости написания программ на языке.

Стоимость компиляции зависит от количества проверок, осуществляемых во время компиляции. К ним относятся проверки типов и обработки

лексических, синтаксических и семантических ошибок.

Стоимость выполнения во многом зависит от структуры языка и от того, какой объём проверок был оставлен компилятором для осуществления во время выполнения. Таким образом, чем выше стоимость компиляции, тем ниже скорость выполнения.

Исследования показали[12], что значительная часть стоимости программы приходится на стоимость сопровождения. Сопровождение включает в себя:

- обработку ошибок (17% времени и стоимости);
- изменения, связанные с обновлением операционного окружения (18% времени и стоимости);
- усовершенствование и расширение функций программы (65% времени и стоимости).

Сопровождение программ зависит от читабельности, так как часто сопровождение осуществляется не авторами программы, а другими лицами [12].

2.1.5 Переносимость

Под *переносимостью* (*портированием*) подразумевается возможность, скомпилировав код один раз запускать его на различных платформах. В таком случае переносимость языка напрямую зависит от переносимости целевого языка компилятора.

Использование языка Scheme в качестве целевого обеспечит достаточно высокую портируемость программ, так как R5RS поддерживаются многими (GNU Guile, Chicken Scheme) компиляторами и интерпретаторами в средах наиболее распространённых ОС.

2.2 Описание языка

Разрабатываемый язык предназначен для написания коротких программ, скриптов, а также для первоначального обучения программированию. Поэтому сам язык должен быть максимально краток и понятен для разра-

ботчика. В языке должны отсутствовать лишние ключевые слова и управляющие конструкции. Для математических формул обязательна инфиксная нотация. Для простоты языка запись функций на нём должна быть близка к их записи в математике.

Для повышения ортогональности языка, основные его конструкции должны быть примениммы к себе и друг другу.

Элементарными конструкциями языка являются числа, идентификаторы и строки. Основные составные конструкции языка — списки и функции. Выражение на языке является комбинацией операторов и основных конструкций.

Функции в языке похожи на математические. Они могут иметь аргументы и должны возвращать значение. При этом функции не меняют значение своих аргументов, то есть можно говорить об *иммутабельности* аргументов. Функции могут являться функциями высшего порядка, то есть аргументы и возвращаемые значения могут быть функциями. Это обеспечивает высокую ортогональность и простоту языка, а также читабельность программ, написанных на нём.

Для повышения читабельности и простоты языка было решено добавить в него сопоставление с образцом. Это позволяет использовать числа и списки, а не только идентификаторы, для определения аргументов функций. При этом, в определении функции аргументы не могут являться сложными выражениями: в списке аргументов в определении функции не должно быть вызовов функций и применения операторов. Это ограничение имеет смысл, так как его отсутствие противоречило бы смыслу определения функции.

Сопоставление с образцом осуществляется сверху вниз, в порядке записи образцов и соответствующих им вариантов функций. При успешном сопоставлении остальные образцы не проверяются. Это означает, что писать шаблоны нужно от частных к общим.

Наиболее общим шаблоном для для одного аргумента является идентификатор. Встретив идентификатор, компилятор проверит только наличие этого аргумента при вызове функции, но не будет проверять его значение.

Наиболее общим образцом для списка аргументов является конструк-

ция

`: ident`,

где `ident` — простой идентификатор. При таком обозначении не имеет значение даже количество аргументов (их может не быть вовсе). Этот шаблон подойдёт для вызова функции с любым количеством аргументов. В теле функции эти аргументы будут доступны в виде списка, обратиться к которому можно будет по заданному идентификатору (в данном случае `ident`).

Конструкция, описанная выше (`: ident`), в разрабатываемом языке отвечает за *хвост списка*. Она почти соответствует таковой в Haskell. Основным отличием этой конструкции от соответствующей в Haskell является то, что для явного указания нескольких элементов списка в Haskell их все нужно указывать через `:`. В разрабатываемом языке двоеточие ставится только перед «хвостом» списка, остальные же элементы списка отделены пробелами. Подобная конструкция есть и в языке Scheme. В нём она выглядит следующим образом:

`. ident` Точка в языке Scheme служит для разделения элементов в *точечной паре*. В основном точечная пара используется для создания списков. В примере ниже две точечные пары, каждая из которых является списком `(1 2 3)`:

```
(1 . (2 3))
(1 . (2 . (3)))
```

Однако, точечная пара в языке Scheme не всегда является списком. Примеры таких пар:

```
(1 . (2 . 3))
(1 2 . 3)
```

В этом примере, видно, что элемент после точки не является списком, поэтому и вся структура, описанная в последнем примере не является списком.

Чтобы не возникало ассоциаций с точечной парой в Scheme в разрабатываемом языке «хвост» списка решено было отделять знаком `:`, как в языке Haskell. При этом двоеточие, решено было оставить только перед «хвостом», чтобы закрепить понимание того, что после двоеточия всегда должен быть список.

Таким образом, конструкция с двоеточием может быть записана не только в виде «двоеточие-идентификатор», но и в виде «двоеточие-список», где список, в свою очередь также может быть записан в виде «головы» и «хвоста списка».

Примеры данной конструкции:

```
; нахождение суммы всех элементов списка
sum-list []       <- 0
sum-list [x : xs] <- x + sum-list xs

; получение первого элемента списка
head [x : xs] <- x

; получение второго элемента списка
second [x y : zs] <- y

; получение хвоста списка
tail [x : xs] <- xs
```

Объявление функций состоит из имени функции, её аргументов и тела функции. Аргументами функции могут быть имена переменных, числа, списки и «хвосты» списка аргументов. Объявлений у функции может быть несколько. При вызове функции её аргументы сопоставляются с представленными образцами и возвращается значение первого подошедшего варианта. Если в списке аргументов одно имя используется несколько раз, то эти аргументы при проверке будут считаться равными. Особо отметим, что аргументы с одинаковыми именами — частный случай аргументов. Рассмотрим пример подсчёта количества вхождений элемента `x` в список:

```
count x []         <- 0
count x [ x : xs ] <- 1 + count x xs
count x [ y : xs ] <- count x xs
```

В первой строке мы видим определение функции `count` от `x` и пустого списка. В этом случае количество вхождений `x` в список равно нулю. Во второй строке мы видим определение функции `count` от `x` и списка, первым элементом которого является `x`. В данном случае нам нужно прибавить еди-

ницу к количеству вхождений `x` в «хвост» списка `xs`. И, наконец, в третьей строке определена функция `count` от `x` и списка, первым элементом которого является `y`. Так как в строке выше мы предусмотрели равенство `x` и первого элемента списка, то в этой строке определения мы можем быть уверены, что `x` не равен `y`. Однако, если мы поменяем вторую и третью строки местами, то до третьей строки вычисление никогда не дойдёт, потому что два значения аргументов с идентификаторами `x` и `y` не будут подвергаться сравнению.

В разрабатываемом языке присутствует два вида стрелок. Как видно из примеров выше, стрелка влево (`<-`) используется для связывания имени функции с телом. Стрелка влево указывает на то, что тому, что находится слева ставится в соответствие то, что находится справа.

Стрелка вправо (`->`) означает, что из левой части следует правая. В математике стрелкой вправо обозначается импликация. Она применяется, например, в условном операторе `if`. В случае выполнения одного из условий при переходе по стрелке будет вычислено выражение, следующее за ней.

Другой случай применения этой стрелки — безымянные функции. В математике стрелка вправо также отвечает за отображение области определения на область значений. Безымянные (lambda) функции фактически являются преобразованием своих аргументов в итоговое значение. В функциональных языках программирования функции являются объектами первого класса, то есть не отличаются от других типов. Анонимные функции фактически являются значением типа «функция», а значит, такой функции не может быть присвоено значение. Поэтому после аргументов `lambda`-функции следует стрелка вправо, за которой следует нужное выражение.

Для определения вложенных конструкций, например, функций внутри функции, должны использоваться отступы. Отступ вложенной конструкции должен быть больше, чем отступ внешней. Таким образом, проверяется только наличие отступа, но ограничений на его размер не налагается. Это может использоваться для улучшения восприятия длинных конструкций, «табличного» форматирования кода.

В примере ниже мы видим определение функции `filter`. `Filter` — функция высшего порядка (то есть функция, которая принимает функции в

качестве аргументов). Традиционно эта функция входит в стандартную библиотеку функциональных языков программирования в числе функций высшего порядка для обработки списков (`map`, `reduce`, `fold`). Она принимает два аргумента. Первый (`pred?`) — предикат — функция одного аргумента, возвращающая логическое значение. Второй — список. Функция `filter` должна вернуть список всех значений исходного списка, для которых выполнено условие `pred?`.

Как видно в первой строке, в случае, если второй аргумент — пустой список, то мы получим пустой список. Во второй строке в качестве второго аргумента мы видим список, состоящий из как минимум одного элемента `x` и «продолжения» `xs`. По отступу в третьей строке мы понимаем, что условная конструкция `if` относится ко второй строчке определения функции. В случае, если выполняется условие `pred? x` мы возвращаем конкатенацию списка из одного элемента `x` и результат рекурсивного вызова функции `filter` от того же предиката и окончания списка `xs`. Если условие выполнено не было, то вычисление продолжится с четвёртой строки, где условие отсутствует и будет осуществлён вызов функции `filter` от `pred?` и `xs`.

```
filter pred? [] <- []
filter pred? [x : xs] <-
    if  | pred? x -> [x] ++ filter pred? xs
        |         -> filter pred? xs
```

Для расширения возможностей Языка предусмотрена вставка кода на языке Scheme. Для этого пишется ключевое слово `scheme` за которым должна следовать одна конструкция на этом языке. Чтобы использовать функции, определённые таким образом их надо экспортировать, используя ключевое слово `export`, после которого должны следовать имена всех функций, которые будут использованы в дальнейшем.

2.3 Синтаксис функционального языка программирования

Программа на Языке представляет собой набор выражений, определений функций и их вызовов.

В отличие от спецификации языка Scheme R5RS, разрабатываемый язык является регистрозависимым. В более поздних спецификациях языка Scheme, например, R7RS, язык является регистрозависимым. В R5RS, например, `item`, `Item` и `ITEM` — три различных идентификатора. Это приучет начинающего программиста аккуратней относится к обозначениям в коде. Кроме того, такой код понятней. Когда в одном месте функция названа `item`, ниже вызывается словом `Item` — не сразу становится понятно о чём идёт речь.

Далее представлен синтаксис Языка в расширенной форме Бэкуса-Наура (РБНФ)[13].

2.3.1 Токены

Существуют следующие виды токенов: идентификатор, число, строка, операторы и ключевые слова. Пробельные символы игнорируются, если они не существенны для разделения двух последовательных токенов.

- Число — целочисленная, вещественная константа, дробь или комплексное число. Число (в том числе дробь и вещественное число) может быть записано в шестнадцатеричной системе счисления. Запись числа в Языке соответствует принятой в языке программирования Scheme.

  Примеры чисел: `#xA9.F`, `4/7`, `2+7i`, `#x7/ad`.

```
Digit    ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.
Hexd     ::= 'A'|'B'|'C'|'D'|'E'|'F'|'a'|'b'|'c'|'d'|'e'|'f'.
Hdigit   ::= Hexd|Digit
Dec      ::= ('+'|'-')? Digit+.
Hex      ::= '#x' Hdigit+.
Int      ::= Dec|Hex.
Fracdec  ::= Dec '/' Digit+.
Frachex  ::= Hex '/' Hdigit+.
Frac     ::= Fracdec|Frachex.
```

```
Expint  ::= Dec 'e' Dec.
Realdec ::= Dec '.' (Digit+ ('e' Dec)? )?.
Realhex ::= Hex '.' Hdigit+
Real    ::= Realdec|Realhex.
Num     ::= Int|Real|Frac.
Complex ::= Num? ('+'|'-') Num 'i'.
Number  ::= Num|Complex.
```

- Идентификатор — последовательность символов, исключающая пробельные символы, точки, скобки, кавычки, двоеточие. В отличие от большинства языков идентификаторы могут начинаться с цифр. В этом случае идентификатор целиком не должен являться числом.

  Примеры идентификаторов: `day-of-week`, `0->1`, `nil?`, `%2!0?`.

```
Ident       ::= BinaryFunc|UnaryFunc|SimpleIdent.
SimpleIdent ::= (?! [\(\)\[\]\{\}\.\:\"]|Number|
                BinaryFunc|UnaryFunc).
BinaryFunc  ::= 'mod'|'div'|'eq?'|'eqv?'|'equal?'|'gcd'|
                'lcm'|'expt'|'map'|'filter'.
UnaryFunc   ::= 'zero?'|'null?'|'abs'|'odd?'|'even?'|
                'round'|'reverse'|'not'|'sin'|'cos'|
                'tg'|'ctg'|'sqrt'.
```

- Строки — последовательности символов, заключённые в двойные(") кавычки.

  Примеры строк: `"valid string"`, `"it's a beautiful day"`.

- Булевые константы — константы, представляющие логические значения «истина» и «ложь».

  Истина: `#t`. Ложь: `#f`.

```
Bool ::= '#t'|'#f'.
```

- Операторы и ключевые слова — специальные символы, пары символов и слова, зарезервированные компилятором Языка и нижлежащим компилятором Scheme.

  Зарезервированные ключевые слова, символы и сочетания символов представлены в таблице 1.

Таблица 1 — Ключевые слова и зарезервированные символы

Слово, символ	**Описание**
`scheme`	позволяет использовать код на языке Scheme
`export`	позволяет использовать функции, определённые в коде на Scheme с помощью ключевого слова scheme
`memo`	обеспечивает мемоизацию функций
`print`	печатает значения
`sin`, `cos`, `tg`, `ctg`	тригонометрические функции, вычисляющие синус, косинус, тангенс и котангенс аргумента
`mod`	находит остаток от деления числа `x` на число `y`
`div`	находит результат целочисленного деления числа `x` на число `y`
`abs`	находит модуль числа
`even?`/ `odd?`	находит результат проверки числа на чётность/нечётность
`gcd`	находит наибольший общий делитель двух чисел
`lcm`	находит наименьшее общее кратное двух чисел
`round`	округляет число до ближайшего целого
`expt`	возводит первый аргумент в степень, равную второму аргументу
`sqrt`	находит корень из числа
`not`	возвращает логическое значение, противоположное значению аргумента
`eq?`, `eqv?`, `equal?`	проверки на равенство. `eq?`, `eqv?` проверяют равенство адресов, на которые ссылаются аргументы. `equal` сравнивает значения
`zero?`	проверяет, является ли аргумент числом 0
`reverse`	переворачивает список
`head`	возвращает «голову» списка

Таблица 1 — Ключевые слова и зарезервированные символы

Слово, символ	**Описание**
`tail`	возвращает «хвост» списка
`.`	`f.x` – применение функции `f` к списку аргументов `x`
`:`	для записи «продолжений» списков `[x : xs]`
`null?`	проверяет, является ли аргумент пустым списком
`car`, `cdr`, `caar`, `cddr`, `cadr`, `cdar`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`	функции для получения элементов списков
`eval`	зарезервировано компилятором языка Scheme
`if`	условный оператор
`|`	маркер начала условного оператора в конструкции if
`!`, `&&`, `||`, `^`	логические операторы: отрицание, и, или, исключающее или
`<`, `<=`, `>`, `>=`, `=`, `!=`	операторы сравнения
`( ) [ ] { }`	различные виды скобок для математических выражений, списков и областей определения
`\`	определение анонимной функции `\ x -> x + 1`
`<-`	оператор связывания переменной / функции со значением
`->`	оператор имликации и отношения
`-`, `+`, `++`, `/`, `//`, `%`, `*`, `**`	математические операторы (вычитание, сложение, конкатенация, деление, целочисленное деление, остаток от деления, умножение, возведение в степень)

2.3.2 Объявление функций

Функции делятся на именованные и безымянные λ-функции. Объявление именованных функций (FunctionDefinition) состоит из имени функции, её аргументов, знака связывания <- и тела функции. Аргументами функции при объявлении могут быть:

- идентификаторы, не являющиеся ключевыми словами (SimpleIdent);
- числа (Number);
- списки, не содержащие выражений (ListDeclaration);
- и «хвост» списка аргументов (ContinuousDeclaration) — этот элемент может быть в списке аргументов только последним и он заменяет все аргументы, которые могут быть переданы в эту функцию после уже объявленных.

```
FunctionDefinition     ::= FunctionDeclaration '<-' FunctionBody.
FunctionDeclaration    ::= SimpleIdent ArgumentDeclaration?.
ArgumentDeclaration    ::= SimpleArgument* ContinuousDeclaration?
SimpleArgument         ::= SimpleIdent|Number|ListDeclaration.
ListDeclaration        ::= '['ArgumentDeclaration?']'.
ContinuousDeclaration ::= ':' SimpleIdent|ListDeclaration.
```

Пример объявления функции:

```
inc x <- x + 1
```

Объявление безымянных функций (LambdaFunction) похоже на объявление именных. Вместо идентификатора, отвечающего за имя функции, используется символ \.

```
LambdaFunction ::= '\' ArgumentDeclaration? '->' FunctionBody.
```

Функции Языка являются объектами первого класса. Это демонстрирует частный случай определения функции, где её телом является lambda-функция.

2.3.3 Мемоизация функций

Мемоизация — это сохранение результатов выполнения функции, с целью предотвращения повторных вычислений.

Если предполагается многократное использование функций с большим объёмом вычилений (таких как вычисление факториала числа, вычисление чисел фибоначчи, трибоначчи), то рекомендуется их мемоизировать.

Чтобы укзать компилятору какие функции необходимо мемоизировать, нужно перечислить их после ключевого слова `memo`.

```
Memoize ::= 'memo' SimpleIdent+.
```

2.3.4 Тело функции

Тело функции, также может содержать в себе определения функций. Функции, определённые таким образом также можно мемоизировать. Тело функции обязательно содержит хотя бы одно выражение. Выражений может быть несколько, но вернёт функция только последнее выражение.

Все выражения и определения внутри тела функции должны иметь больший отступ, чем имя этой функции.

При написании тела функции рекомендуется записывать длинные конструкции в несколько строк, с соблюдением отступов. Нет строгой проверки на величину отступов, поэтому следить за тем, что отстутпы одинаковы программисту придётся самостоятельно.

```
FunctionBody ::= FunctionDefinition* Memoize* Expression+.
```

2.3.5 Выражение

Выражение (Expression) может быть

- условным выражением (IfExpression);
- безымянной функцией (LambdaFunction);
- выражением в инфиксной нотации (InfixExpression).

```
Expression ::= IfExpression | LambdaFunction | InfixExpression .
```

2.3.6 Условный оператор

Условный оператор `if` похож на оператор `cond` в Scheme. Оператор имеет вид:

а) после вертикальной черты (`|`) следует условие. В случае если условие отсутствует, считается, что условие истинно. Таким образом, опустив условие можно организовать ветку `else` для привычных условных операторов или `default` для `switch-case` оператора;

б) после импликации (`->`) следует выражение, которое выполнится, если условие перед стрелочкой было выполнено. Если условие было выполнено, то после вычисления выражения действие оператора заканчивается и остальные условия не проверяются. Если условие выполнено не было — переходим к проверке следующего условия.

```
IfExpression ::= 'if' ('|' Expression? '->' Expression)+.
```

Пример использования конструкции `if`:

```
sign-if x <-
    if  | x > 0 -> 1
        | x = 0 -> 0
        |       -> -1
```

2.3.7 Выражение в инфиксной нотации

Выражение в инфиксной нотации может быть

- строкой или конкатинацией строк (StringExpression);
- вызовом функции (FunctionCall) или применением функции к списку аргументов (FunctionApply);
- списком или конкатинацией списков (ListExpression). Элементы списка при этом не могут содержать вызов функции. Поэтому, если это необходимо, то нужно использовать применение функции к списку аргументов.
- применением операторов к выражениям;
- объединением выражений с помощью операторов и скобок.

```
InfixExpression    ::=StringExpression|ListExpression|Number|Bool|
                       FunctionApply|FunctionCall|
                      (InfixExpression BinoryOperator InfixExpression)|
                       ('('InfixExpression')')|
                       (UnaryOperator InfixExpression).
StringExpression   ::= String ('++' StringExpression)*.
ListExpression     ::= List ('++' ListExpression).
List               ::= '[' ArgumentExpression? ']'.
ArgumentExpression::= ComplexArgument* ContinuousExpression?.
ComplexArgument    ::= Ident|FunctionApply|StringExpression|Number|
                      (ComplexArgument BinoryOperator ComplexArgument)|
                       ListExpression|('('ComplexArgument')').
FunctionCall       ::= Ident ArgumentExpression?.
FunctionApply      ::= Ident '.' (Ident|ListExpression).
BinoryOperator     ::= '+'|'++'|'-'|'*'|'**'|'/'|'//'|'%'|
                       '='|'!='|'>'|'<'|'<='|'>='|'&&'|'||'.
UnaryOperator      ::= '+'|'-'|'!'.
```

Пример выражений в языке:

```
; "string1 string2"
"string1" ++ " " ++ "string2"

; 6
2 + 2 * 2
```

2.3.8 Вставки на языке Scheme

Для использования кода на `Scheme` необходимо написать ключевое слово `scheme`. После него может следовать только одна конструкция на `Scheme`, заключённая в скобки. Если необходимо описать сразу несколько конструкций, то нужно либо объединить их в конструкцию `begin`, либо писать ключевое слово `scheme` перед каждой.

```
Scheme ::= 'scheme' SchemeCode.
```

Если необходимо использовать функции, описанные на языке `Scheme` в дальнейшем коде, то нужно их экспортировать. Сделать это можно с помощью ключевого слова `export`. После него должны быть перечислены имена

всех функций, которые будут использоваться.

```
Export ::= 'export' Ident+.
```

2.3.9 Комментарии

Любая последовательность символов после символа ; и до символа переноса строки является комментарием и будет проигнорирована компилятором.

```
Comment ::= ';' (?= '\n')* '\n'.
```

2.3.10 Программа

Программа представляет собой набор определений функций, выражений, комментариев и вставок на языке Scheme.

```
Program ::= FunctionDefinition | InfixExpression |
            Scheme | Export | Comment | Memoize .
```

Пример программы на разрабатываемом языке:

```
; объявление функции вычисления факториала числа
n! 0 <- 1
n! n <- n * n! (n - 1)

; мемоизация функции n!
memo n!

; вычисление факториала числа 5
n! 5
; вычисление факториала числа 10
n! 10
```

3 Компоненты среды разработки

Компилятор языка состоит из пяти основных компонентов. К ним относятся: чтение входного потока, лексический, синтаксический и семантический анализаторы и генератор кода.

3.1 Чтение входного потока

На этом этапе происходит чтение текста программы из файла и предварительное его разделение на «слова».

«Словом» является:

- любая последовательность символов, заключённая в двойные кавычки;
- любой терминальный символ (символ переноса строки, пробел, табуляция, окончание файла);
- последовательность символов, образующая слово `scheme`, если она не входит в состав другого «слова»;
- последовательность символов, заключённых в круглые скобки, следующая за словом `scheme` и отделённая от него одним или несколькими «словами», являющимися терминальными символами;
- любая иная последовательность символов, не содержащая терминальных символов.

После этого этапа мы получаем список «слов», который подаётся на вход лексическому анализатору.

3.2 Лексический анализ

На этом этапе осуществляется преобразование последовательности «слов» в последовательность токенов. Это происходит по следующей схеме.

Сначала проверяется, входит ли данное «слово» в список ключевых слов. Если проверка оказывается успешной, то вычисляются координаты и создаётся токен с тэгом `tag-kw`, значением которого будет являтся само «слово». При этом, если ключевое слово является словом `scheme`, то ближайшее

следующее за ним нетерминальное слово будет значением нового токена с тэгом `tag-schm`.

Затем осуществляется проверка является ли «слово» корректным числом. Если так, то высчитываются его координаты и создаётся токен с числовым тэгом. Значению этого токена присваивается вычисленное на этапе проверки число.

После этого проверяется, является ли «слово» «строкой», то есть последовательностью символов, заключённых в кавычки. Аналогично предыдущим пунктам вычисляются координаты и создаётся новый токен с тэгом `tag-str`.

В случае, если ни одна из этих проверок не увенчалась успехом, данное «слово» преобразуется в последовательность символов и дальнейшая проверка будет по-символьной.

Если среди последовательности символов нам встретился символ `;`, дальнейшие символы этого «слова» и последующих игнорируются, пока не встретится символ переноса строки.

Если среди последовательности символов встречается один из следующих: `( ) [ ] { } . :`, то эта последовательность разбивается на три части:

1) собственно символ;
2) последовательность символов до него (возможно, пустая);
3) последовательность символов после (возможно, пустая).

Первая часть преобразуется в токен, с тэгом, соответствующем символу. Вторая и третья подвергаются анализу как отдельные «слова».

Для каждого элемента группы символов `\ < ^ # !`, существует свой тэг. Но он останется у токена, только если этот символ был в «слове» единственным.

Итоговый тэг для следующей группы символов: `- + * t f / > | & =` определяется по тому является ли символ первым в слове или он следует за каким-то другим. Например, символ `-` может получить тэг `tag-mns`, если этот символ в слове первый. Если перед ним в «слове» стоит символ `<` с тэгом `tag-lwr`, то итоговый тэг изменится на `tag-to`. Но если перед символом `-` стоит какой-то другой символ или последовательность, то итоговый тэг будет

— `tag-sym`, и вся эта последовательность станет токеном идентификатора.

3.3 Синтаксический анализ

Синтаксический анализ осуществляется с помощью метода рекурсивного спуска. При этом разбор математических выражений осуществляется с помощью алгоритма сортировочной станции.

3.3.1 Метод рекурсивного спуска

Для каждого нетерминала, описанного в пункте 2 создаётся функция. В этой функции сначала определяется вложенность токена по отступу. Затем проверяется выполнение правила грамматики, определяющего нетерминал [13].

Текущий токен, хранится в глобальной для функций-правил переменной. Входная функции соответствует правилу `Program`, описаному в пункте 2.3.10.

При возникновении ошибки, она записывается и список ошибок и продолжается проверка. После обработки последнего токена печатается печатается список ошибок с указанием координат ошибки и сообщением о её типе.

3.3.2 Алгоритм сортировочной станции

Для будущей возможности определения функций-операторов с приоритетами операторов для разбора выражений был выбран алгоритм сортировочной станции. В результате успешного разбора корректного выражения в инфиксной нотации мы получим его запись в обратной польской нотации. Выражения на Scheme имеют прямую польскую запись. Поэтому для их вычисления, понадобится только преобразовать обратную польскую запись к прямой.

3.3.3 Дерево разбора

Синтаксический анализатор возвращает дерево разбора. Для функции вычисления факториала, описанной в пункте 1, её мемоизации и вызовов:

```
memo n!
n! 5
n! 10
```

будет построено следующее дерево:

```
(#(func-def
   (#(func-decl
      (#(func-name #(tag-sym #(1 1) "n!"))
       #(argument (#(simple-argument #(tag-num #(1 4) 0))))))
    #(func-to #(tag-to #(1 6) "<-"))
    #(expr (#(tag-num #(1 9) 1)))))
 #(func-def
   (#(func-decl
      (#(func-name #(tag-sym #(2 1) "n!"))
       #(argument (#(simple-argument #(tag-sym #(2 4) "n"))))))
    #(func-to #(tag-to #(2 6) "<-"))
    #(expr
      (#(func-decl (#(func-name #(tag-sym #(2 9) "n"))))
       #(func-decl
         (#(func-name #(tag-sym #(2 13) "n!"))
          #(argument
            (#(expr
               (#(func-decl (#(func-name #(tag-sym #(2 17) "n"))))
                #(tag-num #(2 21) 1)
                #(tag-mns #(2 19) "-"))))))))
       #(tag-mul #(2 11) "*")))))
 #(memo
   (#(memo-word #(tag-kw #(4 1) "memo"))
    (#(func-name #(tag-sym #(4 6) "n!")))))
 #(expr
   (#(func-decl
      (#(func-name #(tag-sym #(6 1) "n!"))
       #(argument (#(simple-argument #(tag-num #(6 4) 5))))))))
 #(expr
```

```
(#(func-decl
   (#(func-name #(tag-sym #(7 1) "n!"))
    #(argument (#(simple-argument #(tag-num #(7 4) 10)))))))))
```

3.4 Семантический анализ

Семантический анализатор принимает синтаксическое дерево, созданное в процессе синтаксического анализа. Семантический анализатор составляет модель программы, которая состоит из таблицы символов и списка выражений.

Проходя по синтаксическому дереву семантический анализатор составляет таблицу символов — отображение идентификаторов символов, в описания соответствующих этим символам сущностей[13]. Здесь символ — именованная сущность, определяемая парой `<name, info>`, где `name` — идентификатор сущности, а `info` — её описание.

Описание символа представляет собой список из описания мемоизации и возможных значений этого символа.

Мемоизация описывается следующим образом:

- если, не было указано, что функция, или одна из функций, определённых внутри неё нуждается в мемоизации, то описание мемоизации — это константа `#f`;
- если, было указано, что функцию необходимо мемоизировать, но не было указано, что нужно мемоизировать функции, определённые внутри неё, то описание мемоизации — символ `:memo`;
- если, нужно мемоизировать какую-то внутреннюю функцию данной функции, то описание мемоизации — список имён переменных для мемоизации, созданный из слова `:memo`, имени функции, имени вложенной функции, которой требуется мемоизация. Если, при этом саму функцию также необходимо мемоизировать, то символ `:memo`, также добавляется в список описания мемоизации.

Одно значение функции хранится в виде вектора из трёх элементов:

1) описания аргументов;
2) списка внутренних определений;
3) списка выражений.

Описание аргументов также является вектором, но состоящим из четырёх элементов:

1) функции проверки количества аргументов;
2) списка функций проверки значений аргументов;
3) списка имён аргументов;
4) функции проверки равенства аргументов с одинаковыми идентификаторами.

Все эти функции создаются в процессе семантического анализа.

Повторяющиеся имена заменяются на те, которые не могут быть идентификаторами Языка, но являются корректными в Scheme. Для замены создаётся имя, начинающееся с двоеточия. Таким же образом заменяются цифры. Для списков хранится список имён. «Хвосты» записываются в виде `(:continuous xs)` где `xs` – имя переменной, в которой будут хранится все неописанные в основном списке аргументы.

Функции проверки количества аргументов отличаются для функций, у которых в списке аргументов есть «хвост» от тех, у которых их нет. Для первых — это проверка на то, что аргументов не меньше, чем перечислено до «хвоста». Для вторых — проверка количества на равенство.

Список внутренних определений является таблицей символов.

Список выражений является промежуточным вариантом их представления между синтаксическим деревом и итоговым сгенерированным кодом. Выполняется преобразование вызовов функций и применения функций в удобный для итоговой генерации вид.

Когда встречается вызов функции осуществляется проверка существования соответствующего символа в таблице и выполняются проверки аргументов. В случае если такой символ не найден или если список аргументов не соответствует ни одному из описания — в список ошибок семантического анализатора добавляется ошибка с описанием.

Для синтаксического дерева, приведённого в предыдущем пункте получена следующая модель программы:

```
(((("n!"
    :memo
    #(#((lambda (:x) (= :x 1))
        ((lambda (x) (eqv? x 0)))
        (:_)
        (lambda :args #t))
      ()
      (((1))))
    #(#((lambda (:x) (= :x 1))
        ((lambda (x) #t))
        ("n")
        (lambda :args #t))
      ()
      ((("n" (:func-call "n!" (("n" 1 "-"))) "*"))))))
 (((:func-call "n!" 5)) ((:func-call "n!" 10))))
```

3.5 Генерация кода

На этапе генерации кода выполняется преобразование математических выражений из обратной польской нотации в прямую Для всех символов из таблицы генерируются функции. Объединение возможных значений функции осуществлено с помощью видоизменённой `cond`-конструкции. В случае выполнения условий из описания аргументов осуществляется вычисление списка внутренних определений и выражений. Иначе проверяются условия из описания аргументов других возможных значений. На случай, если ни одна из проверок не увенчается успехом генерируется условие с выводом ошибки.

Если для функции указано, что она должна быть мемоизирована, то функция генерируется со статической переменной. Если для неё указано, что какие-то внутренние функции должны быть мемоизированы, то статические переменные создаются для каждой такой внутренней функции.

Сгенерированный код приписывается к базовым функциям.

Пример кода, сгенерированного на основе модели, полученной на предыдущем этапе, описанной в пункте 3.4:

```
(define n!
  (let ((:memo (list)))
    (lambda :args
      (let ((:m (assoc :args :memo)))
        (if :m
            (cadr :m)
            (let ((:res
                   (:map-cond
                     (((and ((lambda (:x) (= :x 1)) (length :args))
                            (:hash '((lambda (x) (eqv? x 0))) :args)
                            ((lambda :args #t) :args))
                       (apply (lambda (:g_) 1) :args))
                      ((and ((lambda (:x) (= :x 1)) (length :args))
                            (:hash '((lambda (x) #t)):args)
                            ((lambda :args #t) :args))
                       (apply (lambda (n) (* n (n! (- n 1)))) :args))
                      (else :error-func-call)))))
              (set! :memo (cons (list :args :res) :memo))
              :res))))))
(n! 5)
(n! 10)
```

3.6 Сборка компилятора

Так как компилятор языка — объёмный проект, который легко делится на слабо связанные компоненты, то удобнее всего разрабатывать эти компонены в виде отдельных модулей.

Для переносимости кода на языке была выбрана спецификация языка Scheme — r5rs. Эта спецификация не поддерживает модульность, поэтому для сборки компилятора была написан скрипт на языке Python.

Этот скрипт обрабатывает основной модуль компилятора, объединяет в нём все необходимые модули. Также этот скрипт прописывает пути к файлу программы на исходном языке и путь к файлу, в котором будет записан

результат компиляции для входной программы.

4 Возможности языка

4.1 Сопоставление с образцом

Под `сопосталением с образцом` в языках программирования подразумевается метод анализа и обработки данных, основанный на выполнении каких-либо действий в зависимости от совпадения данных с образцом. Такими образцами могут служить:

- конкретное значение (например, числовая константа);
- предикат — функция, возвращающая логическое значение;
- тип данных (проверка принадлежности элемента типу);
- какая-то другая конструкция языка.

В разработанном языке сопоставление с образцом используется для объявления функций. При сопоставлении с образцом в списке аргументов функции, в качестве образца могут использоваться:

- идентификаторы (в том числе повторяющиеся);
- числа;
- списки (в том числе списки списков);
- «хвосты» списков (в том числе в виде списков).

Предикаты не разрешены в качестве образца. Логика сопоставления с образцом в данном языке подразумевает полное соответствие. Если нужно проверять выполнение какого-то более сложного условия, чем простое равенство, то необходимо использовать условную конструкцию `if`.

Чтобы продемонстрировать возможности сопоставления с образцом рассмотрим следующий пример:

```
0? 0 <- #t
0? x <- #f
```

В этом примере мы видим определение функции `0?`, которая проверяет является ли входной аргумент числом `0`. В первой строке происходит сопоставление аргумента с нулём. Если сравнение оказалось успешным, то функция вернёт истинное значение (`#t`). Если сравнение оказалось неудачным, выполнение продолжится во второй строке, где образец-идентификатор

`x` — универсален для обозначения одного аргумента.

Рассмотрим более сложный пример сопоставления с образцом: функцию 0->1, которая заменяет все вхождения числа `0` на `1`.

```
0->1 []                 <- []
0->1 [0   : xs]         <- [1   : 0->1 xs]
0->1 [[0  : xs] : ys]   <- [[1  : 0->1 xs] : 0->1 ys]
0->1 [[x  : xs] : ys]   <- [[x  : 0->1 xs] : 0->1 ys]
0->1 [x   : xs]         <- [x   : 0->1 xs]
```

В первой строке мы видедем образец пустого списка. В случае, если функции подан на вход пустой список — он и будет возвращён.

Во второй строке программы, аргументом функции является список, первым элементом которого, является число `0`. Это число нужно заменить на 1 и рекурсивно вызвать функцию `0->1` для «хвоста» списка `xs`.

В третьей строке программы, аргумент является списком, первым элементом которого является список. Первым элементом внутреннего списка, является число `0` — первым элементом вычисленного списка будет являться список, первый элемент которого равен `0`.

В четвёртой строке — аргумент также является списком, первым элементом которого является список. Но, так как образец из предыдущей строки не подошёл, первый элемент внутреннего списка не является нулём, а значит, его не нужно заменять, однако замена может быть нужна как для какого-то количества элементов «хвоста» внутреннего списка (`xs`), так и для какого-то количества элементов «хвоста» внешнего списка (`ys`). Поэтому осуществляется рекурсивный вызов функции `0->1` для этих списков (`xs`, `ys`).

Наконец, в пятой строке аргументом является список, первый элемент которого не яляется нулём или списком, содержащим ноль. Для функции, вызванной с таким элементом, будет осуществлён рекурсивный вызов для «хвоста» списка `xs`.

4.2 Повторные переменные

Одной из характерных особенностей разработанного языка является возможность использования повторных переменных в образцах.

Когда в шаблоне два элемента имеют одно и то же имя, эти элементы считаются равными. Следовательно, для того, чтобы сопоставление с таким образцом прошло успешно, необходимо соблюдение равенства соответствующих элементов.

Рассмотрим следующий пример:

```
element? x [] <- #f
element? x [x : xs] <- #t
element? x [y : ys] <- element? x ys
```

В этом примере определяется предикат `element?`, который возвращает логическое значение `#t` (истина), если первый аргумент этой функции, является элементом второго.

В первой строке программы вторым аргументом является пустой список. Так как никакой элемент не входит в пустой список, `element?` от этого набора аргументов вернёт логическое значение `#f` (ложь).

Во второй строке, первым аргументом функции является идентификатор `x`, в то время как второй аргумент, является списком, первым элементом которого является идентификатор `x`. Равенство идентификаторов в одной области видимости подразумевает равенство значений элементов, соответствующих этим идентификатам. А значит, первый аргумент функции является элементом второго аргумента. Поэтому функция `element` в данном случае вернёт логическое значение `#t` (истина).

В третьей строке первый аргумент `x` не равен первому элементу списка, являющегося вторым аргументом, поэтому для того, чтобы определить является ли первый аргумент элементом второго, нужно осуществить рекурсивный вызов этой функции с таким же первым аргументом (`x`) и «хвостом» второго аргумена (`ys`).

4.3 Мемоизация

Мемоизация — это сохранение результатов выполнения функции с целью предотвращения повторных вычислений.

Мемоизация очень полезна для функций с большим количеством вычислений. При её использовании вычисления будут произведены только один раз, а затем они будут получены из памяти.

Как правило мемоизация используется для таких функций как функция вычисления `n` - ного числа ряда Фибоначчи или Трибоначчи. Эти функции рекурсивны и объём вычислений возрастает пропорционально элементам этих рядов. Без мемоизации эти функции работают очень долго и глубина рекурсии растёт очень быстро, что делает их вычисление очень затратными.

При мемоизации этих функций количество рекурсивных вызовов ограничено числом `n`. потому что вместо них будут использоваться сохранённые ранее значения.

Мемоизация в разработанном языке осуществляется с помощью ключевого слова `memo`.

Рассмотрим следующий пример:

```
fibonacci 0 <- 1
fibonacci 1 <- 1
fibonacci n <- fibonacci (n - 1) + fibonacci (n - 2)

memo fibonacci
```

В примере выше описывается функция вычисления `n` - ного числа Фибоначчи и указывается, что для этой функции нужно использовать мемоизацию.

Рекомендуется использовать мемоизацию для функций, которые будут многократно вызваны с одинаковыми наборами аргументов. Так как при мемоизации функции, все вычисленные значения будут хранится до конца работы программы, то не рекомендуется её использовать для функций, редко повторно вызываемых с тем же набором входных аргументов.

5 Руководство пользователя

5.1 Сборка и инсталляция

Перед началом работы необходимо убедиться, что на компьютере установлены `git`, `python2.7` и хотя бы один из компиляторов или интерпретаторов языка Scheme (например, `guile`).

Для установки компилятора языка необходимо выполнить следующую последовательность действий:

```
git clone https://github.com/juleari/mlscheme.git
cd mlscheme/mllang
sudo make install
```

После этого будет доступна команда mlscheme.

5.2 Компиляция программы

После сборки проекта в консоли становится доступна команда `mlscheme`. При вызове её без аргументов на экран выведется сообщение об использовании:

```
usage: mlscheme <key> <args>
                -h : help
                -g <path_in> <compiler_file> <path_out>: generate
   compiler for path_in file to compiler_file to out_file
                -c <echo_program> <path_in> <path_out>: compile
   echo_program path_in to out_file
                -l <echo_program> <path_in> <path_out>: load
   compiled echo_program path_in
```

Для сборки компилятора необходимо вызвать `mlscheme` с ключом `-g` и параметрами: `<path_in>`, `<compiler_file>` и `<path_out>`

6 Тестирование

6.1 Удобство использования языка

В таблице 2 приведены реализации одних и тех же функций на Языке и на Scheme. Как видно из таблицы, код на Языке легче читается из-за отстутствия большого количества скобок, привычной инфиксной нотации и сопоставления с образцом. Язык позволяет достичь более краткой записи при заметно меньшем числе повторяющихся элементов синтаксиса. Использование повторных переменных в примере №2 позволяет существенно улучшить восприятие кода.

Таблица 2 — Соответствие конструкций Языка конструкциям языка Scheme

Входной язык	**Scheme**

`n! 0 <- 1`
`n! n <- n * n! (n - 1)`
	`(define (n! n)`
	`  (if (zero? n)`
	`  1`
	`  (* n (n! (- n 1)))))`

`count x []  <- 0`
`count x [ x : xs ] <-`
`  1 + count x xs`
`count x [ y : xs ] <-`
`  count x xs`
	`(define (count x xs)`
	`  (if (null? xs)`
	`  0`
	`  (if (equal? x (car xs))`
	`  (+ 1 (count x (cdr xs)))`
	`  (count x (cdr xs)))))`

`cycle xs 0 <- []`
`cycle xs n <-`
`  xs ++ cycle xs ( n - 1 )`
	`(define (cycle xs n)`
	`  (if (zero? n)`
	`  '()`
	`  (append xs`
	`  (cycle xs`
	`  (- n 1)))))`

`day-of-week day month year <-`
`  a <- (14 - month) // 12`
 	
`  y <- year - a`
`  m <- month + (a * 12) - 2`
 	
`  (7000 + day + y`
`  + (y // 4)`
`  + (y // 400)`
`  + (31 * m // 12)`
 	
`  - (y // 100)) % 7`
	`(define (day-of-week day month year)`
	`  (let* ((a (quotient (- 14 month)`
	`  12))`
	`  (y (- year a))`
	`  (m (- (+ month (* a 12))`
	`  2)))`
	
	`  (remainder (- (+ 7000 day y`
	`  (quotient y 4)`
	`  (quotient y 400)`
	`  (quotient`
	`  (* 31 m)`
	`  12))`
	`  (quotient y`
	`  100))`
	`  7))) `

6.2 Корректность работы программ

Тестирование корректности работы программы представлено в таблице 3.

Таблица 3 — Тестирование корректности работы программы

Код программы, на исходном языке	**Вывод**	**Результат**	**ok**

`day-of-week day month year <-`
`  a <- (14 - month) // 12`
`  y <- year - a`
`  m <- month + (a * 12) - 2`
 	
`  (7000 + day + y`
`  + (y // 4)`
`  + (y // 400)`
`  + (31 * m // 12)`
`  - (y // 100)) % 7`
 	
`day-of-week 17 5 2016`
`day-of-week 10 4 2016`
`day-of-week 29 3 2016`
`day-of-week 20 4 2016`
	`LEXER OK!`
	
	`SYNTAX OK!`
	
	`SEMANTIC OK!`
		2
		0
		2
		3
			ok

`n! 0 <- 1`
`n! n <- n * n! (n - 1)`
 	
`n! 5`
`n! 10`
	`LEXER OK!`
	
	`SYNTAX OK!`
	
	`SEMANTIC OK!`
		
		120
		3628800
			ok

Таблица 3 — Тестирование корректности работы программы

Код программы, на исходном языке	**Вывод**	**Результат**	**ok**

`replace pred? proc [] <- []`
`replace pred?`
`  proc`
`  [ x : xs ] <-`
`if | pred? x ->`
`  [ proc.[x] :`
`  replace pred? proc xs ]`
`  |  ->`
`  [ x :`
`  replace pred? proc xs ]`
 	
`replace zero?`
`  \ x -> x + 1`
`  [ 0 1 2 3 0 ]`
 	
`replace odd?`
`  \ x -> x * 2`
`  [ 1 2 3 4 5 6 ]`
 	
`replace \ x -> 0 > x`
`  exp`
`  [ 0 1 -1 2 -2 3 -3]`
	`LEXER OK!`
	
	`SYNTAX OK!`
	
	`SEMANTIC OK!`
		[1 1 2 3 0]
		
		[2 2 6 4 10 6]
		
		[0 1
		0.3678794 2
		0.1353352 3
		0.0497870]
			ok

Таблица 3 — Тестирование корректности работы программы

Код программы, на исходном языке	**Вывод**	**Результат**	**ok**

`sum <- 0`
`sum x : xs <- x + sum . xs`
 	
`sum 1 2 3 4`
`sum.[5 6 7 8 9 10]`
	`LEXER OK!`
	
	`SYNTAX OK!`
	
	`SEMANTIC OK!`
		
		`10`
		`45`
			ok

`count x []  <- 0`
`count x [ x : xs ] <-`
`  1 + count x xs`
`count x [ y : xs ] <-`
`  count x xs`
 	
`count 1 [1 2 3 1]`
`count 0 [1 2 3 4]`
`count 5 [1 2 3 4 5 5 5]`
	`LEXER OK!`
	
	`SYNTAX OK!`
	
	`SEMANTIC OK!`
		
		`2`
		`0`
		`3`
			ok

`flatten [] <- []`
`flatten [x : xs] <-`
`  flatten x ++ flatten xs`
`flatten x <- [x]`
 	
`flatten [[1] 2 [[3 4] [5 [6]]]]`
	`LEXER OK!`
	
	`SYNTAX OK!`
	
	`SEMANTIC OK!`
		
		`[1 2 3 4 5 6]`
			ok

`element? x [] <- #f`
`element? x [x : xs] <- #t`
`element? x [y : ys] <-`
`  element? x ys`
 	
`element? 1 [3 2 1]`
`element? 4 [3 2 1]`
	`LEXER OK!`
	
	`SYNTAX OK!`
	
	`SEMANTIC OK!`
		
		`#t`
		`#f`
			ok

6.3 Производительность

6.3.1 Производительность работы компилятора

Анализ производительности работы компилятора производился в несколько этапов.

На первом этапе определена зависимость времени работы компилятора от количества различных определений функций. В качестве такой функции была выбрана функция `inc`, определённая следующим образом:

```
inc x <- x + 1
```

Каждое определение имело уникальный идентификатор, состоящий из слова `inc` и номера этого определения. Результаты измерения времени работы компилятора от количества таких определений приведено на рисунке 1.

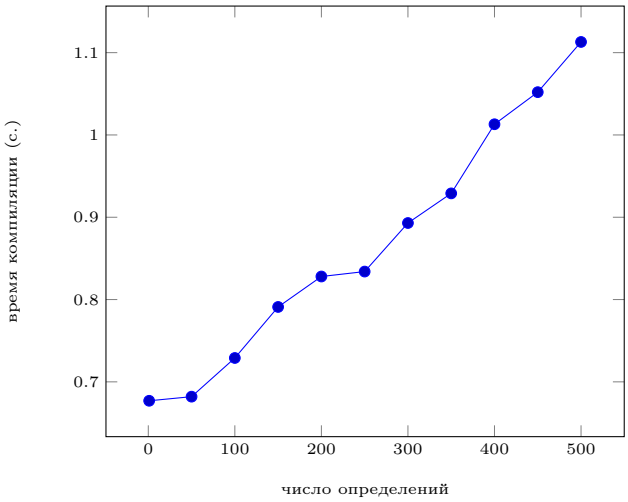

Рисунок 1 — График зависимости времени работы компилятора от количества определений функциӣ

По виду графика можно сделать вывод о линейной зависимости времени работы компилятора от количества определений.

На втором этапе тестирования была определена зависимость времени работы компилятора от количества аргументов функции. График этой зависимости расположен на рисунке 2.

В качестве тестируемой функции была использована функция `test`, идентификаторы аргументов которой состояли из буквы `x` и номера этого

аргумента.

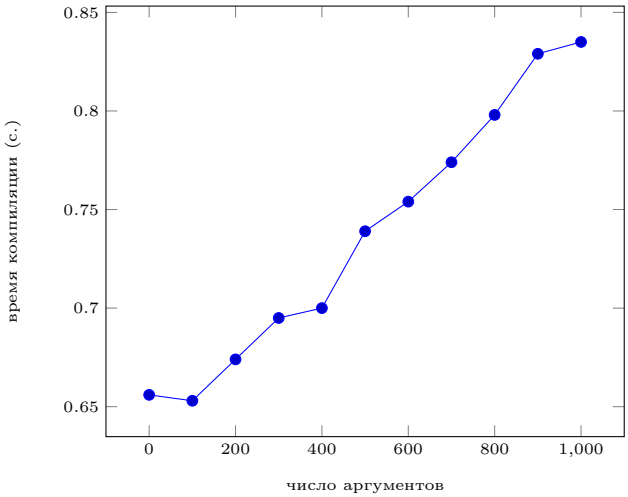

Рисунок 2 — График зависимости времени работы компилятора от количества аргументов

По виду графика можно сделать вывод о линейной зависимости между временем работы компилятора и количеством аргументов функций, описываемых в программе.

На третьем этапе исследовалась зависимость времени работы компилятора от количества повторных аргументов функции. График этой зависимости представлен на рисунке 3.

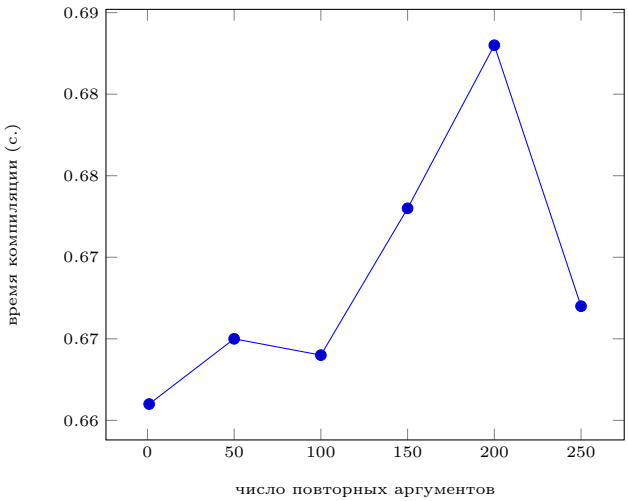

Рисунок 3 — График зависимости времени работы компилятора от количества повторных аргументов

Из графика видно, что при компиляции функций, с количеством аргументов, меньшим 250-ти, время компиляции колеблется в пределах двух сотых секунды.

На четвёртом этапе исследовался «худший случай» объявлений одной функции, у которой первые сто аргументов равны, а последний отличается в каждом определении функции. В этом случае при вызове функции, со списком аргументов, соответствующим списку шаблонов последней строки, будут осуществлены проверки всех предыдущих.

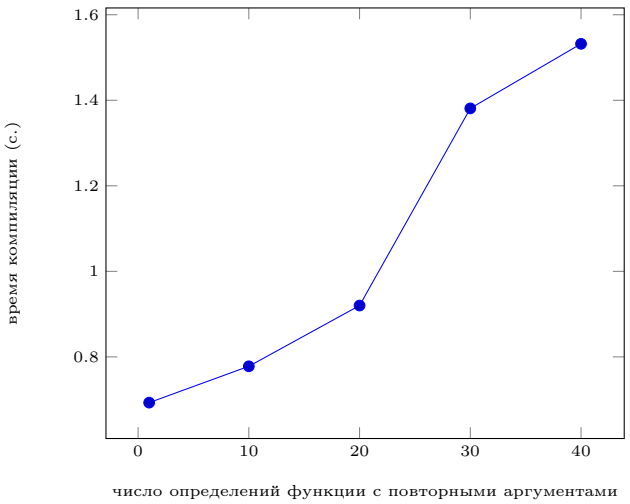

Рисунок 4 — График зависимости времени работы компилятора от количества определений функции с повторными аргументами

6.3.2 Производительность работы программы

Анализ производительности работы программы проводился в два этапа.

На первом этапе исследовалась зависимость времени выполнения программы, осуществляющей вызов функции от набора повторных элементов. На рисунке 5 предоставлен график такой зависимости.

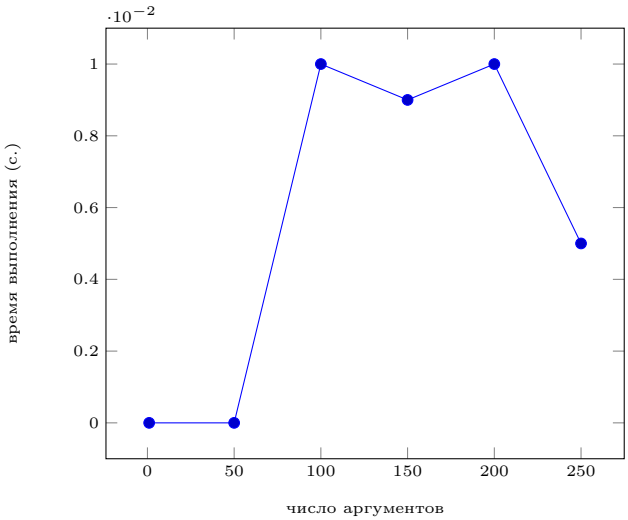

Рисунок 5 — График зависимости времени выполнения от количества повторных аргументов

Как показывает график, при использовании до 50-ти повторных аргументов, функция выполняется практически мгновенно. Однако, при большем их количестве время выполнения существенно увеличивается.

На втором этапе исследовался «худший случай» использования сопоставления с образцом. Результаты исследования предоставлены на рисунке 6.

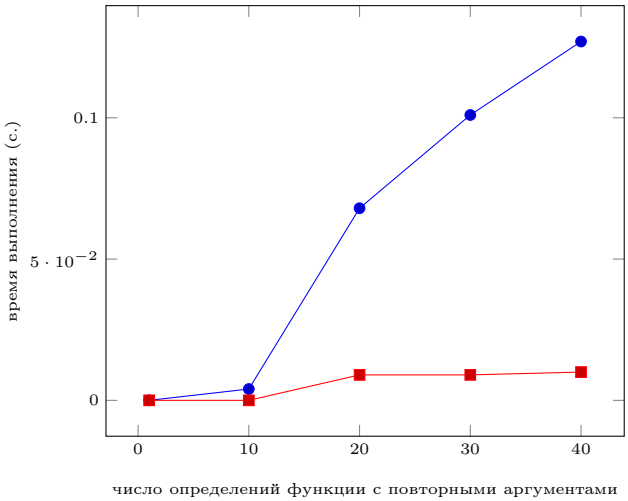

Рисунок 6 — График зависимости времени выполнения от количества определений функции с повторными аргументами

Синим цветом обозначен результат вычисления времени выполнения

такой функции, при её объявлении в виде:

```
test x ... x 1 <- 1
test x ... x 2 <- 1
...
test x ... x n <- 1
```

Здесь `x ... x` обозначает сто идентификаторов `x` подряд.

Красным цветом обозначен результат вычисления, для функции, объявленной следующим образом:

```
test 1 x ... x <- 1
test 2 x ... x <- 1
...
test n x ... x <- 1
```

Сравнивая эти зависимости между собой, можно сделать вывод о выгоде использования второго типа записи. Таким образом, можно рекомендовать организовывать список аргументов так, чтобы наиболее частные аргументы находились в списке аргументов более общих.

6.3.3 Производительность при мемоизации

Тестирование производительности при мемоизации производилось для функции вычисления `n` - ного числа фибоначчи.

```
fibonacci 0 <- 1
fibonacci 1 <- 1
fibonacci n <- fibonacci (n - 1) + fibonacci (n - 2)

memo fibonacci
```

Сравнение результатов приведено на рисунке 7.

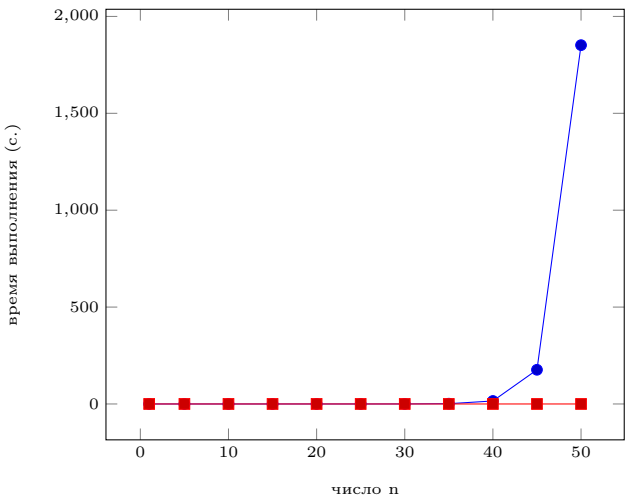

Рисунок 7 — График тестирования производительности при мемоизации

На графике синим цветом обозначена зависимость времени выполнения функции вычисления `n` - ного числа ряда Фибоначчи от числа `n`, без мемоизации. Красным цветом — с мемоизацией.

Как видно из графика мемоизация существенно увеличивает скорость вычислений для числа `n` большего 40.

Сравнение проведено с помощью `,time` для версии компилятора `guile (GNU Guile) 2.0.11`

7 Технико-экономическое обоснование

Разработка программного обеспечения — достаточно трудоемкий и длительный процесс, требующий выполнения большого числа разнообразных операций. Организация и планирование процесса разработки программного продукта или программного комплекса при традиционном методе планирования предусматривает выполнение следующих работ:

- формирование состава выполняемых работ и группировка их по стадиям разработки;
- расчет трудоемкости выполнения работ;
- установление профессионального состава и расчет количества исполнителей;
- определение продолжительности выполнения отдельных этапов разработки;
- построение календарного графика выполнения разработки;
- контроль выполнения календарного графика.

Трудоемкость разработки программной продукции зависит от ряда факторов, основными из которых являются следующие: степень новизны разрабатываемого программного комплекса, сложность алгоритма его функционирования, объем используемой информации, вид ее представления и способ обработки, а также уровень используемого алгоритмического языка программирования. Чем выше уровень языка, тем трудоемкость меньше.

По степени новизны разрабатываемый проект относится к *группе новизны B* – разработка программной продукции, имеющей аналоги.

По степени сложности алгоритма функционирования проект относится к *3 группе сложности* - программная продукция, реализующая алгоритмы стандартных методов решения задач.

По виду представления исходной информации и способа ее контроля программный продукт относится к *группе 12* - исходная информация представлена в форме документов, имеющих различный формат и структуру и *группе 22* - требуется печать документов одинаковой формы и содержания, вывод массивов данных на машинные носители.

7.1 Трудоемкость разработки программной продукции

Трудоемкость разработки программной продукции (τ_{PP}) может быть определена как сумма величин трудоемкости выполнения отдельных стадий разработки программного продукта из выражения:

$$\tau_{PP} = \tau_{TZ} + \tau_{EP} + \tau_{TP} + \tau_{RP} + \tau_V,$$

где τ_{TZ} — трудоемкость разработки технического задания на создание программного продукта; τ_{EP} — трудоемкость разработки эскизного проекта программного продукта; τ_{TP} — трудоемкость разработки технического проекта программного продукта; τ_{RP} — трудоемкость разработки рабочего проекта программного продукта; τ_V — трудоемкость внедрения разработанного программного продукта.

7.1.1 Трудоемкость разработки технического задания

Расчёт трудоёмкости разработки технического задания (τ_{TZ}) [чел.-дни] производится по формуле:

$$\tau_{TZ} = T^Z_{RZ} + T^Z_{RP},$$

где T^Z_{RZ} — затраты времени разработчика постановки задачи на разработку ТЗ, [чел.-дни]; T^Z_{RP} — затраты времени разработчика программного обеспечения на разработку ТЗ, [чел.-дни]. Их значения рассчитываются по формулам:

$$T^Z_{RZ} = t_Z * K^Z_{RZ},$$

$$T^Z_{RP} = t_Z * K^Z_{RP},$$

где t_Z – норма времени на разработку ТЗ на программный продукт (зависит от функционального назначения и степени новизны разрабатываемо-

го программного продукта), [чел.-дни]. В нашем случае по таблице получаем значение (группа новизны – В, функциональное назначение – технико-экономическое):

$$t_Z = 37.$$

K_{RZ}^Z — коэффициент, учитывающий удельный вес трудоемкости работ, выполняемых разработчиком постановки задачи на стадии ТЗ. В нашем случае (совместная разработка с разработчиком ПО):

$$K_{RZ}^Z = 0.65.$$

K_{RP}^Z — коэффициент, учитывающий удельный вес трудоемкости работ, выполняемых разработчиком программного обеспечения на стадии ТЗ. В нашем случае (совместная разработка с разработчиком постановки задач):

$$K_{RP}^Z = 0.35.$$

Тогда:

$$\tau_{TZ} = 37 * \ (0.35 + 0.65) = 37.$$

7.1.2 Трудоемкость разработки эскизного проекта

Расчёт трудоёмкости разработки эскизного проекта (τ_{EP}) [чел.-дни] производится по формуле:

$$\tau_{EP} = T_{RZ}^E + T_{RP}^E,$$

где T_{RZ}^E — затраты времени разработчика постановки задачи на разработку эскизного проекта (ЭП), [чел.-дни]; T_{RP}^E — затраты времени разработчика программного обеспечения на разработку ЭП, [чел.-дни]. Их значения рассчитываются по формулам:

$$T_{RZ}^E = t_E * K_{RZ}^E,$$

$$T_{RP}^{E} = t_E * K_{RP}^{E},$$

где t_E – норма времени на разработку ЭП на программный продукт (зависит от функционального назначения и степени новизны разрабатываемого программного продукта), [чел.-дни]. В нашем случае по таблице получаем значение (группа новизны – В, функциональное назначение – технико-экономическое):

$$t_E = 77.$$

K_{RZ}^{E} — коэффициент, учитывающий удельный вес трудоемкости работ, выполняемых разработчиком постановки задачи на стадии ЭП. В нашем случае (совместная разработка с разработчиком ПО):

$$K_{RZ}^{E} = 0.7.$$

K_{RP}^{E} — коэффициент, учитывающий удельный вес трудоемкости работ, выполняемых разработчиком программного обеспечения на стадии ТЗ. В нашем случае (совместная разработка с разработчиком постановки задач):

$$K_{RP}^{E} = 0.3.$$

Тогда:

$$\tau_{EP} = 77 * \ (0.3 + 0.7) = 77.$$

7.1.3 Трудоемкость разработки технического проекта

Трудоёмкость разработки технического проекта (τ_{TP}) [чел.-дни] зависит от функционального назначения программного продукта, количества разновидностей форм входной и выходной информации и определяется по формуле:

$$\tau_{TP} = (t_{RZ}^{T} + t_{RP}^{T}) * K_V * K_R,$$

где t^T_{RZ} — норма времени, затрачиваемого на разработку технического проекта (ТП) разработчиком постановки задач, [чел.-дни]; t^T_{RP} — норма времени, затрачиваемого на разработку ТП разработчиком ПО, [чел.-дни]. По таблице принимаем (функциональное назначение — технико-экономическое планирование, количество разновидностей форм входной информации — 1 (файл с текстом программы на исходном языке), количество разновидностей форм выходной информации — 1 (файл с текстом программы на языке Scheme)):

$$t^T_{RZ} = 30,$$

$$t^T_{RP} = 8.$$

K_R — коэффициент учета режима обработки информации. По таблице принимаем (группа новизны — B, режим обработки информации — реальный масштаб времени):

$$K_R = 1.26.$$

K_V — коэффициент учета вида используемой информации, определяется по формуле:

$$K_V = \frac{K_P * n_P + K_{NS} * n_{NS} + K_B * n_B}{n_P + n_{NS} + n_B},$$

где K_P — коэффициент учета вида используемой информации для переменной информации; K_{NS} — коэффициент учета вида используемой информации для нормативно-справочной информации; K_B — коэффициент учета вида используемой информации для баз данных; n_P — количество наборов данных переменной информации; n_{NS} — количество наборов данных нормативно-справочной информации; n_B — количество баз данных. Коэффициенты находим по таблице (группа новизны - B):

$$K_P = 1.00,$$

$$K_{NS} = 0.72,$$

$$K_B = 2.08.$$

Количество наборов данных, используемых в рамках задачи:

$$n_P = 10,$$

$$n_{NS} = 0,$$

$$n_B = 0.$$

Находим значение K_V:

$$K_V = \frac{1.00 * 10 + 0.72 * 0 + 2.08 * 1}{10 + 0 + 1} = 1.098.$$

Тогда:

$$\tau_{TP} = (30 + 8) * 1.098 * 1.26 = 53.$$

7.1.4 Трудоемкость разработки рабочего проекта

Трудоёмкость разработки рабочего проекта (τ_{RP}) [чел.-дни] зависит от функционального назначения программного продукта, количества разновидностей форм входной и выходной информации, сложности алгоритма функционирования, сложности контроля информации, степени использования готовых программных модулей, уровня алгоритмического языка программирования и определяется по формуле:

$$\tau_{RP} = (t^R_{RZ} + t^R_{RP}) * K_K * K_R * K_Y * K_Z * K_{IA},$$

где t^R_{RZ} — норма времени, затраченного на разработку рабочего проекта на алгоритмическом языке высокого уровня разработчиком постановки задач, [чел.-дни]. t^R_{RP} — норма времени, затраченного на разработку рабочего проекта на алгоритмическом языке высокого уровня разработчиком ПО, [чел.-дни]. По таблице принимаем (функциональное назначение — технико-экономическое планирование, количество разновидностей форм входной информации — 1 (файл с текстом программы на исходном языке), количество разновидностей форм выходной информации — 1 (файл

с текстом программы на языке Scheme)):

$$t_{RZ}^{R} = 8,$$

$$t_{RP}^{R} = 51.$$

K_K — коэффициент учета сложности контроля информации. По таблице принимаем (степень сложности контроля входной информации — 12, степень сложности контроля выходной информации — 22):

$$K_K = 1.00.$$

K_R — коэффициент учета режима обработки информации. По таблице принимаем (группа новизны — B, режим обработки информации — реальный масштаб времени):

$$K_R = 1.26.$$

K_Y — коэффициент учета уровня используемого алгоритмического языка программирования. По таблице принимаем значение (интерпретаторы, языковые описатели):

$$K_Y = 0.8.$$

K_Z — коэффициент учета степени использования готовых программных модулей. По таблице принимаем (использование готовых программных модулей составляет менее 25

$$K_Z = 0.8.$$

K_{IA} — коэффициент учета вида используемой информации и сложности алгоритма программного продукта, его значение определяется по формуле:

$$K_I A = \frac{K'_P * n_P + K'_{NS} * n_{NS} + K'_B * n_B}{n_P + n_{NS} + n_B},$$

где K'_P — коэффициент учета сложности алгоритма ПП и вида используемой информации для переменной информации; K'_{NS} — коэффициент учета сложности алгоритма ПП и вида используемой информации для нормативно-

справочной информации; K'_B — коэффициент учета сложности алгоритма ПП и вида используемой информации для баз данных. n_P — количество наборов данных переменной информации; n_{NS} — количество наборов данных нормативно-справочной информации; n_B — количество баз данных. Коэффициенты находим по таблице (группа новизны - B):

$$K'_P = 1.00,$$

$$K'_{NS} = 0.48,$$

$$K'_B = 0.4.$$

Количество наборов данных, используемых в рамках задачи:

$$n_P = 10,$$

$$n_{NS} = 0,$$

$$n_B = 1.$$

Находим значение K_{IA}:

$$K_{IA} = \frac{1.00 * 10 + 0.48 * 0 + 0.4 * 1}{10 + 0 + 1} = 0.945.$$

Тогда:

$$\tau_{RP} = (8 + 51) * 1.00 * 1.26 * 0.8 * 0.8 * 0.945 = 45.$$

7.1.5 Трудоемкость выполнения стадии «Внедрение»

Расчёт трудоёмкости разработки технического проекта (τ_V) [чел.-дни] производится по формуле:

$$\tau_V = (t^V_{RZ} + t^V_{RP}) * K_K * K_R * K_Z,$$

где t^V_{RZ} — норма времени, затрачиваемого разработчиком постановки задач на выполнение процедур внедрения программного продукта, [чел.-дни]; t^V_{RP} — норма времени, затрачиваемого разработчиком программного обеспечения на выполнение процедур внедрения программного продукта, [чел.-дни]. По таблице принимаем (функциональное назначение — технико-экономическое планирование, количество разновидностей форм входной информации — 1 (файл с текстом программы на исходном языке), количество разновидностей форм выходной информации — 1 (файл с текстом программы на языке Scheme)):

$$t^V_{RZ} = 9,$$

$$t^V_{RP} = 11.$$

Коэффициент K_K и K_Z были найдены выше:

$$K_K = 1.00,$$

$$K_Z = 0.8.$$

K_R — коэффициент учета режима обработки информации. По таблице принимаем (группа новизны — B, режим обработки информации — реальный масштаб времени):

$$K_R = 1.26.$$

Тогда:

$$\tau_V = (9 + 11) * 1.00 * 1.26 * 0.8 = 21.$$

Общая трудоёмкость разработки ПП:

$$\tau_{PP} = 37 + 77 + 53 + 45 + 21 = 233.$$

7.2 Расчет количества исполнителей

Средняя численность исполнителей при реализации проекта разработки и внедрения ПО определяется соотношением:

$$N = \frac{t}{F},$$

где t — затраты труда на выполнение проекта (разработка и внедрение ПО); F — фонд рабочего времени. Разработка велась 5 месяцев с 1 января 2016 по 31 мая 2016. Количество рабочих дней по месяцам приведено в таблице 4. Из таблицы получаем, что фонд рабочего времени

$$F = 96.$$

Таблица 4 — Количество рабочих дней по месяцам

Номер месяца	**Интервал дней**	**Количество рабочих дней**
1	01.01.2016 - 31.01.2016	15
3	01.02.2016 - 29.02.2016	20
4	01.03.2016 - 31.03.2016	21
5	01.04.2016 - 30.04.2016	21
6	01.05.2016 - 31.05.2016	19
Итого		96

Получаем число исполнителей проекта:

$$N = \frac{233}{96} = 3$$

Для реализации проекта потребуются 1 старший инженер и 2 простых инженера.

7.3 Ленточный график выполнения работ

На основе рассчитанных в главах 7.1, 7.2 трудоёмкости и фонда рабочего времени найдём количество рабочих дней, требуемых для выполнения каждого этапа разработка. Результаты приведены в таблице 5.

Таблица 5 — Трудоёмкость выполнения работы над проектом

Номер стадии	**Название стадии**	**Трудоёмкость [чел.-дни]**	**Удельный вес [%]**	**Количество раб. дней**
1	Техническое задание	37	11	10
2	Эскизный проект	77	24	23
3	Технический проект	53	35	34
4	Рабочий проект	45	25	24
5	Внедрение	21	5	5
Итого		233	100	96

Планирование и контроль хода выполнения разработки проводится по ленточному графику выполнения работ. По данным в таблице 5 в ленточный график (таблица 6), в ячейки столбца “продолжительности рабочих дней” заносятся времена, которые требуются на выполнение соответствующего этапа. Все исполнители работают одновременно.

7.4 Определение себестоимости программной продукции

Затраты, образующие себестоимость продукции (работ, услуг), состоят из затрат на заработную плату исполнителям, затрат на закупку или аренду оборудования, затрат на организацию рабочих мест, и затрат на накладные расходы.

В таблице 7 приведены затраты на заработную плату и отчисления на социальное страхование в пенсионный фонд, фонд занятости и фонд обязательного медицинского страхования (30.5 %). Для старшего инженера предполагается оклад в размере 120000 рублей в месяц, для инженера предполагается оклад в размере 100000 рублей в месяц.

Таблица 6 — Ленточный график выполнения работ

Номер стадии	Продолжительность [раб.-дни]	Календарные дни																				
		11.01.2016 - 17.01.2016	18.01.2016 - 24.01.2016	25.01.2016 - 31.01.2016	01.02.2016 - 07.02.2016	08.02.2016 - 14.02.2016	15.02.2016 - 21.02.2016	22.02.2016 - 28.02.2016	29.02.2016 - 06.03.2016	07.03.2016 - 13.03.2016	14.03.2016 - 20.03.2016	21.03.2016 - 27.03.2016	28.03.2016 - 03.04.2016	04.04.2016 - 10.04.2016	11.04.2016 - 17.04.2016	18.04.2016 - 24.04.2016	25.04.2016 - 01.05.2016	02.05.2016 - 08.05.2016	08.05.2016 - 15.05.2016	16.05.2016 - 22.05.2016	23.05.2016 - 29.05.2016	30.05.2016 - 31.05.2016
		Количество рабочих дней																				
		5	5	5	5	5	6	3	5	3	5	5	5	5	5	5	5	3	4	5	5	2
1	10	5	5																			
2	23			5	5	5	6	2														
3	34							1	5	3	5	5	5	5	5							
4	24															5	5	3	4	5	2	
5	5																				3	2

Таблица 7 — Затраты на зарплату и отчисления на социальное страхование

Должность	**Зарплата в месяц**	**Рабочих месяцев**	**Суммарная зарплата**	**Затраты на соц. нужды**
Старший инженер	120000	5	600000	183000
Инженер	100000	5	500000	152500
Инженер	100000	5	500000	152500
Суммарные затраты			2088000	

Расходы на материалы, необходимые для разработки программной продукции, указаны в таблице 8.

Таблица 8 — Затраты на материалы

Наименование материала	**Единица измерения**	**Кол-во**	**Цена за единицу, руб.**	**Сумма, руб.**
Бумага А4	Пачка 400 л.	2	200	400
Картридж для принтера HP F4275	Шт.	2	675	1350
Суммарные затраты				1750

В работе над проектом используется специальное оборудование — персональные электронно-вычислительные машины (ПЭВМ) в количестве 9 шт. Стоимость одной ПЭВМ составляет 90000 рублей. Месячная норма амортизации К = 2,7%. Тогда за 5 месяцев работы расходы на амортизацию составят $P = 90000 * 3 * 0.027 * 5 = 36450$ рублей.

Накладные расходы рассчитываются по следующе формуле:

$$C_n = A_n * C_z$$

$$N = 2.1 * 1600000 = 3360000$$

Общие затраты на разработку программного продукта (ПП) составят $2088000 + 1750 + 36450 + 3360000 = 5486200$ рублей.

7.5 Определение стоимости программной продукции

Для определения стоимости работ необходимо на основании плановых сроков выполнения работ и численности исполнителей рассчитать общую сумму затрат на разработку программного продукта. Если ПП рассматривается и создается как продукция производственно-технического назначения, допускающая многократное тиражирование и отчуждение от непосредственных разработчиков, то ее цена P определяется по формуле:

$$P = K * C + Pr,$$

где C — затраты на разработку ПП (сметная себестоимость); K — коэффициент учёта затрат на изготовление опытного образца ПП как продукции производственно-технического назначения ($K = 1.1$); Pr — нормативная прибыль, рассчитываемая по формуле:

$$Pr = \frac{C * \rho_N}{100},$$

где ρ_N — норматив рентабельности, $\rho_N = 30\%$;

Получаем стоимость программного продукта:

$$P = 1.1 * 5486200 + 5486200 * 0.3 = 7680680 \text{ рублей.}$$

7.6 Расчет экономической эффективности

Основными показателями экономической эффективности является чистый дисконтированный доход (NPV) и срок окупаемости вложенных средств. Чистый дисконтированный доход определяется по формуле:

$$NPV = \sum_{t=0}^{T}(R_t - Z_t) * \frac{1}{(1+E)^t},$$

где T — горизонт расчета по месяцам; t — период расчета; R_t — результат, достигнутый на t шаге (стоимость); Z_t — текущие затраты (на шаге t); E — приемлемая для инвестора норма прибыли на вложенный капитал.

На момент начала 2016 года, ставка рефинансирования 11% годовых (ЦБ РФ), что эквивалентно (0.916% в месяц). В виду особенности разрабатываемого продукта он может быть продан лишь однократно. Отсюда получаем

$$E = 0.00916.$$

В таблице 9 находится расчёт чистого дисконтированного дохода. График его изменения приведён на рисунке 8.

Таблица 9 — Расчёт чистого дисконтированного дохода

Месяц	**Текущие затраты, руб.**	**Затраты с начала года, руб.**	**Текущий доход, руб.**	**ЧДД, руб.**
Январь	1096890	1096890	0	-1096890
Февраль	1096890	2193780	0	-2183823.7
Март	1096890	3290670	0	-3260891.4
Апрель	1096890	4387560	0	-4328182.7
Мая	1098640	5486200	7680680	2019802

Согласно проведенным расчетам, проект является рентабельным. Разрабатываемый проект позволит превысить показатели качества существую-

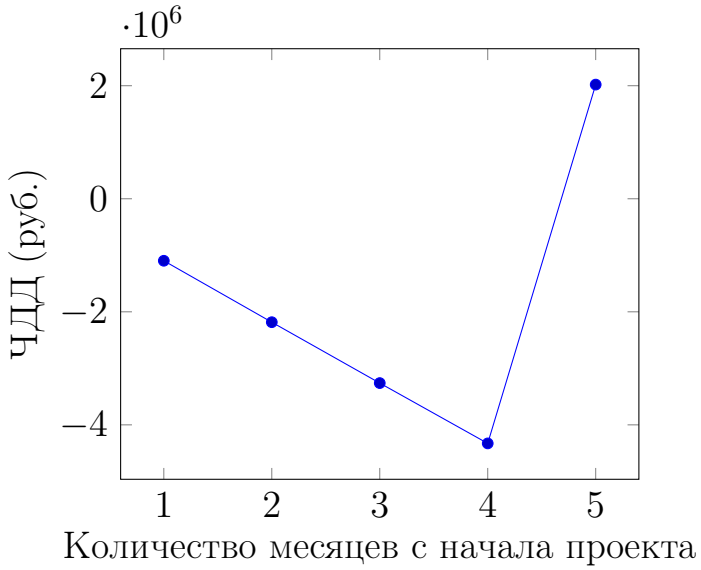

Рисунок 8 — График изменения чистого дисконтированного дохода

щих систем и сможет их заменить. Итоговый ЧДД составил: 2019802 рубля.

7.7 Результаты

В рамках организационно-экономической части был спланирован календарный график проведения работ по созданию подсистемы поддержки проведения диагностики промышленных, а также были проведены расчеты по трудозатратам. Были исследованы и рассчитаны следующие статьи затрат: материальные затраты; заработная плата исполнителей; отчисления на социальное страхование; накладные расходы.

В результате расчетов было получено общее время выполнения проекта, которое составило 96 рабочих дней, получены данные по суммарным затратам на создание и разработку функционального языка, которые составили 5486200 рублей. Согласно проведенным расчетам, проект является рентабельным. Цена данного программного проекта составила 7680680 рублей, итоговый ЧДД составил 2019802 рублей.

ЗАКЛЮЧЕНИЕ

В ходе данной работы был предложен функциональный язык программирования с динамической типизацией и ML-подобным синтаксисом и разработан компилятор этого языка.

В качестве языков-прототипов выступили функциональные языки Scheme, Haskell, объектно-функциональный язык OCaml и язык логического программирования Prolog. В результате анализа синтаксиса, семантики и основных конструкций и возможностей этих языков был выработан оригинальный язык с синтаксисом, близким к языку Haskell и с семантикой, наиболее близкой к языку Scheme. Программы на этом языке не перегружены элементами синтаксиса и ключевыми словами, выражения записываются в нотации, близкой к принятой в математике. На наш взгляд, это способствует удобству использования разработанного языка для быстрого написания коротких программ (скриптов, сценариев), а также для обучения программированию.

В язык введено сопоставление с образцом, которое расширено использованием повторных переменных. Применение таких переменных в образцах является новым для функциональных языков программирования. В ходе выполнения работы на примере было показано, что эта возможность органично дополняет функциональный язык и позволяет писать короткие и выразительные определения функций. При реализации генератора кода было обнаружено, что эта особенность не приводит к значительным накладным расходам при выполнении программ на целевом языке. Таким образом, эта конструкция может быть рекомендована к использованию при разработке новых и дополнения существующих функциональных языков программирования.

Указание выполнить мемоизацию результатов вычисления функции с помощью единственного ключевого слова, введенного в язык, также благоприятно влияет на краткость и выразительность кода.

Разработанный компилятор может быть использован на различных платформах совместно с различными реализациями языка Scheme, поддерживающих спецификацию R5RS.

Имеется возможность сборки компилятора в виде исполнимого файла,

хотя это требует дополнения кода интерфейсом командной стоки, который специфичен для различных реализаций языка Scheme.

Исходя из изложенного, цель работы была достигнута.

В ходе работы было показано, что использование одного и того же языка одновременно в качестве основного прототипа, целевого языка трансляции и основного языка разработки компилятора может быть удачным решением с точки зрения экономии времени при разработке первой версии компилятора нового языка. Также следует отметить, что несмотря на возраст, язык Scheme по-прежнему является гибким, современным и удобным инструментом для исследовательского программирования и быстрого прототипирования приложений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. TIOBE [Электронный ресурс] URL: http://www.tiobe.com/tiobe_index (accessed 17 October 2015)

2. Peter Seibel // Practical Common Lisp // Publication Date: April 6, 2005

3. Revised5 Report on the Algorithmic Language Scheme [Электронный ресурс] URL: http://www.schemers.org/Documents/Standards/R5RS/HTML/ (accessed 26 September 2015)

4. R. Kent Dybvig // The Scheme Programming Language, Fourth Edition // The MIT Press. 2009

5. Information technology — Programming languages — Prolog. 1995

6. An Efficient Unification Algorithm [Электронный ресурс] URL: http://www.nsl.com/misc/papers/martelli-montanari.pdf (accessed 14 November)

7. Братко, Иван. Алгоритмы искусственного интеллекта на языке PROLOG, 3-е издание. Пер. с англ. — М. : Издательский дом «Вильяме», 2004.

8. A Byte of Python [Электронный ресурс] URL: http://python.swaroopch.com/ (accessed 10 June 2016)

9. Yaron Minsky, Anil Madhavapeddy, Jason Hickey // Real world OCaml // 2013

10. Miran Lipovača // LEARN YOU A HASKELL FOR GREAT GOOD! // 2011

11. Орлов С.А., Цилькер Б.Я. // Организация ЭВМ и систем: Учебник для вузов. 2-е изд. — СПб.: Питер, 2011

12. Сергей Александрович Орлов. // Теория и практика языков программирования: Учебник для вузов. Стандарт 3-го поколения. — СПб.: Питер, 2014

13. Скоробогатов С.Ю. // Лекции по курсу «Компиляторы», 2014.

14. Арсеньев В.В., Сажин Ю.Б. Методические указания к выполнению организационно-экономической части дипломных проектов по созданию программной продукции. М.: изд. МГТУ им. Баумана, 1994. 52 с. 2.

ПРИЛОЖЕНИЕ

ПРИЛОЖЕНИЕ А. Функция вычисления дня недели

```
; count days of week
day-of-week day month year <-
    a <- (14 - month) // 12
    y <- year - a
    m <- month + (a * 12) - 2

    (7000 + day + y + (y // 4) + (y // 400) + (31 * m // 12)
        - (y // 100)) % 7

day-of-week 17 5 2016
day-of-week 10 4 2016
day-of-week 29 3 2016
day-of-week 20 4 2016
```

В этом программе определена функция вычисления дня недели по дате.

В этом примере используются определения внутри функции. Так, три переменные `a`, `y` и `m` определены и доступны только внутри функции `day-of-week`.

В процессе выполнения такой программы будут возвращены четыре значения: 2 (вторник), 0 (воскресенье), 2 (вторник), 3 (среда).

ПРИЛОЖЕНИЕ Б. Функция замены всех нулей на единицы

```
0->1 [] <- []
0->1 [0 : xs] <- [1 : 0->1 xs]
0->1 [x : xs] <- [x : 0->1 xs]

0->1 [0 2 7 0 5]
0->1 [0 1 0 1 0]
```

В этом примере используется сопоставление с образцом. Если аргумент

функции — пустой список, то она возвращает пустой список. Если первым элементом списка является число 0, то возвращается список, первым элементом которого является 1, а остаток этого списка вычисляется рекурсивно. Если шаблоны, описанные в первых двух строках не подошли, будет использован шаблон в третьей строке. Он более универсален. Результатом вычисления в этом случае будет список, состоящий из значения аргумента `x` и списка, вычисленного рекурсивно.

Вычисленные в ходе выполнения программы списки выглядят следующим образом:

```
[1 2 7 1 5]
[1 1 1 1 1]
```

ПРИЛОЖЕНИЕ В. Функция подсчёта количества вхождений

```
; count x in xs
count x []          <- 0
count x [ x : xs ] <- 1 + count x xs
count x [ y : xs ] <- count x xs

count 1 [1 2 3 1]
count 0 [1 2 3 4]
count 5 [1 2 3 4 5 5 5]
```

Функция `count` возвращает количество вхождений первого аргумента, в список, являющийся вторым аргументом.

В первой строке мы видим определение функции `count` от `x` и пустого списка. В этом случае количество вхождений `x` в список равно нулю. Во второй строке мы видим определение функции `count` от `x` и списка, первым элементом которого является `x`. В данном случае нам нужно прибавить единицу к количеству вхождений `x` в «хвост» списка `xs`. И, наконец, в третьей строке определена функция `count` от `x` и списка, первым элементом которого является `y`. Так как в строке выше мы предусмотрели равенство `x` и первого элемента списка, то в этой строке определения мы можем быть уверены, что `x` не равен `y`.

Результат выполнения программы:

```
2
0
3
```

ПРИЛОЖЕНИЕ Г. Функция вычисления факториала числа

```
n! 0 <- 1
n! n <- n * n! (n - 1)

memo n!

n! 5
n! 10
```

ПРИЛОЖЕНИЕ Д. Функция вычисления суммы произвольного количества аргументов

```
sum <- 0
sum x : xs <- x + sum . xs

sum 1 2 3 4
sum.[5 6 7 8 9 10]
```

В данном примере определена функция вычисления факториала числа. Запись этой функции похожа на математическую запись этой функции. В случае равенства первого аргумента нулю — будет возвращена единица, иначе будет посчитано произведение числа `n` на факториал от числа `n - 1`.

В четвёртой строке программы указано, что эту функцию необходимо мемоизировать.

Значение, вычисленное для числа 5 — 120. Значение, вычисленное для числа 10 — 3628800.

Стоит отметить, что при вычислении факториала 10, высчитываются только значения этой функции для чисел 9, 8, 7, 6. Значение для числа 5 уже вычисленно при первом вызове функции.

ПРИЛОЖЕНИЕ Е. Функция замены элементов списка

```
replace pred? proc []          <- []
replace pred? proc [ x : xs ] <-
    if | pred? x -> [ proc.[x] : replace pred? proc xs ]
       |         -> [ x : replace pred? proc xs ]

replace zero?
        \ x -> x + 1
        [ 0 1 2 3 0 ]

replace odd?
        \ x -> x * 2
        [ 1 2 3 4 5 6 ]

replace \ x -> 0 > x
        exp
        [ 0 1 -1 2 -2 3 -3]
```

В этой программе описана функция `replace`, принимающая три аргумента:

- предикат (`pred?`) — функцию, проверяющую какое-либо условие;
- функцию одного аргумента (`proc`), с помощью которой будет осуществлено преобразование;
- список.

В случае, если список пуст, возвращается пустой список.

Для списка, не являющегося пустым будет осуществлена проверка первого элемента с помощью предиката. В случае, если условие, описанное в предикате выполнено, будет возвращён список, первым элементом которого будет являться изменённый функцией `proc` первый элемент списка (`x`). «Хвост» итогового списка вычисляется рекурсивно для списка `xs`.

С шестой по восьмую строках программы записан вызов функции `replace` от стандартного предиката `zero?` (проверяющего, является ли входной аргумент нулём), λ-функции, увеличивающей число на единицу и списка

из 5 элементов. Результатом этого вызова будет список `[1 1 2 3 0]`.

С десятой до двенадцатую строки программы описывается вызов функции `replace` от стандартной функции `odd?` (осуществляющей проверку на нечётность), λ-функции, удваивающей свой аргумент и списка из шести чисел. Результатом этого вызова будет список `[1 1 2 3 0]`.

С четырнадцатой по шестнадцатую строку программы дано описание вызова функции от λ-функции, проверяющей, является ли число большим нуля, функции вычисления экспоненты числа и списка из семи аргументов. Результат этого вызова: `[0 1 0.3678794 2 0.1353352 3 0.0497870]`.

ПРИЛОЖЕНИЕ Ё. Функция повтора элемента

```
replicate x 0 <- []
replicate x n <- [ x : replicate x ( n - 1 ) ]

replicate "a" 5
replicate [ "a" "b" ] 3
replicate "a" 0
```

Результат выполнения программы:

```
["a" "a" "a" "a" "a"]
[["a" "b"] ["a" "b"] ["a" "b"]]
[]
```

ПРИЛОЖЕНИЕ Ж. Функция повторения списка

```
cycle xs 0 <- []
cycle xs n <- xs ++ cycle xs ( n - 1 )

cycle [ 0 1 ] 3
cycle [ 'a 'b 'c ] 5
cycle [] 0
```

Результат выполнения программы:

```
[0 1 0 1 0 1]
```

```
["a" "b" "c" "a" "b" "c" "a" "b" "c" "a" "b" "c" "a" "b" "c"]
[]
```

ПРИЛОЖЕНИЕ З. Функция AND для произвольного количества аргументов

```
and-fold          <- #t
and-fold x : xs <- x && and-fold : xs

and-fold #f #f #f
and-fold #f #f #t
and-fold #f #t #t
and-fold #t #t #t
and-fold
```

Результат выполнения программы:

```
#f
#f
#f
#t
#t
```

ПРИЛОЖЕНИЕ И. Вставка кода на языке Scheme

Пример вставки кода на языке Scheme. Вставляемые функции — функции сортировки выбором и вставками.

```
scheme (define (selection-sort pred? xs)
         (define (min-xs xs x)
           (cond ((null? xs)          x)
                 ((pred? (car xs) x) (min-xs (cdr xs) (car xs)))
                 (else               (min-xs (cdr xs) x))))

         (define (swap j xs)
           (let ((xj (list-ref xs j))
                 (vs (list->vector xs)))
             (vector-set! vs j (car xs))
             (vector-set! vs 0 xj)
```

```
            (vector->list vs)))

          (define (ind x xs)
            (- (length xs) (length (member x xs))))

          (define (helper xs)
            (if (null? xs)
                '()
                (let ((x (min-xs xs (car xs))))
                  (cons x (helper (cdr (swap (ind x xs) xs)))))))

          (helper xs))
scheme (define (insertion-sort pred? xs)
          (define (insert xs ys x)
            (cond ((null? ys) (append xs (list x)))
                  ((pred? (car ys) x) (insert (append xs (list (car
   ys))) (cdr ys) x))
                  (else               (append xs (list x) ys))))

          (define (helper xs ys)
            (if (null? ys)
                xs
                (helper (insert '() xs (car ys)) (cdr ys))))

          (helper '() xs))

export selection-sort insertion-sort

selection-sort \ x y -> x <= y
               [9 6 2 4 3 5 7 1 8 0]
insertion-sort \ x y -> x <= y
               [9 6 2 4 3 5 7 1 8 0]
```

Результат выполнения программы:

```
[0 1 2 3 4 5 6 7 8 9]
[0 1 2 3 4 5 6 7 8 9]
```

ПРИЛОЖЕНИЕ К. Функция проверки числа на простоту

```
0? 0 <- #t
0? x <- #f

prime? n <-
    fact 0 <- 1
    fact n <- n * fact (n - 1)

    memo fact

    0?.[ (fact.[n - 1] + 1) % n ]

prime? 11
prime? 12
prime? 3571
```

Результат выполнения программы:

```
#t
#f
#t
```

ПРИЛОЖЕНИЕ Л. Функции НОД и НОК

```
0? 0 <- #t
0? x <- #f

my-gcd a b <-
    r <- a % b

    if  | a < b -> my-gcd b a
        | 0? r  -> b
        |       -> my-gcd b r

my-lcm a b <-
    abs (a * b / my-gcd a b)
```

```
my-gcd 3542 2464
my-lcm 3 4
```

Результат выполнения программы:

```
154
12
```

ПРИЛОЖЕНИЕ М. Метод половинного сечения

```
bisection f a b e <-
    sign 0 <- 0
    sign x <- if | x > 0 -> 1
                 |       -> -1

    mid a b <-
        x <- a + (b - a) / 2

        if   | (abs f x) <= e           -> x
             | (sign f.[b]) = (sign f.[x]) -> mid a x
             |                           -> mid x b

    if | f.[a] = 0 -> a
       | f.[b] = 0 -> b
       |          -> mid a b

bisection cos -3.0 0.0 0.001
```

Результат выполнения программы:

```
-1.5703125
```

ПРИЛОЖЕНИЕ Н. Функция вычисляющая диапазон значений

```
range a b d <-
    if   | a < b -> [a] ++ range (a + d) b d
         |       -> []

range 0 11 3
```

Результат выполнения программы:

```
[0 3 6 9]
```

ПРИЛОЖЕНИЕ О. Функция разворачивания списков

```
flatten [] <- []
flatten [x : xs] <- flatten x ++ flatten xs
flatten x <- [x]

flatten [[1] 2 [3 [4 5] [6 [7 8]]] 9]
```

Результат выполнения программы:

```
[1 2 3 4 5 6 7 8 9]
```

ПРИЛОЖЕНИЕ П. Функция проверки вхождения в список

```
element? x [] <- #f
element? x [x : xs] <- #t
element? x [y : ys] <- element? x ys

element? 1 [3 2 1]
element? 4 [3 2 1]
```

Результат выполнения программы:

```
#t
#f
```