МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Н. Э. БАУМАНА

Факультет информатики и систем управления

Кафедра теоретической информатики и компьютерных технологий

УТВЕРЖДАЮ:

Заведующий кафедрой ИУ-9

_____(И.П. Иванов)

«__»_____201__г.

Расчётно-пояснительная записка

к дипломному проекту

«Функциональный язык программирования с динамической типизацией и ML-подобным синтаксисом»

Исполнитель: Ю.А. Волкова

Группа: ИУ9-121

Руководитель

квалификационной работы

А.В. Дубанов

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.

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».

Все объекты, в том числе функции, являются данными первого порядка, 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 существующие [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.

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).

В языке Scheme всё представлено в виде списков. Таким способ пред-

ставления код не отличается от способа представления данных. Это позволяет работать с кодом как с данными.

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.

```
1 street( saint-petersburg, nevskii ).
2 street( moscow, arbat ).
3 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:

```
1 ?- 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:

```
1 X = moscow;
2 X = berlin;
3 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.

Также, поддерживаются алгебраические типы данных и сопоставление с образцом, чтобы определять и управлять сложными структурами данных [9]. Пример вычисления факториала числа на языке OCaml, используя сопоставление с образцом:

```
1 let rec factorial = function
2     | 0 -> 1
3     | 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 зна-

чения, пока это не нужно. «Ленивость» ускоряет работу программы, однако позволяет допускать ошибки, так как выражение, содержащее ошибку может быть невыполнено.

Haskell обладает статической сильной полной типизацией с атоматическим выводом типов.

Также, Haskell поддерживает функции высшего порядка и частичное применение.

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:

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

1.5 Выбор концепций для 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.

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:

```
1 n! 0 <- 1
2 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.

2 Разработка языка

2.1 Критерии оценки языка

Для разрабатываемого языка выбраны следующие критерии оценки:

- читабельность (легкость чтения и понятность программы)
- лёгкость написания программ
- надёжность (обработка исключительных ситуаций)
- стоимость

2.2 Описание языка

Разрабатываемый язык должен быть удобен для написания коротких программ, скриптов, а также для первоначального обучения программированию. Поэтому сам язык должен быть максимально краток и понятен для разработчика. В языке должны отсутствовать лишние ключевые слова и управляющие конструкции. Для математических формул обязательна инфиксная нотация.

В разрабатываемом языке есть два вида стрелок. Стрелка влево (`<-`) означает, что фрагменту слева присвается выражение справа. Эта стрелка используется, например, для присваивания значения функции или простой переменной.

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 опреде-

ления 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.

Объявление функций состоит из имени функции, её аргументов и тела функции. Аргументами функции могут быть имена переменных, числа, списки и «продолжения». 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.

«Продолжения» в списке аргументов — это списки из 0 и более аргументов, которые используются, когда количество аргументов функции неизвестно.

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, возвра-

щающая 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.

```
1 filter pred? [] <- []
2 filter pred? [x : xs] <-
3     if   | pred? x -> [x] ++ filter pred? xs
4          |          -> 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 Scheme, Язык является регистрозависимым. 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)[11].

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`.

```
1  Digit    ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.
2  Hexd     ::= 'A'|'B'|'C'|'D'|'E'|'F'|'a'|'b'|'c'|'d'|'e'|'f
       '.
3  Hdigit   ::= Hexd|Digit
4  Dec      ::= ('+'|'-')? Digit+.
5  Hex      ::= '#x' Hdigit+.
6  Int      ::= Dec|Hex.
7  Fracdec  ::= Dec '/' Digit+.
8  Frachex  ::= Hex '/' Hdigit+.
9  Frac     ::= Fracdec|Frachex.
10 Expint   ::= Dec 'e' Dec.
11 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 нижележащим 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
sin, *cos*, *tg*, *ctg*	тригонометрические функции, вычисляющие синус, косинус, тангенс, 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
null?	проверяет, является ли аргумент пустым списком
reverse	переворачивает список
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
.	`f.x` – применить функцию `f` к списку аргументов `x`
:	для записи «продолжений» списков `[x : xs]`
\	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 присваивания $\leftarrow$ и тела функции. Аргументами функции при объявлении могут быть:

- идентификаторы, не являющиеся ключевыми словами (SimpleIdent);
- числа (Number);
- списки, не содержащие выражений (ListDeclaration);
- и «продолжение» 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.

```
1 FunctionDefinition     ::= FunctionDeclaration '<-' FunctionBody.
2 FunctionDeclaration    ::= SimpleIdent ArgumentDeclaration?.
3 ArgumentDeclaration    ::= SimpleArgument* ContinuousDeclaration?
4 SimpleArgument         ::= SimpleIdent|Number|ListDeclaration.
5 ListDeclaration        ::= '['ArgumentDeclaration?']'.
6 ContinuousDeclaration ::= ':' SimpleIdent|ListDeclaration.
```

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

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

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

2.3.3 Тело функции

Тело функции, также может содержать в себе определения функций. 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.

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

2.3.4 Выражение

Выражение (Expression) может быть

- условным выражением (IfExpression);
- безымянной функцией (LambdaFunction);
- математическим выражением (MathExpression).

```
1 Expression ::= IfExpression | LambdaFunction | MathExpression .
```

2.3.5 Условный оператор

Условный оператор if похож на оператор $cond$ в $Scheme$. Структура его аргументов устроена следующим образом:

1. 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;

2. 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
```

2.3.6 Математическое выражение

Математическое выражение 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 математическим выражениям;
- объединением математических выражений с помощью операторов и скобок.

```
MathExpression     ::= StringExpression | ListExpression | Number | Bool |
                       FunctionApply | FunctionCall |
                       (MathExpression BinoryOperator MathExpression) |
                       ('(' MathExpression ')') |
                       (UnaryOperator MathExpression).
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      ::= '+' | '-' | '!'.
```

2.3.7 Вставки на 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+.
```

2.3.8 Комментарии

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

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

2.3.9 Программа

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

```
1 Program ::= FunctionDefinition | MathExpression |
2             Scheme | Export | Comment.
```

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[11].

Текущий токен, хранится в глобальной для функций-правил переменной. Входная функции соответствует правилу $Program$.

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 случае корректного математического выражения 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, и её вызовов

```
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
```

```
18                  (#(func-decl (#(func-name #(tag-sym #(2 17) "n"))))
19                   #(tag-num #(2 21) 1)
20                   #(tag-mns #(2 19) "-")))))))
21          #(tag-mul #(2 11) "*")))))
22  #(expr
23    (#(func-decl
24       (#(func-name #(tag-sym #(4 1) "n!"))
25        #(argument (#(simple-argument #(tag-num #(4 4) 5))))))))
26  #(expr
27    (#(func-decl
28       (#(func-name #(tag-sym #(5 1) "n!"))
29        #(argument (#(simple-argument #(tag-num #(5 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[11]. Здесь символ — именованная сущность, определяемая парой `<name, info>`, где `name` — идентификатор сущности, а `info` — её описание.

Описание символа представляет собой список возможных значений этого символа. Одно такое значение 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. «Продолжения» 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 «продолжения» и нет. Для первых — это проверка на то, что аргументов не меньше, чем перечислено до «продолжения». 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!" #(#((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.

Сгенерированный код приписывается к базовым функциям.

Пример кода, сгенерированного на основе модели из предыдущего пункта:

```
(define (n! . :args)
  (:map-cond (((and ((lambda (:x) (= :x 1)) (length :args))
                    (:hash (quote ((lambda (x) (eqv? x 0)))) :args)
                    ((lambda :args #t) :args))
                (apply (lambda (:g_) (begin 1)) :args))
               ((and ((lambda (:x) (= :x 1)) (length :args))
                    (:hash (quote ((lambda (x) #t))) :args)
                    ((lambda :args #t) :args))
                (apply (lambda (n) (begin (* n (n! (- n 1)))))
                       :args)))))
(n! 5)
(n! 10)
```

4 Тестирование

4.1 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.

4.2 Тестирование корректности работы

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)`
	`  (quotient y 100))`
	` 7)))

5 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.

5.1 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.

5.1.1 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 обеспече-

ния на разработку ТЗ, [чел.-дни]. Их значения рассчитываются по формулам:

$$T_{RZ}^{Z} = t_Z * K_{RZ}^{Z},$$

$$T_{RP}^{Z} = t_Z * K_{RP}^{Z},$$

где t_Z – норма времени на разработку ТЗ 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):

$$t_Z = 37.$$

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:

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

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

$$T^E_{RZ} = t_E * K^E_{RZ},$$

$$T^E_{RP} = t_E * K^E_{RP},$$

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

$$K^E_{RZ} = 0.7.$$

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

$$K^E_{RP} = 0.3.$$

Тогда:

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

5.1.3 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:

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

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:

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

5.1.4 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)):

$$t^R_{RZ} = 8,$$

$$t^R_{RP} = 51.$$

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 — В, 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:

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

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:

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

5.1.5 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:

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

5.2 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 3. Из таблицы получаем, что фонд рабочего времени

$$F = 96.$$

Таблица 3: 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.

5.3 Ленточный график выполнения работ

На основе рассчитанных в главах 5.1, 5.2 трудоёмкости и фонда рабочего времени найдём количество рабочих дней, требуемых для выполнения каждого этапа разработка. Результаты приведены в таблице 4.

Таблица 4: Трудоёмкость выполнения работы над проектом

Номер стадии	**Название стадии**	**Трудоёмкость [чел.-дни]**	**Удельный вес [%]**	**Количество рабочих 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 4 в ленточный график (таблица 5), в ячейки столбца “продолжительности рабочих дней” заносятся времена, которые требуются на выполнение соответствующего этапа. Все исполнители работают одновременно.

5.4 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 6 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 5: 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 6: Затраты на зарплату и отчисления на социальное страхование

Должность	**Зарплата в месяц**	**Рабочих месяцев**	**Суммарная зарплата**	**Затраты на социальные нужд**
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 7.

Таблица 7: 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 K = 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.

5.5 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:

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

5.6 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 8 находится расчёт чистого дисконтированного дохода. График его изменения приведён на рисунке 1.

Таблица 8: 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 существующих систем и сможет их заменить. Итоговый ЧДД составил: 2019802 рубля.

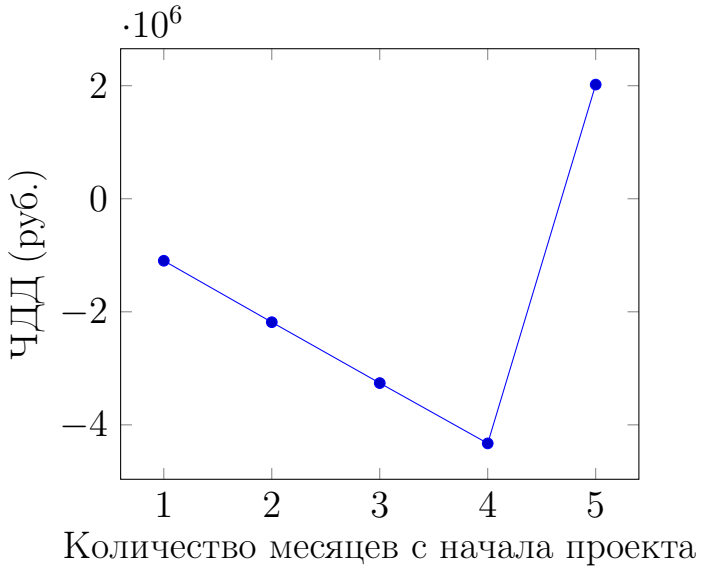

Рисунок 1 — График изменения чистого дисконтированного дохода

5.7 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.

6 Примеры кода

```
; 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

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]

; count x in xs
count x []         <- 0
count x [ x : xs ] <- 1 + count x xs
count x [ y : xs ] <- count x xsповторные
; переменные в образцах

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

```
34
35 fact 0 <- 1
36 fact n <- n * fact (n - 1)
37
38 fact 5
39 fact 10
40
41 sum <- 0
42 sum x : xs <- x + sum . xs
43
44 sum 1 2 3 4
45 sum.[5 6 7 8 9 10]
46
47 replace pred? proc []         <- []
48 replace pred? proc [ x : xs ] <-
49     if | pred? x -> [ proc.[x] : replace pred? proc xs ]
50        |         -> [ x : replace pred? proc xs ]
51
52 replace zero?
53         \ x -> x + 1
54         [ 0 1 2 3 0 ]
55
56 replace odd?
57         \ x -> x * 2
58         [ 1 2 3 4 5 6 ]
59
60 replace \ x -> 0 > x
61         exp
62         [ 0 1 -1 2 -2 3 -3]
63
64 replicate x 0 <- []
65 replicate x n <- [ x : replicate x ( n - 1 ) ]
66
67 replicate "a" 5
68 replicate [ "a" "b" ] 3
```

```
69 replicate "a" 0
70
71 cycle xs 0 <- []
72 cycle xs n <-
73     xs ++ cycle xs ( n - 1 )
74
75 cycle [ 0 1 ] 3
76 cycle [ 'a 'b 'c ] 5
77 cycle [] 0
78
79 and-fold         <- #t
80 and-fold x : xs <- x && and-fold : xs
81
82 and-fold #f #f #f
83 and-fold #f #f #t
84 and-fold #f #t #t
85 and-fold #t #t #t
86 and-fold
87
88 0? 0 <- #t
89 0? x <- #f
90
91 nil? 0  <- #t
92 nil? [] <- #t
93 nil? x  <- #f
94
95 %2=0? x <- x % 2 = 0
96 %2!0? x <- ! %2=0? x
97
98 0? 0
99 0? 1
100
101 nil? 0
102 nil? 1
103 nil? []
```

```
104 nil? [1 2 3]
105
106 %2=0? 4
107 %2=0? 5
108
109 %2!0? 1
110 %2!0? 2
111
112 scheme (define (selection-sort pred? xs)
113          (define (min-xs xs x)
114            (cond ((null? xs)         x)
115                  ((pred? (car xs) x) (min-xs (cdr xs) (car xs)))
116                  (else               (min-xs (cdr xs) x))))
117
118          (define (swap j xs)
119            (let ((xj (list-ref xs j))
120                  (vs (list->vector xs)))
121              (vector-set! vs j (car xs))
122              (vector-set! vs 0 xj)
123              (vector->list vs)))
124
125          (define (ind x xs)
126            (- (length xs) (length (member x xs))))
127
128          (define (helper xs)
129            (if (null? xs)
130                '()
131                (let ((x (min-xs xs (car xs))))
132                  (cons x (helper (cdr (swap (ind x xs) xs)))))))
133
134          (helper xs))
135 scheme (define (insertion-sort pred? xs)
136          (define (insert xs ys x)
137            (cond ((null? ys) (append xs (list x)))
138                  ((pred? (car ys) x) (insert (append xs (list (car
```

```
    ys))) (cdr ys) x))
139                 (else                 (append xs (list x) ys))))
140
141         (define (helper xs ys)
142           (if (null? ys)
143               xs
144               (helper (insert '() xs (car ys)) (cdr ys))))
145
146         (helper '() xs))
147
148 export selection-sort insertion-sort
149
150 selection-sort \ x y -> x <= y
151                [9 6 2 4 3 5 7 1 8 0]
152 insertion-sort \ x y -> x <= y
153                [9 6 2 4 3 5 7 1 8 0]
154
155 str <- "long string"
156 replicate str 8
157
158 my-gcd a b <-
159     r <- a % b
160
161     if  | a < b -> my-gcd b a
162         | 0? r  -> b
163         |       -> my-gcd b r
164
165 my-lcm a b <-
166     abs (a * b / my-gcd a b)
167
168 prime? n <-
169     n! 0 <- 1
170     n! n <- n * n! (n - 1)
171
172     memo n!
```

```
173
174     0?.[ (n!.[n - 1] + 1) % n ]
175
176 my-gcd 3542 2464
177 my-lcm 3 4
178 prime? 11
179 prime? 12
180
181 bisection f a b e <-
182     sign 0 <- 0
183     sign x <- if | x > 0 -> 1
184                  |       -> -1
185
186     mid a b <-
187         x <- a + (b - a) / 2
188
189         if   | (abs f x) <= e          -> x
190              | (sign f.[b]) = (sign f.[x]) -> mid a x
191              |                          -> mid x b
192
193     if | f.[a] = 0 -> a
194        | f.[b] = 0 -> b
195        |         -> mid a b
196
197 bisection cos -3.0 0.0 0.001
198
199 newton f df x e <-
200     if   | (abs f x) < e -> x
201          |             -> newton f df (x - f.[x] / df.[x]) e
202
203 golden f x0 x1 e <-
204     fi <- ((sqrt 5) + 1) / 2
205
206     loop f x0 x1 e <-
207         a <- x1 - (x1 - x0) / fi
```

```
208         b <- x0 + (x1 - x0) / fi
209
210         if  | f.[a] >= f.[b] ->
211                 if  | (abs x1 - a) < e -> (a + x1) / 2
212                     |                   -> loop f a x1 e
213             | ->
214                 if  | (abs b - x0) < e -> (x0 + b) / 2
215                     |                   -> loop f x0 b e
216
217     loop f x0 x1 e
218
219 round newton \ x -> x ** 2
220               \ x -> 2 * x
221               1.0
222               1e-8
223
224 round newton \ x -> x ** 2 + 4 * x + 4
225               \ x -> 2 * x + 4
226               5.0
227               1e-8
228
229 round golden \ x -> x ** 2
230               -2.0
231               2.0
232               1e-08
233
234 round golden \ x -> x ** 2 + 4 * x + 4
235               -5.0
236               5.0
237               1e-06
238
239 5 + 6
240
241 ! 8
242
```

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

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

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

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

range 0 11 3
flatten [[1] 2 [3 [4 5] [6 [7 8]]] 9]
element? 1 [3 2 1]
element? 4 [3 2 1]
filter odd?
       range.[0 10 1]
filter \ x -> x % 3 = 0
       range.[0 13 1]

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

memo fibonacci fact

fibonacci 99
fibonacci 100
```

Список литературы

[1] http://www.tiobe.com/tiobe_index

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

[3] http://www.schemers.org/Documents/Standards/R5RS/HTML/

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

[5] Information technology — Programming languages — Prolog. 1995

[6] http://www.nsl.com/misc/papers/martelli-montanari.pdf

[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, http://python.swaroopch.com/

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

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

[11] Скоробогатов С.Ю. Лекции по курсу «Компиляторы», 2014.

[12] 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.

[13] Под ред. Смирнова С.В. Организационно-экономическая часть дипломных проектов исследовательского профиля. М.: изд. МГТУ им. Баумана, 1995. 100 с.