Приднестровский государственный университет им. Т. Г. Шевченко

Физико-математический факультет

Кафедра прикладной математики и информатики

Реферат

по дисциплине «Спецсеминар»

ЯЗЫК SCHEME

Выполнил:

студент гр. 403 д/о

физ.-мат. факультета

Стефанишин Никита Алексеевич

Руководитель:

Ст. преподаватель

Кафедры информатики и ВТ

Великодный Вадим Игоревич

Тирасполь, 2017.

**Содержание**

Аннотация . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . 3

Введение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . 4

1. Что такое Scheme ? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.1 Семантика Scheme . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2 Синтаксис Scheme . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.1 Алфавит и соглашения Scheme . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . 10

2.2 Основные конструкции Scheme . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . 12

2.3 Управляющие конструкции Scheme . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.4 Рекурсии . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.5 Ввод/вывод в Scheme . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

Заключение . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

Список литературы . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

Ссылки . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**Аннотация**

В данном реферате рассматриваются основные понятия и конструкции языка функционального программирования Scheme, который является одним из диалектов системы программирования LISP. В отличие от большинства языков программирования, требующих формализации задачи, Scheme предоставляет программисту создавать среду программирования наиболее соответствующую решаемой задаче.

**Введение**

Scheme – это функциональный язык программирования, разработанный Гаем Стилом и Джеральдом Сассменом из Массачусетского технологического института. Создан в середине 1970-х годов. При разработке Scheme упор был сделан на элегантность и простоту языка. Философия языка подчёркнуто минималистская. В результате, Scheme содержит минимум примитивных конструкций и позволяет выразить все, что угодно путём надстройки над ними. Основой построения языка является

λ-исчисление.

Scheme начинался с попытки понять модель акторов  Карла Хьюитта, для чего Стил и Сассман написали «крошечный интерпретатор Лиспа», а затем «добавили механизм создания акторов и посылки сообщений». Scheme был первым диалектом Лиспа, применяющим исключительно статические области видимости переменных, гарантирующим оптимизацию хвостовой рекурсии и поддерживающим данные булевского типа. Scheme был первым функциональным языком, непосредственно поддерживающим механизм продолжений.

В современных спецификациях языка реализовано мощное и удобное средство для записи макросов на основе шаблонов синтаксического преобразования с «соблюдением гигиены» (анг. hygienic macro), «сборка мусора», то есть автоматическое освобождение памяти от неиспользуемых более объектов.

Базовые структуры данных языка ограничены списками и одномерными массивами («векторами»). В соответствии с декларируемым минимализмом, нет стандартного синтаксиса для поддержки многомерных структур, структур с именованными полями, а также средств объектно —ориентированного программирования — все это может быть реализовано программистом по его предпочтению.

Для того, чтобы сохранить ядро ​​языка малым и способствовать стандартизации расширений, в сообществе Scheme принят процесс "Scheme Request for Implementation" (запрос на реализацию, SRFI), с помощью которого предлагаемые расширения проходят тщательное обсуждение. Это способствует переносимости кода. Многие SRFI поддерживаются всеми или большинством реализаций Scheme.

1. **Что такое Scheme?**

Scheme – компактный, исключительно «чистый» язык. Стандарт языка занимает всего около 50(!) страниц, включая формальное определение семантики. Scheme разрабатывался таким образом, чтобы малое число универсальных конструкций можно было легко использовать в разных стилях программирования: функциональном, объектно-ориентированном и императивном.

Программирование в Scheme предполагает не формализацию решаемой задачи под выбранный язык, а создание среды программирования, наиболее полно описывающей решаемую задачу. Такой метод называется программирование «снизу-вверх» и является достаточно непривычным для большинства программистов.

Основанный на формальной модели λ-исчисления язык Scheme особенно удобен для разработки таких приложений как текстовые редакторы, операционные системы, оптимизирующие компиляторы, экспертные и финансово-аналитические системы, системы виртуальной реальности и многих научно-экспериментальных приложений в области теории и практики.

Современные реализации Scheme имеют возможности итерактивного исполнения команд, интерпретации и компиляции программного кода.

На сегодняшний день Scheme является одним из трех основных диалектов системы программирования LISP.

**1.1. Семантика Scheme.**

Формальная семантика Scheme основана на λ-исчислении, это означает что функции могут быть определены в месте их использования и при этом иметь доступ к локальным переменным родительской функции. Подобный механизм позволяет создавать надстройки над базовыми понятиями языка.

Объекты Scheme (функции, переменные, константы и т.д.) по умолчанию лексически связываются со значением, полученным при вычислении. Такое связывание называют «скрытыми типами», а язык, использующий подобный механизм, слабо типизированным. Исключением является прямое определение с использованием предикатов типа.

Объекты Scheme, созданные в ходе вычислений, хранятся неограниченное время. Но если процесс может подтвердить что данный объект не будет использован для дальнейших вычислений, его область хранения освобождается (так называемая «сборка мусора»). Это приводит к тому, что программы на Scheme практически избегают переполнения области хранения данных и стеков программы. Подобными особенностями обладает ограниченное количество языков программирования (APL, Common LISP, Erlang & etc.)

Динамическая типизация языка Scheme позволяет реализовать рекурсивные вычисления с правильной типизацией вызовов. Это упрощает итерационные вычисления, даже если они описываются рекурсивной процедурой. Именно это свойство позволяет обойтись в Scheme только двумя способами организации циклических вычислений:

- хвостовая рекурсия;

- простая итерация.

Прочие циклические процедуры в Scheme теряют смысл.

Все объекты Scheme самостоятельны и могут создаваться динамически, возвращаться как результат вычисления функции, сохраняться как элементы данных. (подобными возможностями обладают Common Lisp и ML).

Scheme является первым языком программирования в котором был реализован механизм реализации «продолжений», т.е. приостановки и продолжения выполнения вычислений с определенного места с сохранением текущего состояния окружения. Подобный механизм в настоящее время весьма распространен. Механизм «продолжений» оказывается весьма удобным для реализации широкого круга управляющих функций, нелокальных выходов, возвратов, подпрограмм и т.п.

Аргументу функций Scheme всегда передаются по значению, то есть фактическое значение аргумента вычисляется до момента выполнения процедуры, что дает возможность оценки необходимости выполнения.

Арифметическая модель Scheme построена таким образом, что каждое целое число является рациональным, рациональное – вещественным, вещественное – комплексным. Следовательно, различие между целочисленной и вещественной арифметикой сведено на нет. Однако существует различие между точной (идеальной) арифметикой и арифметической аппроксимацией, которые не ограничиваются целыми числами.

1. **Синтаксис Scheme.**

**2.1 Алфавит и соглашения Scheme.**

В языке Scheme используются ключевые слова, идентификаторы, переменные, строки, одномерные векторы, символы, числа, коментарии.

Набор символов соответствует набору символов Unicode и включает в себя буквы **A-Z**,**a-z**, цифры **0-9**, специальные символы **?**, **!**, **.**, **+**, **-**, **\***, **/**, **<**, **=**, **>**, **:**, **$**, **%**, **^**, **&**, **\_**, **~**, **@** и **#**.

Идентификаторы и имена переменных не могут начинаться с символов **@** и символов с которых могут начинаться числа (цифры и знаки **+**, **-**, **.**). Идентификаторы ограничиваются пробелами и могут содержать внутри конструкцию **->**.

Символы предворяются одиночной кавычкой **‘**, а строки заключаются в двойные кавычки **“ ”**.

Комментарии начинаются с символа **;** и продолжаются до конца строки. В комментарии можно использовать любые печатаемые символы.

Числа в Scheme могут представляться как целые **(123, -123)**, в виде дроби **( 4/9)**, с плавающей точкой **(123,45)**, в экспоненциальном виде **(1,2345е2)** или в виде комплексного числа в обычном и полярном виде **(1.3-2.7i или** [**-1.2@73**](mailto:-1.2@73)**)**. Как уже отмечалось выше, выбор синтаксиса числа неважен для выполнения вычислений.

Логические значения, которые интерпретируются как true и false записываются **#t** и **#f**. При этом Scheme определяет как «ложь» только #f, все остальные объекты языка интерпретируются как «истина».

Списки и структурированные формы заключаются в круглые скобки, например **(a b c)**, **(+ 5 (\* x 2) 15)**. Для удобства чтения отдельных синтаксических форм допускается использование квадратных скобок **[ ]**.

Вектор массивы записываются также как списки, но предваряются символом # - **#(this is a vector of symbols)**.

Байт векторы записываются без знаковыми числами (от 0 до 255) и предваряются конструкцией #vu8 - **#vu8(15 205 122 15)**.

В языке Scheme принят ряд соглашений относительно имен функций.

* предикаты типов завершаются знаком **?** (**pair?**);
* предикаты сравнения, возвращающие логическое значение, оканчиваются знаком **?** (**eq?**, **zero?**, и **string=?**), числовые операции сравнения **=**, **<**, **>**, **<=**, и **>=** не попадают под это соглашение;
* имена большинства символьных, строковых и векторных процедур начинаются с префиксов **char-**, **string-**, и **vector-**;
* имена процедур преобразования типов записываются следующим образом **type1->type2**, например **vector->list**.

**2.2 Основные конструкции Scheme.**

Основной конструкцией языка Scheme является функция. Тело функции обрамляется круглыми скобками **( )**. В общем виде функцию можно записать так:

**(<имя> арг1 арг2 … аргn)**

в качестве аргументов в функцию могут передаваться любые объекты Scheme, в качестве разделителя используется пробел. Следует отметить, что в Scheme простые арифметические операции реализованы как функции, т.е. имеют предикатную запись (именем функции является знак операции):

**(+ 5 7)**

> 12

**(- 15 (\* 3 2))**

> 9

Определение функции должно соответствовать следующему прототипу:

**(define** имя-функции **(lambda (**аргументы**) (**тело функции**)))**,

на практике чаще используют сокращенную запись

**(define (**имя-функции аргументы**) (**тело функции**))**

Пример:

; Функция вычисления квадрата числа

(define square

(lambda (n) (\* n n)))

(square 5)

> 25

(+ 5 (square 7))

> 54

Для присвоения начальных значений переменным используется конструкция **(let ((**имя выр**)** ...**)** тело1 тело2 ...**)**

(let ((x 2) (y (square x)) (+ x y))

>6

Использование ранее описанной функции в теле описания другой функции (как в приведенном примере) называется подстановочной моделью использования функции.

**2.3 Управляющие конструкции Scheme.**

В языке Scheme используются управляющие конструкции. Одной из них является условная функция ее общий вид

**(if** *тест действие альтернатива***)** или

**(if** *тест действие***)**

где *тест* – проверочное выражение, возвращающее логическое значение;

действие – функция, выполняемая, если тест #t

альтернатива – функция, выполняемая, если тест #f.

(let ([ls '(a b c)]) ; определение списка

(if (null? ls) ; проверка наполненности списка

'() ; если список пустой вернуть пустой список

(cdr ls)) ; иначе вернуть все элементы списка кроме первого

> (b c)

(let ([abs ; функция нахождения абсолютного значения числа

(lambda (x)

(if (< x 0) ; если x<0

(- 0 x) ; конвертируем х

x))]) ; иначе возвращаем значение х

(abs -6)

> 6

Другой управляющей конструкцией, которую рассмотрим, является функция итерационного цикла

**(do ((**пер старт изм**)** ...**) (**тест ...**)** выр ...**)**

где пер – переменная параметр итерации

старт – начальное значение переменной

изм – шаговое изменение переменной

тест – проверка конечного условия

выр – функция, выполняемая при каждом шаге итерации.

В качестве примеров приведем функции вычисления факториала и числа Фибоначчи по порядковому номеру

(define factorial

(lambda (n)

(do ([i n (- i 1)] [a1 (\* a i)]) ; выполнять умножение пока i=n

((zero? i) a)))) ; не станет равно 0

(factorial 10)

> 3628800

(define fibonacci

(lambda (n)

(if (= n 0) ; если n=0 число равно 0

0

(do ([i n (- i 1)] [a1 1 (+ a1 a2)] [a2 0 a1]) ;иначе выполнять для i от n

((= i 1) a1))))) ; с вычитанием 1 до i=1 сложение текущего числа

; с предыдущим

(fibonacci 6)

>8

**2.4 Рекурсии.**

Рекурсия — мощный метод обработки иерархических, древовидных объектов. Рекурсивная функция представляет собой процедуру, в которой в качестве одного из параметров передается вызов самой функции. В силу специфики вычислительного процесса использование рекурсии во многих языках программирования запрещено и вызывает ошибку. В других языках, таких как Erlang, LISP, Scheme реализован механизм хвостовой рекурсии.

Хвостовая рекурсия предусматривает размещение рекурсивного вызова в конце описания функции. В этом случае за счет оптимизации кода не используется стек вызовов.

Использование хвостовой рекурсии позволяет организовать циклические вычисления без применения специальных операторов.

Примеры определения рекурсивных функций:

;; факториал в рекурсивном стиле

(define (fact x)

(if (< x 2)

1

(\* (fact (- x 1)) x)))

;; функция Фибоначчи — требует параллельной рекурсии

(define (fib n)

(if ((= n 0) 0)

((= n 1) 1)

(else (+ (fib (- n 1))

(fib (- n 2))))))

**2.5 Ввод/вывод в Scheme.**

Для ввода и вывода в Scheme используется тип порт . Scheme определяет два стандартных порта, доступные как ***current-input-port*** и ***current-output-port***, отвечающие [стандартным потокам ввода-вывода](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D1%8B%D0%B5_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B8) Unix. Большинство реализаций также предоставляют current-error-port. [Перенаправление ввода-вывода](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BD%D0%B0%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B2%D0%B2%D0%BE%D0%B4%D0%B0-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4%D0%B0) поддерживается в стандарте с помощью процедур ***with-input-from-file*** и ***with-output-to-file***. У реализаций также имеются строковые порты, с помощью которых многие операции ввода-вывода могут выполняться со строковым буфером вместо файла, используя процедуры из SRFI 6.

*(write (+ (read) (read)))*

Вывод в порт по умолчанию (current-output-port):

*(let ((hello0 (lambda() (display "Hello world") (newline)))) (hello0))*

Передача порта в качестве аргумента:

*(let ((hello1 (lambda (p) (display "Hello world" p) (newline p))))*

*(hello1 (current-output-port)))*

Перенаправление вывода в файл:

*(let ((hello0 (lambda () (display "Hello world") (newline))))*

*(with-output-to-file "outputfile" hello0))*

Явное открытие файла и закрытие порта:

*(let ((hello1 (lambda (p) (display "Hello world" p) (newline p)))*

*(output-port (open-output-file "outputfile")))*

*(hello1 output-port)*

*(close-output-port output-port)*

*)*

*call-with-output-file:*

*(let ((hello1 (lambda (p) (display "Hello world" p) (newline p))))*

*(call-with-output-file "outputfile" hello1))*

Подобные процедуры есть и для ввода. Scheme предоставляет предикаты **input-port?** и **output-port?**. Для символьного ввода и вывода существуют **write-char**, **read-char**, **peek-char**и **char-ready?**. Для чтения и записи выражений Scheme используются процедуры **read** и **write**. Если порт достиг конца файла при операции чтения, возвращается eof-объект, который может быть распознан предикатом **eof-object?**.

**Заключение**

Язык функционального программирования Scheme за счет своего минимализма, возможности использования составных функций и подстановочной модели использования функций обеспечивает гибкость программирования и позволяет построить среду описания задачи, наиболее соответствующую ее пониманию. В то же время следует отметить необычность заложенного в языке подхода к программированию и, как следствие, сложность освоения языка.

**Список литературы**

1. R. Kent Dybvig / The Scheme Programming Language, Fourth Edition

Copyright © 2009 The MIT Press. Electronically reproduced by permission.

Illustrations © 2009 Jean-Pierre Hébert

**Ссылки**

<http://o-ili-v.ru/wiki/Scheme>

<http://www.scheme.com/tspl4/>

<http://ilammy.github.io/lisp/>

<https://github.com/ilammy/lisp>

<http://5top100.ru/upload/iblock/85f/85f3f644ead9ca3fb0e76057242926ea.pdf>