МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Н. Э. БАУМАНА

Факультет информатики и систем управления

Кафедра теоретической информатики и компьютерных технологий

Курсовой проект

по курсу «Конструирование компиляторов»

«Препроцессор синаксического сахара для языка Scheme»

Выполнил:
студент ИУ9-101
Выборнов А. И.

Руководитель:
Дубанов А. В.

Москва 2015

Содержание

Введение

Lisp — это семейство динамических функциональных языков программирования. Первая версия языка Lisp была создана в 1958 году в ходе работ по созданию искусственного интеллекта. К настоящему времени сфера применения Lisp значительно увеличилась, а Lisp представляет собой целое семейство языков: Common Lisp, Scheme, Racket и другие.

В рамках работы рассматривается язык Scheme. Этот язык был разработан специально для учебных целей, благодаря чему включает в себя очень ограниченный, но весьма гибкий набор примитивов. Scheme удобен для написания скриптов и расширений (имеется специально для этого предназначенная реализация — GNU Guile).

Форматирование кода на Scheme, отслеживание парных открывающих и закрывающих скобок и некоторые другие синтаксические особенности требуют применения не слишком распространенных и привычных приложений таких как специализированная среда разработки, к примеру — Racket или текстовый редактор со специальными расширениями, наиболее часто используется Emacs. К числу недостатков можно также отнести: обилие скобок, затрудняющее чтение программы, отсутствие возможности записи выражений в привычном инфиксном формате.

В последнее время отмечается значительный рост интереса к практическому применению функционального программирования, а таже удобству чтения и написания кода. Это привело к созданию большого числа языков программирования, в той или иной степени использующих синтаксис и семантику языка ML (Meta Language). В реализациях этих языков основной упор сделан на типобезопасность. Языки являются компилируемыми, со строгой статической типизацией.

Целью данной работы является разработка и реализация функционального динамического языка на основе Scheme, который имеет более дружелюбный синтаксис.

На разработку входного языка сильное влияние оказал Haskell [3] — представитель семейства ML подобных языков, который много унаследовал от Scheme: управляющие конструкции, связывание имён со значениями и т.д. В качестве целевого языка используется язык Scheme, удолетворяющий стандарту r5rs [2].

Компилятор входного языка в целевой реализован как препроцессор к языку Scheme.

Входной язык и компилятор получили название lactose (лактоза — молочный сахар). Названия языка это отсылка как к синтаксическому сахару, так и первоначальному обучению программированию.

1. Теоретическая часть

1.1. Scheme

Scheme — функциональный язык программирования. При разработке Scheme был сделан упор на элегантность и простоту языка. В результате, Scheme содержит минимум примитивных конструкций и позволяет выразить всё, что угодно путём надстройки над ними.

Racket — мультипарадигменный язык программирования общего назначения, принадлежащий семейству Lisp/Scheme. Одно из предназначений Racket — создание, разработка и реализация языков программирования. Racket это не просто язык программирования, а целая платформа, которая предоставляет реализацию языка Racket, развитую среду выполнения, различные библиотеки, JIT-компилятор, а также среду разработки DrRacket (ранее известную, как DrScheme) написанную на Racket.

Racket поддерживает исполнение программ написанных согласно стандарту Scheme r5rs.

Результатом работы компилятора является файл Racket, содержащий в себе программу на языке Scheme.

1.2. Входной язык

Программа на входном языке представляет собой множество определений функций и операторов, которые позволяют импортировать символы из Scheme. Если в корне программы определена функция main, то она будет являться точкой входа.

1.2.1. Токены

Токены языка аналогичны токенам языка программирования Scheme. Представлены следующие виды токенов: строка, идентификатор, число, логический тип и символ (литеральная константа). Примеры правильных токенов:

- символ закрывающей скобки — #\)
- строка — "\"r5rs\" standart\n"
- идентификатор — _ident
- логическая истина — #t
- десятичное число — 123
- шестнадцатеричное число — #xDEADBEAF

В рамках грамматики под нетерминалом $\langle token \rangle$ подразумевают строку, число, логический тип или символ:
$\langle token \rangle$ ::= $\langle BOOLEAN \rangle$ | $\langle NUMBER \rangle$ | $\langle CHARACTER \rangle$ | $\langle STRING \rangle$

Терминал, соответствующий идентификатору — $\langle IDENTIFIER \rangle$, рассматривается отдельно.

Единственным существенным отличием токенов входного языка от токенов Scheme является запрет на использование в токенах символов, задающих арифметические операции, а именно: ‘-’, ‘*’, ‘/’, ‘<’, ‘>’, ‘=’, ‘!’, ‘%’, ‘&’, ‘ ’. Как и в Scheme токены регистронезависимы.

Документация Scheme [2] содержит следующее определение строки в РБНФ:
$\langle string \rangle$ ::= ‘"’ $\langle string\ element \rangle$* ‘"’

$\langle string\ element \rangle$ ::= любой символ кроме ‘"’ или ‘\’ | ‘\"’ | ‘\\’

Данное определение содержит ошибку. Оно не позволяет определить одиночный слэш. Поэтому использовалось определение строки, совпадающее с таковым в реализации стандарта r5rs в рамках языка Racket:
$\langle string \rangle$ ::= ‘"’ $\langle string\ element \rangle$* ‘"’

$\langle string\ element \rangle$ ::= любой символ кроме ‘"’ или ‘\’ | ‘\"’ | ‘\’

1.2.2. Выражения

Основной структурной единицей языка является выражение. В качестве выражения рассматриваются унарные и бинарные инфиксные операции, условный оператор, вызов функции, анонимные функции, списки. Входной язык поддерживает инфиксные операции, аналогичные операциям языка Python. Порядок вычислений определяется приоритетом операторов, а также может быть задан явно с помощью круглых скобок.

Выражение задаётся нетерминалом ⟨*expression*⟩. Грамматика выражений в формате РБНФ:

⟨*expression*⟩ ::= ⟨*arithmetic expression*⟩
| ⟨*lambda function*⟩
| ⟨*list expression*⟩

⟨*arithmetic expression*⟩ ::= ⟨*arithmetic expression*⟩ '`**`' ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ ('`*`'|'`/`'|'`%`'|'`//`') ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ ('`+`'|'`-`') ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ ('`<<`' | '`>>`') ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ '`&`' ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ '`^`' ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ '`|`' ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ '`and`' ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ '`or`' ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ ('`<`' | '`<=`' | '`>`' | '`>=`') ⟨*arithmetic expression*⟩
| ⟨*arithmetic expression*⟩ ('`==`' | '`!=`') ⟨*arithmetic expression*⟩
| ('`+`'|'`-`') ⟨*arithmetic expression*⟩
| ('`not`'|'`~`') ⟨*arithmetic expression*⟩
| ⟨*if condition*⟩
| ⟨*token*⟩
| ⟨*IDENTIFIER*⟩
| ⟨*function call*⟩
| ⟨*lambda function call*⟩
| '`(`' ⟨*arithmetic expression*⟩ '`)`'

Приоритеты заданы в РБНФ следующим образом: если тело правила состоит из нескольких частей, разделённых знаком '|', то чем левее часть, тем у неё выше приоритет.

Пример выражения: not (x $>$ 2 and x $<$ 5).

1.2.3. Условия

В качестве условного оператора используется традиционный оператор if. В грамматике условный оператор задаётся с помощью нетерминала $\langle if\ condition \rangle$. Условный оператор в формате РБНФ:

$\langle$*if condition*$\rangle$ ::= ‘`if`’ $\langle$*expression*$\rangle$ ‘`then`’ $\langle$*expression*$\rangle$ ‘`else`’ $\langle$*expression*$\rangle$

Условный оператор работает аналогично тернарному оператору: если выражение после ‘if’ истинно, то он возвращает выражение после ‘then’, в противном случае он возвращает выражение после ‘else’.

Пример условия: if not (x $>$ 2 and x $<$ 5)then 1 else 0.

1.2.4. Определение функций

Определение функции начинается с ключевого слова ‘def’. Функция задаётся идентификатором, соответствующим имени функции, списком идентификаторов, которые соответствуют аргументам функции и телом функции. Тело функции состоит из последовательности определений функций и выражений, которые, при вызове функции, последовательно исполняются. Для устранения неоднозначности тело функции можно обернуть в фигурные скобки. Функция возвращает результат последнего выражения из тела функции.

В грамматике определение функции соответствует нетерминалу $\langle function\ define \rangle$. Синтаксис определения функции в РБНФ:

$\langle$*function define*$\rangle$ ::= ‘`def`’ $\langle$*IDENTIFIER*$\rangle$ $\langle$*function arguments*$\rangle$ ‘`=`’ ($\langle$*function body*$\rangle$ | ‘`{`’ $\langle$*function body*$\rangle$ ‘`}`’)

$\langle$*function body*$\rangle$ ::= $\langle$*function_body_token*$\rangle$ (‘`;`’ $\langle$*function_body_token*$\rangle$)*

$\langle$*function body token*$\rangle$ ::= $\langle$*function define*$\rangle$ | $\langle$*expression*$\rangle$;

$\langle$*function arguments*$\rangle$ ::= $\langle$*IDENTIFIER*$\rangle$*

Пример определения функции abs от одного аргумента, которая возвращает модуль числа: def abs a = if a $<$ 0 then $-$a else a.

1.2.5. Локальные определения функций

Входной язык поддерживает вложенные объявления функций. Определение вложенной функции записывается точно также как и определение функции.

Пример функции fact рассчитывающей факториал числа, которая содержит в себе вложенную функцию loop:

```
def fact n =
    def loop i = {if i < n then i*loop(i+1) else n};
    loop 1
```

Из примера видно, что входной язык поддерживает замыкания (функция loop использует переменную n, которая объявлена вне её контекста).

1.2.6. Анонимные функции

Грамматика языка поддерживает анонимные функции (λ-функции). Анонимные функции особый вид функций, которые объявляются в месте использования и не получают уникального идентификатора для доступа к ним. Анонимные функции можно либо вызвать напрямую при создании, либо присвоить ссылку на неё в переменную (в контексте входного языка в функцию), для косвенного вызова в дальнейшем.

Анонимной функции соответствует нетерминал $\langle lambda\ function \rangle$. Синтаксис анонимной функции в РБНФ:

$\langle lambda\ function \rangle$ ::= '\' $\langle function\ arguments \rangle$ '->' $\langle function\ body \rangle$

Пример анонимной функции от двух аргументов, которая находит сумму квадратов двух чисел: \x y−> x∗x + y∗y. Анонимные функции также поддерживают замыкания. Пример определения функции, которая возращает анонимную функцию: def make_adder n = \x −> x+n.

1.2.7. Вызов функций

Вызов функции представляет собой имя функции и список выражений, соответствующих аргументам функции. При вызове анонимной функции имя заменяется на определение анонимной функции, заключённое в круглые скобки.

В грамматике вызовы функций задаются нетерминалами $\langle function\ call \rangle$ и $\langle lambda\ function\ call \rangle$, для обычных и анонимных функций соответственно. Синтаксис вызовов функций в РБНФ:

⟨*function call*⟩ ::= ⟨*IDENTIFIER*⟩ ⟨*expression*⟩*

⟨*lambda function call*⟩ ::= '(' ⟨*lambda function*⟩ ')' ⟨*expression*⟩*

Пример определения рекурсивной функции fact от одного аргумента, которая находит факториал числа: def fact n = if n == 0 then 1 else n*(fact n-1). Пример содержит вызов функции fact.

Пример вызова анонимной функции от одного аргумента, который также является анонимной функцией: (\f -> f 3) \x->x**2.

1.2.8. Списки

Во входном языке список это набор выражений, который обрамляется квадратными скобками. Список напрямую отображается в список языка Scheme. Для работы со списками определён набор функций (в круглых скобках указывает аналог из языка Scheme, при наличии):

- len (length) — принимает на вход список и возвращает длину списка.
- concat (append) — принимает на вход произвольное количество списков и возвращает список, представляющий собой конкатенацию переданных списков.
- ref (list-ref) — принимает на вход список (lst) и число (pos), возвращает элемент списка lst под номером pos (нумерация элементов списка начинается, как и в Scheme с нуля).
- head — принимает на вход список (lst) и число (pos), возвращает список, состоящий из элементов lst с номером меньшим pos в том же порядке.
- tail (list-tail) — принимает на вход список (lst) и число (pos), возвращает список, состоящий из элементов lst с номером большим либо равным pos в том же порядке.

В грамматике список задаётся нетерминалом ⟨*list expression*⟩. Синтаксис списка в РБНФ:

⟨*list expression*⟩ ::= '[' expression* ']'

Пример списка, содержащего два числа: [1 2+3].

1.2.9. Комментарии

Комментарии представляют собой часть строки, которая начинается с символа '–' и заканчивается переводом строки. Комментарии препроцессором полностью игнорируются.

Синтаксис нетерминала ⟨*comment*⟩, соответствующего комментарию, в формате РБНФ:

⟨*comment*⟩ ::= '`--`' (любой символ кроме '`\n`')*;

Пример комментария: −− commented text\n

1.2.10. Импортирование символов из Scheme

Входной язык поддерживает импорт символов функций из Scheme. Для это необходимо написать произвольный код на Scheme, обрамлённый с помощью двойных фигурных скобок, и, после ключевого слова 'export', описать импортированную функцию.

Под описанием импортированной функции подразумевается идентификатор, задающий имя функции и список аргументов. В качестве списка аргументов рекомендуется либо передать единственный аргумент '_', если функция принимает какие-либо аргументы на вход, либо ничего не передавать в противном случае.

Импорт символов из Scheme в грамматике описывается нетерминалом ⟨*scheme block*⟩. Синтаксис импорта символов в формате РБНФ:

⟨*scheme block*⟩ ::= ⟨*SCHEME BODY*⟩ '`export`' ⟨*IDENTIFIER*⟩ ⟨*function arguments*⟩

⟨*SCHEME BODY*⟩ ::= '`{{`' (любой символ кроме '`}`')* '`}}`'

Пример определения функции, печатающей квадрат переданного числа в консоль, на языке Scheme и экспорта символа этой функции в lactose: {{(define (print_sqr x) (display (expt x 2)))}} export print_sqr _. Пример импортирования функции car из стандартной библиотеки Scheme в lactose: {{}} export car _.

2. Реализация

Компиляция входного языка состоит из трёх основных этапов:

1. Лексический и синтаксический анализ входного языка, получение абстрактного синтаксического дерева.

2. Семантический анализ, во время которого выполняется расширение абстрактного синтаксического дерева дополнительной информацией.

3. Преобразование синтаксического дерева входного языка в синтаксическое дерево языка Scheme и порождение по полученному дереву файла, который является программой на Racket.

На первом этапе, с помощью лексера и парсера, созданных генератором парсеров ANTLR (описан в главе 2.2.1) выполняется лексический и синтаксический анализ. По завершении анализа ANTLR возвращает абстрактное синтаксического дерево.

На следующем этапе выполняется преобразование синтаксического дерева во внутренний формат препроцессора: каждый узел расширяется с помощью дополнительной информации — таблица символов, уникальные идентификаторы узлов и т.д.. Одновременно с этим происходит семантический анализ, заключающийся в проверке символов на видимость.

На третьем этапе выполняется рекурсивный обход абстрактного синтаксического дерева. Рекурсивный обход выполняется с помощью рекурсивного спуска — каждая вызываемая при обходе функция соответствует одному токену из грамматики входного языка. Во время обхода порождается синаксическое дерево языка Scheme — иерархическая структура из списков и строк. Которая, по окончании обхода преобразуется в файл, который является программой на Racket. Если функция main определена в теле программы, то в конец файла добавляется её вызов.

2.1. Особенности реализации

2.1.1. Обработка ошибок

Все ошибки условно делятся на две группы — синтаксические и семантические ошибки.

Синтаксические ошибки — это ошибки возникающие в лексическом и синтаксическом анализаторах. Эти ошибки порождаются с помощью классов, сгенерированных ANTLR.

В ANTLR предусмотрен интерфейс для изменения поведения лексера и парсера при возникновении ошибки. Этот интерфейс представляет собой работу с наследниками класса ErrorListener. ErrorListener определяет поведение лексера или парсера при возникновении синтаксической ошибки. Можно добавлять или удалять наследников класса ErrorListener, тем самым модифицируя обработку ошибок.

В версии ANTLR для Java данный механизм полноценно работает, но в Python версии ANTLR находится только ядро этого механизма и отсутствуют такой важный функционал, как удаление наследников класса ErrorListner. То есть можно расширишить уже существующую обработку ошибок, но переопределить её нельзя. Из-за описанного недостатка Python версии ANTLR необходимо обращаться к приватной переменной, содержащей наследников класса ErrorListner для переопределения обработки ошибок.

Семантические ошибки — это ошибки возникающий при семантическом разборе. В рамках данной работы рассматривалась только одна семантическая ошибка — идентификатор не найден.

Все ошибки выводятся в момент возникновения в формате: [line, column]: message, где line — номер строки во входном файле, column — номер столбца во входном файле, message — сообщение об ошибке.

Если по завершении лексического и синтаксического анализа произошла хотя бы одна синтаксическая ошибка, то выполнение прерывается. Если синтаксических ошибок не было, то начинает работу семантический анализатор.

Если по завершении семантического анализа произошла хотя бы одна семантическая ошибка, то выполнение прерывается. Если семантических ошибок не было, то порождается результирующий файл.

2.1.2. Проверка символов на доступность

Для проверки символов на доступность используются таблицы символов. Таблица символов для узла дерева синтаксического разбора — это отображение иден-

тификаторов символов, видимых в этом узле, в описания соответствующих этим символам сущностей.

В реализации препроцессора таблицы символов хранятся только в корневом узле, а также в узлах соответствующих определениям функций (как обычных, так и анонимных). Таблицы символов заполняются одновременно с преобразованием абстрактного синтаксического дерева ANTLR в внутренний формат препроцессора.

Во время обхода дерева в глубину, последовательно заполняются таблицы и выполняется проверка на видимость символов. Таблицы изменяются, если во время обхода заходим в нетерминалы $\langle scheme\ block \rangle$, $\langle function - define \rangle$ или $\langle function - define \rangle$ следующим образом:

1. Если нетерминал $\langle scheme\ block \rangle$, то аргументы экспортируемой функции добавляются в локальную таблицу символов, имя экспортируемой функции добавляется в таблицу символов на уровень выше (то есть в корень дерева разбора).

2. Если нетерминал $\langle function\ define \rangle$, то аргументы определяемой функции добавляются в локальную таблицу символов, имя экспортируемой функции добавляется в таблицу символов ближайщего, при подъёме по дереву вверх, нетерминала $\langle function - define \rangle$, если такой нетерминал не найден, то добавляется в корень дерева разбора.

3. Если нетерминал $\langle lambda\ function \rangle$, то аргументы определяемой функции добавляются в локальную таблицу символов.

Если во время обхода заходим в терминал $IDENTIFIER$, то происходт подъём по абстрактному синтаксическому дереву с целью поиска этого идентификатора в какой-либо таблице символов. Если символ найден — то в синтаксическое дерево добавляется информация о его типе, иначе выводится информация об ошибке.

2.1.3. Защитное программирование

Зашитное программирование — способ написания программ при котором появляющиеся ошибки легко обнаруживаются и идентифицируются программистом. При написании компилятора использовался такой метод защитного программирования, как утверждения.

Утверждение (assertion) — код, используемый во время разработки, с помощью которого программа проверяет правильность своего выполнения. Утверждения используются для прерывания программы на ситуациях, которые никогда не должны были произойти [6].

Язык Python поддерживает использование утверждений с помощью ключевого слова ‘assert’.

В препроцессоре на стадии семантического анализа используются утверждения. Каждая функция, являющаяся частью рекурсивного спуска проверяет узел синтаксического дерева, передаваемый на вход, на совпадение с ожидаемым типом этого узла.

2.1.4. Тестирование

Для повышения эффективности разработки применяется модульное тестирование. В качестве программного обеспечения для тестирования используется фреймворк unittest.

Фреймворк unittest входит в стандартную библиотеку Python и служит базовым инструментом для организации модульных тестов.

В рамках работы выполняется тестирование результатов компиляции входного языка в Scheme. Для этого функционал входного языка разделён на небольшие смысловые части: токены, строки, комментарии, условия, вызовы функций и т.д.. Для каждой части функционала входного языка написан набор тестов, по-возможности, покрывающий все возможные сценарии работы.

2.2. Используемые технологии

2.2.1. ANTLR

ANTLR (ANother Tool for Language Recognition) — генератор нисходящих анализаторов для формальных языков. ANTLR преобразует контекстно-свободную грамматику в виде РБНФ в программу на C++, Java, C#, Python, Ruby. В препроцессоре используется ANTLR версии 4. С помощью него порождается лексический и синтакический анализаторы для языка Python.

Преимуществами ANTLR являются: использование единой нотации грамматики как для генератора лексического, так и для генератора синаксического анализатора, применение нисходящего анализа, предоставление сообщений об ошибках.

Главным недостатком ANTLR, в отличие от Flex/Bison, является использование LL-анализатора, заместо более быстрого LALR. Стоит отметить, что в ходе использования выяснились многие проблемы версии ANTLR для Python:

- отсутствие документации — разработчики предполагают использование документацию для Java версии ANTLR,
- ограниченный функционал — многие элементы интерфейса ANTLR, описанные в документации для Java версии ANTLR, не реализованы, это вызвало проблемы при обработке ошибок (подробнее в главе 2.1.1),
- незадокументированные различия в поведении классов InputStream и FileStream, превращающих в поток символов строку и текстовый файл соответственно,
- низкая производительность (подробнее в главе 3.2).

ANTLR поставляется в виде библиотеки для языка Python — antlr4-python2-runtime, а также java-приложения. С помощью java-приложения по описанию грамматики генерируются файлы на языке Python, содержащие лексический и синтаксический анализаторы. При наличии установленной библиотеки antlr4-python2-runtime, полученные анализаторы можно использовать в произвольных Python-приложениях.

2.2.2. Graphviz

Graphviz (сокращение от англ. Graph Visualization Software) — пакет утилит по автоматической визуализации графов, заданных в виде описания на языке DOT.

Graphviz используется для визуализации абстрактного синтаксического дерева. По синтаксическому дереву генерируется файл на языке DOT, который с использованием утилиты dot из пакета graphviz преобразуется в изображение абстрактного синтаксического дерева в формате pdf.

2.3. Установка

Установка приложения происходит в два этапа: генерация лексического и синтаксического анализаторов с помощью ANTLR (описано в главе 2.2.1) и установка препроцессора в виде пакета на языке python.

Для установки препроцессора используется пакет distutils, который входит в стандартную библиотеку Python. Пакет distutils обеспечивает сборку и установку дополнительных модулей для языка Python. На листинге 1 показан скрипт, конфигурирующий distutils. В нём устанавливается две точки входа: одна для препроцессора (lactose), другая для модуля тестирования (tests), зависимость от пакета antlr4-python2-runtime, а также указывается какую точку входа использовать для тестирования.

```
from setuptools import setup, find_packages

setup(name='lactose',
      packages=find_packages(),
      entry_points = {
        'console_scripts': [
            'lactose = lactose.main:main',
            'tests = tests.tests:main',
        ]
      },
      install_requires=['antlr4-python2-runtime'],
      zip_safe=False,
      test_suite="tests")
```

Листинг 1: Скрипт конфигурации для утилиты distutils

Для связки двух этапов используется make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Утилита make использует файл конфигурации: Makefile. Ипользуемый Makefile приведён на листинге 2. Данный Makefile позволяет устанавливать пропроцессор в директорию, конфигурируемую с помощью переменной окружение PREFIX, а также очищать временные файлы, которые генерируются при сборке пакета.
Установка препроцессора состоит из следующих шагов:

1. получение свежей версии препроцессора из репозитория,
2. установка пакета antlr4-python2-runtime,
3. установка препроцессора с помощью утилиты make,

```
PREFIX ?= /usr/local
BIN_DIR = $(PREFIX)/bin
LIB_DIR = $(PREFIX)/lib/python2.7/site-packages


INSTALL_TARGETS = install-dir install-package clean

install: $(INSTALL_TARGETS)

install-dir:
	install -d $(PREFIX) $(BIN_DIR) $(LIB_DIR)

install-package: install-dir clean
	java -jar ./lib/antlr-4.5-complete.jar -Dlanguage=Python2 \
	    ./lactose/grammar/lactose.g4
	PYTHONPATH=$(LIB_DIR) python setup.py test install --prefix=$(
	    PREFIX)
	@echo 'Lactose successfully installed'

clean:
	@find . -name \*.pyc -delete
	@rm -rf build dist lactose.egg-info
```

Листинг 2: Скрипт конфигурации для утилиты make

4. если требуется функционал по выводу синтаксического дерева в pdf файл, то также необходимо установить пакет graphviz.

Пример установки препроцессора, использующий утилиты: git, pip и apt:

```
pip install antlr4-python2-runtime
apt-get install graphviz
git clone https://github.com/art-vybor/lactose.git
cd lactose
make install
```

2.4. Интерфейс

Препроцессор является приложением, предоставляющим интерфейс командной строки (в квадратных скобках указаны необязательные параметры):

```
lactose [-h] -i filename [-o filename] [--run]
        [--stack_trace] [--lexems] [--console_tree] [--pdf_tree]
```

Интерфейс командной строки обеспечивает доступ к основным функциям и настройкам:

- печать справки по интерфейсу командной строки (ключ –h),
- параметризация входного файла (параметр –i),
- параметризация выходного файла — по умолчанию создаёт выходной файл рядом с входным с заменой расширения файла на “rkt” (параметр –o),
- немедленное выполнение программы после компиляции (ключ ––run),
- печать стека вызовов при возникновении ошибки (ключ ––stack_trace),
- вывод распознанных лексем (ключ ––stack_trace),
- вывод дерева разбора в консоль (ключ ––console_tree),
- вывод дерева разбора в графическом виде в файл PDF — создаёт файл рядом с входным с заменой расширения файла на “pdf” (параметр (ключ ––pdf_tree).

Пример компиляции файла sample.lc с последующим запуском:

```
lactose -i sample.lc --run
```

3. Тестирование

3.1. Удобство использования

В таблице 3.1 приведены реализации одних и тех же функций на входном языке и на Scheme. Как видно из таблицы, код на входном языке намного более легковесный, не перегружен скобками, обладает более выразительными арифметическими операциями.

Входной язык	**Scheme**
`def abs x = if x < 0 then -x else x`	`(define (abs x) (if (< x 0) -x x))`

`def fact n =`
`  if n == 0 then 1`
`  else n*(fact n-1)`
	`(define (fact n)`
	`  (if (= n 0) 1`
	`  (* n (fact (- n 1)))))`

`def make_adder n = \x -> x+n`	`(define (make_adder n)` `  (lambda (x) (+ x n)))`

`def filter proc lst =`
`  def loop lst n = {`
`  if (len lst) == n then`
`  lst`
`  else if proc (ref lst n)`
`  then loop lst n+1`
`  else`
`  loop (concat (take lst n)`
`  (tail lst n+1)) n};`
`  loop lst 0`
	`(define (filter proc lst)`
	`  (define (loop lst result)`
	`  (cond ((null? lst)`
	`  (reverse result))`
	`  ((proc (car lst))`
	`  (loop (cdr lst)`
	`  (cons (car lst) result)))`
	`  (else`
	`  (loop (cdr lst) result))))`
	`  (loop lst '()))`

Таблица 1: Сравнение входного языка и Scheme

3.2. Производительность

Во время разработки обнаружен факт очень медленной компиляции небольших программ. Был составлен тест, состоящий из множественного определения функций abs, вида: def abs{INDEX} x = if x >= 0 then x else −x, где INDEX — целое число, уникальное для каждого определения функции. На рисунке 1 показан гра-

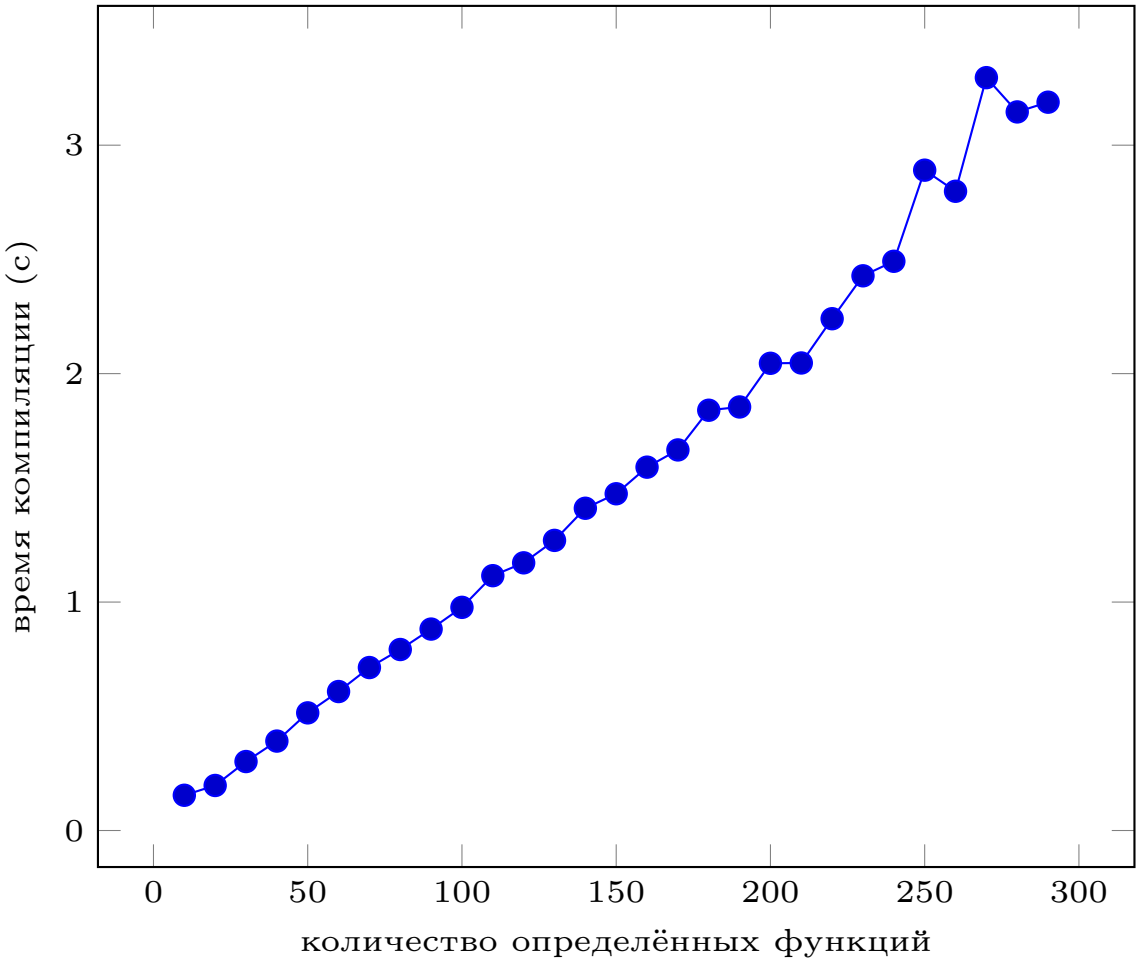

Рисунок 1 — Результаты тестирования производительности

фик зависимости времени компиляции от количества определённых функций abs в теле программы. Как видно из рисунка время компиляции программы линейно зависит от количества определений. При этом абсолютное значение времени достаточно велико — для программы из 100 определений порядка 1 секунды.

Для определения узкого места в программе, воспользуемся профилировщиком cProfile и визулизатором gprof2dot. Профилировщик находит время выполнения каждой функции программы, а визуализатор строит дерево выполнения программы в формате dot. Для каждой функции создаётся узел дерева, содержащий информа-

цию об имени функции, проценте времени, затраченного на выполнение функции, относительно времени работы всей программы и количестве вызовов функции. Получившееся дерево можно визуализировать с помощью утилиты graphviz.

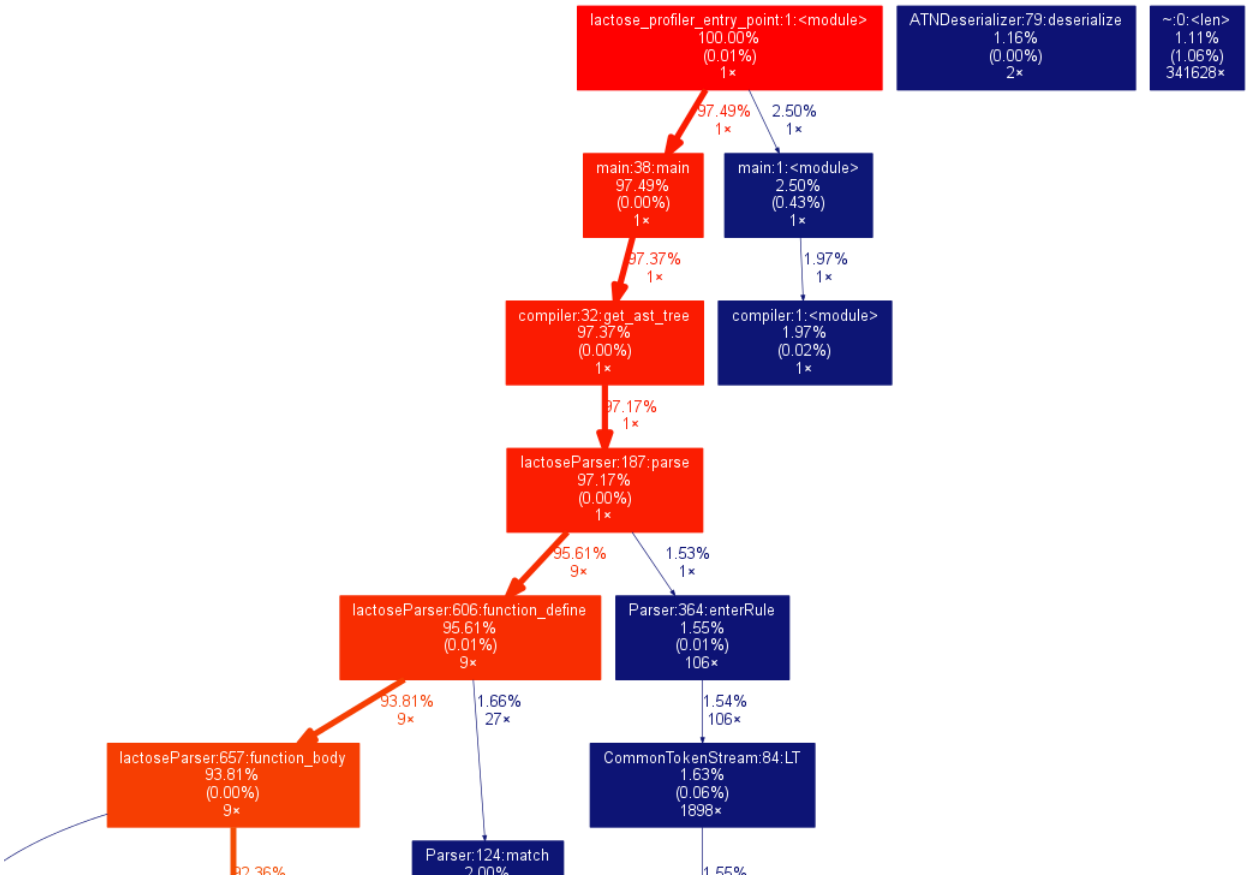

Рисунок 2 — Результат работы профилировщика

Часть получившегося дерева, приведена на рисунке 2 (использованы настройки gprof2dot, которые отсекают от дерева вершины, соответствующие функциям занявшим менее 1 процента времени исполнения программы). Из рисунка видно, что 97 процентов времени выполнения программы заняла функция parse. Функция parse — это функция запускающая лексический и синтаксический анализ с помощью ANTLR.

Из полученных результатов можно сделать вывод, что python версия ANTLR очень медленна и непригодна для промышленного использования.

4. Заключение

В рамках данной работы была сделана попытка освежить Scheme, добавить ему элегантности, сделать код более удобочитаемым. Для этого был разработан функциональный динамический язык, обладающий синтаксическим сахаром, позаимствованным из Haskell и Python. С помощью написанного препроцессора, входной язык комплируется в Scheme.

Полученный входной язык обладает сильно ограниченным набором возможностей, поэтому он представляет чисто академический интерес и, требует серьёзной доработки, перед практическим применением.

Список литературы

[1] Matthew Flatt, Robert Bruce Findler. The Racket Guide. Racket Documentation: URL: http://docs.racket-lang.org/guide/index.html.

[2] R. Kelsey, W. Clinger, J. Rees. Revised^5 Report on the Algorithmic Language Scheme. Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998

[3] Simon Marlow. Haskell 2010 Language Report, 2010.

[4] Terence Parr. The Definitive ANTLR 4 Reference. 2013.

[5] Ахо, Альфред В., Лам, Моника С., Сети, Рави, Ульман, Джеффри Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. – М.: ООО «И.Д.Вильямс», 2008 – 1184 с.

[6] Макконнелл С., Совершенный код. Мастер-класс: Пер. с англ. — М.: Издательство «Русская редакция», 2014 — 896 стр.