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

- уточнить название работы
- что делать с тестированием
-

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 реадактор 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.

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

1. Теоретическая часть

1.1. Scheme

Поподробнее про схему. В данной работе рассматривается версия языка Scheme, удолетворяющая стандарту r5rs [1].

1.2. Входной язык

вступленние

Программа на исходном языке представляет собой множество определений функций. Переменные также рассматриваются как функции. Входной точкой программы является функция main.

подробнее про функцию main

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:

$\langle token \rangle ::= \langle BOOLEAN \rangle \mid \langle NUMBER \rangle \mid \langle CHARACTER \rangle \mid \langle STRING \rangle$

Нетерминал, соответствующий идентификатору — <IDENTIFIER>, рассматривается отдельно.

Единственным существенным отличием лексической структуры входного языка от Scheme является отсутствие знака перед числом, так как во входном языке знак

перед числом определяется с помощью унарного минуса, который, в свою очередь, встроен в интерпретатор и фактически является элементом синтаксиса языка. Зависимость от регистра символов определяется настройками интерпретатора Scheme, работающем в связке с нашим компилятором.

Документация Scheme [1] содержит следующее определение строки в РБНФ:

⟨*string*⟩ ::= ‘"’ ⟨*string element*⟩* ‘"’

⟨*string element*⟩ ::= any character other than ‘"’ or ‘\’ | ‘\" ’| ‘\\’

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.

⟨*string*⟩ ::= ‘"’ ⟨*string element*⟩* ‘"’

⟨*string element*⟩ ::= any character other than ‘"’ or ‘\’ | ‘\" ’| ‘\’

1.2.2. Выражения

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

Грамматика выражений в формате РБНФ:

⟨*expression*⟩ ::= ⟨*expression*⟩ ‘`**`’ ⟨*expression*⟩
| ⟨*expression*⟩ (‘`*`’|‘`/`’|‘`%`’|‘`//`’) ⟨*expression*⟩
| ⟨*expression*⟩ (‘`+`’|‘`-`’) ⟨*expression*⟩
| ⟨*expression*⟩ (‘`<<`’ | ‘`>>`’) ⟨*expression*⟩
| ⟨*expression*⟩ ‘`&`’ ⟨*expression*⟩
| ⟨*expression*⟩ ‘`^`’ ⟨*expression*⟩
| ⟨*expression*⟩ ‘`|`’ ⟨*expression*⟩
| ⟨*expression*⟩ ‘`and`’ ⟨*expression*⟩
| ⟨*expression*⟩ ‘`or`’ ⟨*expression*⟩
| ⟨*expression*⟩ (‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’) ⟨*expression*⟩
| ⟨*expression*⟩ (‘`==`’ | ‘`!=`’) ⟨*expression*⟩
| (‘`+`’|‘`-`’) ⟨*expression*⟩
| (‘`not`’|‘`~`’) ⟨*expression*⟩
| ⟨*if condition*⟩

| ⟨*token*⟩
| ⟨*IDENTIFIER*⟩
| ⟨*function call*⟩
| ⟨*lambda function call*⟩
| '`(`' ⟨*expression*⟩ '`)`'

написать про приоритеты

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

1.2.3. Условия

В качестве условного оператора используется традиционный оператор if, который записывается в формате РБНФ следующим образом:

⟨*if condition*⟩ ::= '`if`' ⟨*expression*⟩ '`then`' ⟨*expression*⟩ '`else`' ⟨*expression*⟩

Оператор if 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

добавить текст

⟨*function define*⟩ ::= '`def`' ⟨*IDENTIFIER*⟩ ⟨*function arguments*⟩ '`=`' ⟨*function body*⟩

⟨*function body*⟩ ::= ⟨*function body token*⟩ ('`;`' ⟨*function body token*⟩)*

⟨*function body token*⟩ ::= ⟨*function define*⟩ | ⟨*expression*⟩

⟨*function arguments*⟩ ::= ⟨*IDENTIFIER*⟩*

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

1.2.5. Анонимные функции

добавить текст

⟨*lambda function*⟩ ::= '`\`' ⟨*function arguments*⟩ '`->`' ⟨*function body*⟩

Пример анонимной функции от двух аргументов, которая находит сумму квадратов двух чисел: \x y$->$ x$*$x + y$*$y.

1.2.6. Вызов функций

добавить текст

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

⟨*lambda function call*⟩ ::= ‘(’ ⟨*lambda function*⟩ ‘)’ ⟨*expression*⟩*

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

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

1.2.7. Импортирование символов из Scheme

сначала надо реализовать

Пример: {...} export a,b

2. Объекты и методы

Характеристики программного обеспечения:

- Операционная система — Ubuntu 14.04 LTS x64.
- Язык программирования — Python 2.7.3.

Характеристики оборудования:

- Процессор — Intel Core i7-3770K 3.50 Гц 8 ядер.
- Оперативная память — 16 Гбайт DDR3.

3. Реализация

3.1. Используемые технологии

3.1.1. ANTLR

что круто, а что не очень обязательно про минусы python версии особенно про различное поведение при компиляции из файла и строки

3.1.2. Graphviz

...

3.1.3. Python unittest

...

3.1.4. Python distutils

...

3.2. Особенности реализации

общая структура:
особоенности каждого этапа преобразования

3.2.1. Обработка ошибок

сделано

3.2.2. Видимость символов

увы нет, надо обязательно доделать

3.3. Интерфейс

CLI все дела
примеры использования

4. Тестирование

что писать? сравнение синтаксиса с другими языками

4.1. Производительность

График производительности от объёма нагенеренной программы.

Это медленно.

Для того, чтобы разобраться в чём проблема воспользуемся планировщикова cProfile и визулизатором gprof2pdf. Описание того, что это и как использовали.

Вуаля:

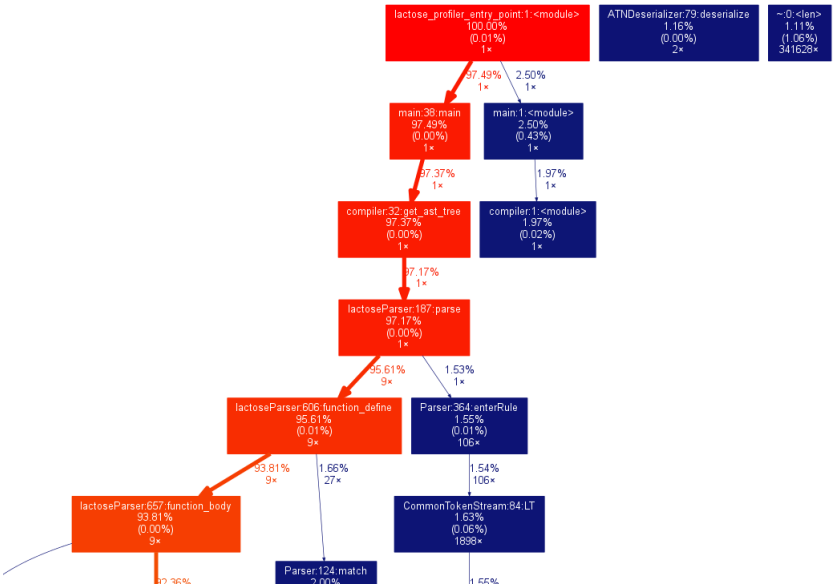

Рисунок 1 — Результат работы профилировщика

ANTLR для Python слоупок.

5. Заключение

Что получилось. Привнесена няшность, но возможности далеко не такие как были.

Список литературы

[1] стандарт r5rs

[2] ахо ульман книга дракона

[3] какая нибудь дока по antlr