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
- упоминание lactose
- объекты и методы, мб туда внести что-то из реализации?
- куда вставить инфу про замыкания?
- куда впихнуть пару слов об assert?
- хватит ли литературы? мб зафилирить её ссылкой на статьи по технологиям

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 ДОПИСАТЬ

схема - это ...

racket - это ...

такие-то основные фишки

такие-то основные недостатки

В данной работе порождается язык Scheme, удолетворяющая стандарту r5rs [2].

1.2. Входной язык ДОПИСАТЬ

Программа на исходном языке представляет собой множество определений функций.

Переменные также рассматриваются как функции.

Входной точкой программы является функция 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 — $\langle IDENTIFIER \rangle$, 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 [2] 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*⟩ ::= любой символ кроме ‘"’ или ‘\’ | ‘\" ’| ‘\’

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.

Выражение задаётся нетерминалом ⟨*expression*⟩. 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*⟩ ‘)’

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 ⟨*if condition*⟩. Условный оператор в формате РБНФ:

⟨*if condition*⟩ ::= ‘if’ ⟨*expression*⟩ ‘then’ ⟨*expression*⟩ ‘else’ ⟨*expression*⟩

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 набора выражений, которые, при вызове функции, последовательно исполняются. Функция возвращает результат последнего выражения из тела функции.

В грамматике определение функции соответствует нетерминалу ⟨*function define*⟩. Синтаксис определения функции в РБНФ:

⟨*function define*⟩ ::= ‘def’ ⟨*IDENTIFIER*⟩ ⟨*function arguments*⟩ ‘=’ ⟨*function body*⟩

⟨*function body*⟩ ::= ⟨*expression*⟩ (‘;’ ⟨*expression*⟩)*

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

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

1.2.5. 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 ⟨*lambda function*⟩. Синтаксис анонимной функции в РБНФ:

⟨*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

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 ⟨*function call*⟩ и ⟨*lambda function call*⟩, 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.

1.2.7. Комментарии

Комментарии представляют собой часть строки, начинающуюся с символа ‘–’ и заканчивающуюся при переводе строки. Комментарии препроцессором полностью игнорируются.

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

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

Пример комментария: −− commented text\n

1.2.8. Импортирование символов из Scheme ДОПИСАТЬ

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

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 синтаксическое дерево входного языка преобразуется в абстрактное синтаксическое дерево языка Scheme.

3. По дереву языка Scheme порождается файл, который является программой на Racket.

На первом этапе, с помощью лексера и парсера, созданных генератором парсеров ANTLR (описан в главе 3.2.1) выполняется лексический и синтаксический анализ. По завершении анализа ANTLR возвращает абстрактное синтаксического дерево, которое преобразуется в внутренний формат препроцессора: каждый узел расширяется с помощью дополнительной информации — таблица символов, уникальные идентификаторы узлов и т.д..

На следующем этапе выполняется семантический анализ, который заключается в рекурсивном обходе абстрактного синтаксического дерева. Во время обхода порождается синаксическое дерево языка Scheme — иерархическая структура из списков и строк, а также проверяется видимость символов, а именно, идентификаторов.

Последний этап по иерархической структуре из списков и строк, содержащей абстрактное синтаксическое дерево языка Scheme порождается файл, который является программой на Racket. Если функция main определена в теле программы, то в конец файла добавляется её вызов.

3.1. Особенности реализации

3.1.1. 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.

3.1.2. Видимость символов

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 заполняется одновременно с преобразованием абстрактного синтаксического дерева ANTLR в внутренний формат препроцессора. В таблицу символов для узла попадают все функции определённые на нижележащем уровне дерева. Для каждой функции, в узле её определяющем, хранится список аргументов.

На этапе семантического анализа для каждого идентификатора выполняется подъём по абстрактному синтаксическому дереву. Если на каком-либо уровне находится либо символ из таблицы, либо аргумент из функции совпадающий с идентификатором, то семантический анализ продолжается, иначе считается, что данный идентификатор на найден и печатается сообщение об ошибке.

3.1.3. 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.

3.2. Используемые технологии

3.2.1. ANTLR ДОПИСАТЬ

antlr это ....

antlr обладает такими-то преимуществами и такими-то недостатками

python версия antlr сырована, но работает.

Во время работы с ней пришлось нарушить инкапсуляцию в рамках обработки ошибок. Отсылка в обработку ошибок.

Также возникла проблема с различием поведения при компиляции из файла и из строки

упомянуть про тормознутость и отослать в тестирование

рассказать как скомпилировать

3.2.2. 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.

3.3. Установка

Установка приложения происходит в два этапа: генерация лексического и синтаксического анализаторов с помощью ANTLR (описано в главе 3.2.1) 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
```

3.4. 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
```

4. Тестирование

что писать?

4.1. Няшность ПЕРЕИМЕНОВАТЬ, ДОПИСАТЬ

сравнение синтаксиса с другими языками

4.2. Производительность ДОПИСАТЬ

График производительности от объёма нагенеренной программы.

Это медленно.

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

Вуаля:

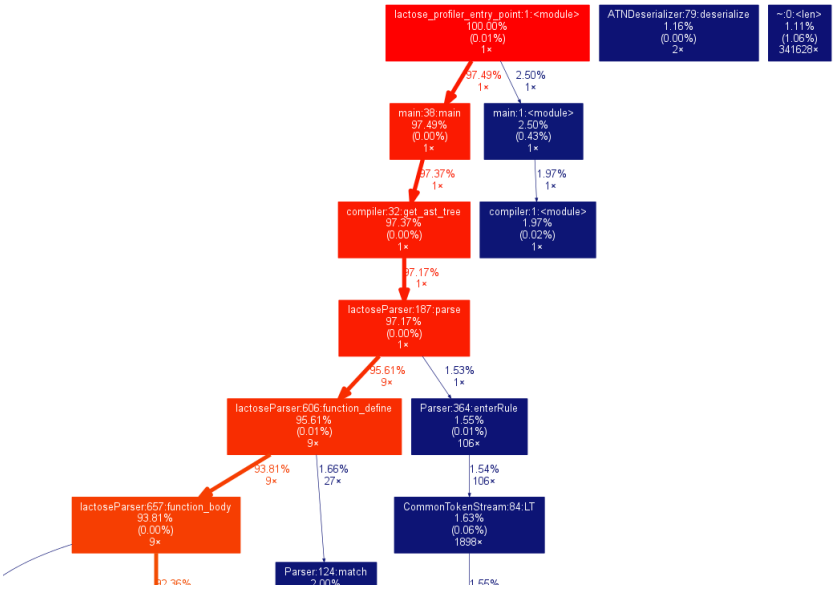

Рисунок 1 — Результат работы профилировщика

Вывод ANTLR для Python слоупок.

5. Заключение ДОПИСАТЬ

Что получилось. Привнесена няшность, но возможности ограничены. Предложенный язык надо дальше развивать

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

[4] 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.