МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра информационных систем управления

Палюхович Антон Адамович

РАЗРАБОТКА ИНТЕРПРЕТАТОРА ЯЗЫКА LISP

Курсовой проект
студента 3 курса 2 группы

“Допустить к защите“

Зав. кафедрой ИСУ

_____

“___” _____ 2020 г.

Руководитель:

Конах Валентина Владимировна

старший преподаватель кафедры ИСУ

Минск 2020

АННОТАЦИЯ

Палюхович А.А. Разработка интерпретатора языка Lisp: Курсовой проект/Минск: БГУ, 2020. – 31 с.

В курсовом проекте рассматриваются основные концепции и особенности интерпретирования языков. На основе прочитанной литературы проанализированы и описаны общие черты и особенности организации интерпретатора языка Lisp, а также реализован диалект Scheme.

АНАТАЦЫЯ

Палюхович А.А. Распрацоўка інтэрпрэтатара мовы Lisp: Курсавы праект / Мінск: БДУ, 2020. – 31 с.

У курсавым праекце разглядаюцца асноўныя канцэпцыі і асаблівасці интерпретирования моў. На аснове прачытанай літаратуры прааналізаваныя і апісаны агульныя рысы і асаблівасці арганізацыі інтэрпрэтатара мовы Lisp, а таксама рэалізаваны дыялект Scheme.

ANNOTATION

Paliukhovich A.A. Lisp interpreter development: Course project / Minsk: BSU, 2020. – 31 p.

In the course project addresses the basic concepts and features of the interpretation of languages. Based on the read literature, the general features and organization features of the Lisp language interpreter are analyzed and described, and the Scheme dialect is implemented.

РЕФЕРАТ

Курсовой проект, 31 с., 1 рис., 6 источников, 1 приложение.

Ключевые слова: ИНТЕРПРИТАТОР, LISP, SCHEME, ПРИЛОЖЕНИЕ, C++.

Объект исследования – концепция интерпретирование высокоуровневого языка.

Цель работы – изучить основные концепции и особенности процесса интерпретирования высокоуровневого языка Lisp. Проанализировать и описать общие черты и особенности исследуемой области. Подготовить пример приложения, реализованного на языке C++, производящего процесс интерпретации диалекта Scheme.

Методы исследования – теория программирования, анализ.

Результаты работы – произведен анализ процесса интерпретирования языка, написан интерпретатор языка Lisp.

Областью применения является разработка высокоуровневых интерпретаторов языков.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Процедура интерпретации — пооператорный анализ, обработка и тут же выполнение исходной программы или запроса, в отличие от компиляции, при которой программа транслируется без её выполнения. Таким образом, можно заключить, что компилятор преобразует исходный код к близкому к машинному коду. Интерпретатор же, позволяет выполнять некоторое представление программы, в процессе преобразуя программный в машинный код. Как привило, поэтому интерпретируемые программы имеют следующие преимущества:

- переносимость программ
- более совершенные и наглядные средства диагностики ошибок в исходных кодах
- упрощение отладки исходных кодов программ
- Меньшие размеры кода, по сравнению с машинным кодом, полученным после обычных компиляторов.

Множество современные интерпретатор и компиляторов языков программирования, позаимствовали некоторые прогрессивные на тот момент концепций из языка Lisp, такие как: автоматическое управление памятью, сборку мусора, императивность и объектно-ориентированность.

Задачей данного курсового проекта является изучение особенностей организации интерпретаторов, а также углубленное изучение и реализация интерпретатора Lisp.

ГЛАВА 1. КЛАССИФИКАЦИЯ ИНТЕРПРИТАТОРОВ

В области компьютерных наук интерпретатор – это компьютерная программа, которая непосредственно выполняет инструкции, написанные на языке программирования или языке сценариев, не требуя, чтобы они были предварительно скомпилированы в программу на машинном языке. Интерпретатор использует одну из следующих стратегий выполнения программы:

- Разбирает исходный код и выполняет его поведение напрямую;
- Переводит исходный код в некоторое эффективное промежуточное представление и немедленно выполните это;
- Явно выполняет хранимый предварительно скомпилированный код, созданный компилятором, который является частью системы интерпретатора.

Ранние версии языка программирования Lisp и Dartmouth BASIC примеры первого типа. Perl, Python, MATLAB и Ruby являются примерами второго, в то время как UCSD Pascal является примером третьего типа. Исходные программы компилируются заранее и сохраняются как машинно-независимый код, который затем связывается во время выполнения и выполняется интерпретатором и / или компилятором (для систем JIT). Некоторые системы, такие как Smalltalk и современные версии BASIC и Java, также могут сочетать две и три. Интерпретаторы различных типов также были созданы для многих языков, традиционно связанных с компиляцией, таких как Algol, Fortran, Cobol, C и C ++.

Большинство систем интерпретации также выполняют некоторые переводческие работы, как и компиляторы. Термины «интерпретируемый язык» или «скомпилированный язык» означают, что каноническая реализация этого языка является интерпретатором или компилятором, соответственно. Язык высокого уровня является абстракцией, независимой от конкретных реализаций.

Программы, написанные на языке высокого уровня, либо напрямую выполняются каким-либо интерпретатором, либо преобразуются в машинный код компилятором для выполнения центральным процессором.

ГЛАВА 2. СРАВНИТЕЛЬНЫЙ АНАЛИЗ ИНТЕРПРЕТАТОРОВ И КОМПИЛЯТОРОВ

Компиляторы создают машинный код, непосредственно исполняемый компьютерным оборудованием, также компиляторы в большинстве случаев могут создавать промежуточную форму, называемую объектным кодом. Это в основном тот же машинно-специфический код, но дополненный таблицей символов с именами и тегами, чтобы сделать исполняемые блоки идентифицируемыми и перемещаемыми. Скомпилированные программы обычно используют функции, хранящиеся в библиотеке таких модулей объектного кода. Линкер используется для объединения предварительно созданных библиотечных файлов с объектными файлами приложения для формирования единого исполняемого файла. Таким образом, объектные файлы, которые используются для генерации исполняемого файла, часто создаются в разное время, а иногда даже на разных языках способных генерировать один и тот же формат объекта.

Простой интерпретатор, написанный на языке низкого уровня, может иметь аналогичные блоки машинного кода, реализующие функции языка высокого уровня, которые хранятся и выполняются, когда запись функции в таблице поиска указывает на этот код. Однако интерпретатор, написанный на языке высокого уровня, использует другой подход, такой как генерация и последующее обход дерева разбора, или генерация и выполнение промежуточных программно-определенных инструкций, или обоих.

Таким образом, и компиляторы, и интерпретаторы превращают исходный код в токены и генерируют дерево разбора. Основные различия между компилятором и интерпретатором состоят в том, что система компилятора, включая компоновщик, генерирует отдельную программу машинного кода, тогда как система интерпретатора выполняет действия, описанные программой высокого уровня.

Таким образом, компилятор может сделать почти все преобразования из семантики исходного кода на машинный уровень раз и навсегда, в то время как интерпретатор должен выполнять некоторые из этих операций преобразования каждый раз, когда выполняется оператор или функция. Однако в эффективном интерпретаторе большая часть работы по переводу выстраивается и выполняется только при первом запуске программы, модуля, функции или даже оператора, таким образом, довольно сродни тому, как Компилятор работает. Однако скомпилированная программа по-прежнему работает намного быстрее, в большинстве случаев, отчасти потому, что компиляторы предназначены для оптимизации кода, и на это может быть выделено достаточно времени. Это особенно верно для простых языков высокого уровня без большой части динамических структур данных, проверок типов.

В традиционной компиляции исполняемый вывод компоновщиков .exe-файлы или .dll-файлы или библиотека обычно перемещается при работе в общей операционной системе, так же как модули объектного кода, но с той разницей, что

это перемещение выполняется динамически во время выполнения, т.е. когда программа загружена для выполнения. С другой стороны, скомпилированные и связанные программы для небольших встроенных систем обычно размещаются статически, часто жестко запрограммированы во флэш-памяти NOR, так как зачастую в этом смысле нет вторичного хранилища и операционной системы. [2]

ГЛАВА 3. СЕМЕЙСТВО ЯЗЫКОВ LISP

Lisp — семейство языков программирования, программы и данные в которых представляются системами линейных списков символов. Lisp был создан для работ по искусственному интеллекту и до сих пор остаётся одним из основных инструментальных средств в данной области. Применяется он и как средство обычного промышленного программирования, от встроенных скриптов до веб-приложений массового использования.

Традиционный Лисп имеет динамическую систему типов, а также использует автоматическое управление памятью и сборку мусора. Язык является функциональным, но начиная уже с ранних версий обладает также чертами императивности, к тому же, имея полноценные средства символьной обработки, позволяет реализовать объектно-ориентированность.

Common Lisp, Scheme и Clojure основные диалекты языка Lisp.

Common Lisp является языком программирования общего назначения и поэтому имеет большой языковой стандарт, включающий множество встроенных типов данных, функций, макросов и других элементов языка, а также объектную систему. Common Lisp также заимствовал некоторые особенности из Scheme, такие как лексическая область видимости и лексические замыкания. Стандартные реализации Lisp доступны для работы с различными платформами, такими как LLVM, [39] виртуальная машина Java, [40] x86-64, PowerPC, Alpha, ARM, Motorola 68000 и MIPS, [41] и операционными системами, такими как Windows. , macOS, Linux, Solaris, FreeBSD, NetBSD, OpenBSD, Dragonfly BSD и Heroku. [42]

Scheme - это статически ограниченный и правильно рекурсивный диалект языка программирования Лисп. Он был разработан, чтобы иметь исключительно ясную и простую семантику и несколько различных способов формирования выражений. Scheme, разработанная примерно на десять лет раньше, чем Common Lisp, является более минималистской. Он имеет гораздо меньший набор стандартных функций, но с некоторыми функциями реализации, не заданными в Common Lisp. Широкий выбор парадигм программирования, включая императивный, функциональный и стиль передачи сообщений.

Clojure – это поздний диалект Lisp, ориентированный в основном на виртуальную машину Java, а также Common Language Runtime (CLR), Python VM, Ruby VM YARV и компиляция в JavaScript. Он разработан, чтобы быть прагматичным языком общего назначения. Clojure предоставляет доступ к фреймворкам и библиотекам Java с необязательными подсказками типов и выводами типов, так что вызовы Java могут избежать отражения и обеспечить быстрые примитивные операции. [1]

ГЛАВА 4. СИНТАКСИС ЯЗЫКА SCHEME

В Scheme можно использовать символы, знаки, строки, списки, числа, логические значения, векторы, порты и функции.

Каждый из этих типов данных имеет соответствующий предикат: `symbol?`, `char?`, `string?`, `pair?`, `number?`, `boolean?`, `vector?`, `procedure?`.

Помимо них в наличии есть процедуры-аксессоры и модификаторы для тех типов, где это имеет смысл: `string-ref`, `string-set!`, `vector-ref` и `vector-set!`.

Для списков они называются `car`, `cdr`, `set-car!` и `set-cdr!`.

Функции `car` и `cdr` могут комбинироваться. Например, для доступа ко второму элементу списка используется `cadr`.

Все значения этих типов могут быть непосредственно записаны в программе. С символами и числами всё обстоит следующем образом. Перед знаками пишется префикс `#\`, например: `#\Z`, `#\+`, `#\space`. Строки окружаются двойными кавычками "", списки — круглыми скобками `()`. Логические значения записываются как `#t` и `#f` соответственно. Для записи векторов используется синтаксис `#(do re mi)`. Естественно, такие значения могут быть построены и динамически с помощью `cons`, `list`, `string`, `make-string`, `vector`, `make-vector`. Также в наличии есть функции приведения типов вроде `string->symbol` и `int->char`.

Программы на Scheme представляются так называемыми формами. Форма `begin` позволяет сгруппировать формы и вычислить их последовательно; например, `(begin (display 1) (display 2) (newline))`.

Присутствует несколько форм ветвления. Простейшей из них является `if-then-else`, которая на Scheme так и записывается: `(if условие тогда иначе)`. Если вариантов больше двух, то для этого случая в Scheme есть формы `cond` и `case`. Форма `cond` содержит список утверждений, каждое из которых начинается с условия — выражения, возвращающего логическое значение, — за которым располагается последовательность других форм. Она последовательно вычисляет условия утверждений до тех пор, пока одно из них не вернёт истину, а точнее: не ложь, не `#f`; затем вычисляется следствие данного утверждения, и результат его вычисления становится результатом всей формы `cond`. Пример использования этой формы, который заодно показывает ключевое слово `else`:

```
(cond ((eq? x 'flip) 'flop)
      ((eq? x 'flop) 'flip)
      (else (list x "neither flip nor flop")) )
```

Форма `case` похожа на `cond`, но она принимает первым параметром форму, на основе значения которой производится выбор между вариантами. Каждый из вариантов в начале содержит список значений, которые подходят для него. Как только найден подходящий вариант, он вычисляется и этот результат становится результатом всей формы `case`. Аналогично, в конце может стоять универсальный вариант `else`. Вот так можно переписать предыдущий пример с помощью `case`:

```
(case x
  ((flip) 'flop)
  ((flop) 'flip)
  (else (list x "neither flip nor flop")) )
```

Функции определяются формой `lambda`. За словом `lambda` следует список аргументов, а после него — последовательность выражений, которые описывают собственно вычисление функции. Формы `let`, `let*` и `letrec` определяют локальные переменные, они отличаются тонкостями вычисления начальных значений определяемых переменны. Значения переменных в дальнейшем можно изменять с помощью формы `set!`. Для записи литералов используется форма quote.

С помощью формы `define` можно назначить имя любому значению. У неё есть особые возможности, которые мы будем использовать. В частности, возможность использовать её как подобие `let`, а также вариант синтаксиса этой формы, позволяющий удобнее определять функции [3]. Пример программы:

```
(define (rev l)
  (define nil '())
  (define (reverse l r)
    (if (pair? l) (reverse (cdr l) (cons (car l) r)) r))
  (reverse l nil) )
```

Данную программу можно также реализовать следующим образом:

```
(define rev
  (lambda (l)
    (letrec ((reverse (lambda (l r)
                        (if (pair? l) (reverse (cdr l)
                                               (cons (car l) r))
                            r) )))
      (reverse l '()) ) ) )
```

4.1 ПРОСТЫЕ ВЫРОЖЕНИЯ ЯЗЫКА

Ключевые слова, переменные и символы вместе называются идентификаторами. Идентификаторы могут быть сформированы из букв, цифр и определенных специальных символов, включая `?, !, ., +, -, *, /, <, =,>, :, $, %, ^, &, _, ~` и `@`, а также набор дополнительных символов Юникода. Идентификаторы не могут начинаться со знака `(@)` и, как правило, не могут начинаться с любого символа, который может начинать число, то есть цифру, знак плюс `(+)`, знак минус `(-)` или десятичную точку `(.)`. Исключением являются +, - и ..., которые являются допустимыми идентификаторами, и любой идентификатор, начинающийся с ->. Например, `hello, hello, n, x, x3, x + 2 и? $ & * !!!` все идентификаторы. Идентификаторы разделяются пробелами, комментариями, круглыми скобками, скобками, строковыми двойными кавычками " и хэш-метками #.

Простейшими выражениями Scheme являются константные объекты данных, такие как строки, числа, символы и списки. Scheme поддерживает другие типы объектов, но этих четырех достаточно для многих программ.

```
1234567899876543 21 ⇒ 1234567899876543 21
3/4 ⇒ 3/4
2.718281828 ⇒ 2.718281828
2.2+1.1i ⇒ 2.2+1.1i
```

Числа схемы включают точные и неточные целые, рациональные, действительные и комплексные числа. Точные целые и рациональные числа имеют произвольную точность, то есть они могут иметь произвольный размер. Неточные числа обычно представляются внутри, используя стандартные представления IEEE с плавающей точкой.

Scheme предоставляет имена спецификаторов `+, -, *` и `/` для соответствующих арифметических процедур. Каждая процедура принимает два числовых аргумента, записанные в префиксной нотации. Приведенные ниже выражения называются приложениями процедур, поскольку они определяют применение процедуры к набору аргументов.

```
(+ 1/2 1/2) ⇒ 1
(- 1.5 1/2) ⇒ 1.0
(* 3 1/2) ⇒ 3/2
(/ 1.5 3/4) ⇒ 2.0
```

Схема использует префиксную нотацию даже для обычных арифметических операций. Любое приложение процедуры, независимо от того, принимает ли процедура ноль, один, два или более аргументов, записывается как `(procedure arg ...)`. Эта закономерность упрощает синтаксис выражений; одна запись используется независимо от операции, и нет никаких сложных правил относительно приоритета или ассоциативности операторов.

Приложения процедур могут быть вложенными, и в этом случае самые внутренние значения вычисляются первыми. Таким образом, мы можем вкладывать приложения арифметических процедур, приведенных выше, для оценки более сложных формул.

```
(+ (+ 2 2) (+ 2 2)) ⇒ 8
(- 2 (* 4 1/3)) ⇒ 2/3
(* 2 (* 2 (* 2 (* 2 2)))) ⇒ 32
(/ (* 6/7 7/2) (- 4.5 1.5)) ⇒ 1.0
```

Простых числовых объектов достаточно для многих задач, но иногда требуются совокупные структуры данных, содержащие два или более значений. Во многих языках основной структурой совокупных данных является массив. На схеме это список. Списки пишутся в виде последовательности объектов в круглых скобках. Например, `(1 2 3 4 5)` представляет собой список чисел, а `("this" "is" "a" "list")` представляет собой список строк. Списки не обязательно должны содержать только один тип объекта, поэтому `(4.2 "hi")` является допустимым списком, содержащим число и строку. Списки могут быть вложенными (могут содержать другие списки), поэтому `((1 2) (3 4))` является допустимым списком с двумя элементами, каждый из которых представляет собой список из двух элементов.

4.2 ВЫРАЖЕНИЕ QUOTE

Списки выглядят так же, как приложения процедур, пример: список объектов `(obj1 obj2 ...)` и приложение процедуры `(procedure arg ...)`.

В некоторых случаях можно установить различие. Список чисел `(1 2 3 4 5)` можно отличить от процедуры, поскольку 1 – это число, а не название процедура. Таким образом, ответ может заключаться в том, что Scheme смотрит на первый элемент списка или приложения процедуры и принимает решение на основе того, является ли этот первый элемент процедурой или нет. Это решение не всегда работает, так как мы можно рассматривать действующее приложение процедуры, такое как `(+ 3 4)`, как список. Для решения данной проблемы в Scheme заключается в том, что следует рассматривать список как данные, а не как приложение процедуры. Поэтому используется спецификатор `quote`.

```
(quote (1 2 3 4 5)) ⇒ (1 2 3 4 5)
(quote ("this" "is" "a" "list")) ⇒ ("this" "is" "a" "lis
t")
(quote (+ 3 4)) ⇒ (+ 3 4)
```

`quote` заставляет список обрабатываться как данные.

Поскольку в коде Scheme довольно часто требуется `quote`, Scheme распознает одиночную кавычку ('), предшествующую выражению, как сокращение для кавычки.

```
'(1 2 3 4) ⇒ (1 2 3 4)
'((1 2) (3 4)) ⇒ ((1 2) (3 4))
'(/ (* 2 -1) 3) ⇒ (/ (* 2 -1) 3)
```

Выражение кавычки не является приложением процедуры, так как оно препятствует оценке его подвыражения. Это совершенно другая синтаксическая форма. Схема поддерживает несколько других синтаксических форм в дополнение к процедурам приложений и выражений в кавычках. Каждая синтаксическая форма оценивается по-разному. К счастью, количество различных синтаксических форм невелико. Мы увидим больше из них позже в этой главе.

Не все выражения в кавычках содержат списки.

```
(quote hello) ⇒ hello
```

Символ hello должен быть заключен в кавычки, чтобы Scheme не воспринимал hello как переменную. Символы и переменные в схеме аналогичны символам и переменным в математических выражениях и уравнениях. При оценивании математическое выражение 1 - x для некоторого значения x, x полагается переменной. С другой стороны, при рассмотрении алгебраического уравнения x2 - 1 = (x - 1) (x + 1), x полагается символом, фактически все уравнение символическое. Точно так же, как цитирование списка указывает Scheme на

обработку заключенной в скобки формы как списка, а не как приложения процедуры, так и цитирование идентификатора указывает Scheme обрабатывать идентификатор как символ, а не как переменную. Хотя символы обычно используются для представления переменных в символических представлениях уравнений или программ, символы также могут использоваться, например, в качестве слов в представлении предложений на естественном языке.

Числа и строки также могут быть цитируемыми.

```
'2 ⇒ 2
'2/3 ⇒ 2/3
(quote "Hi Mom!") ⇒ "Hi Mom!"
```

Данные переменные рассматриваются как константы, поэтому цитировать их нет необходимости.

4.3 ПРОЦЕДУРЫ ДЛЯ СПИСКОВ

Существуют две основные процедуры со списков: `car` и `cdr`. `car` возвращает первый элемент списка, а `cdr` возвращает остаток списка. Имена «`car`» и «`cdr`» получены из операций, поддерживаемых первым компьютером, на котором был реализован язык Lisp, IBM 704. Каждый из них требует непустого списка в качестве аргумента.

```
(car '(a b c)) ⇒ a
(cdr '(a b c)) ⇒ (b c)
(cdr '(a)) ⇒ ()
(car (cdr '(a b c))) ⇒ b
(cdr (cdr '(a b c))) ⇒ (c)
(car '((a b) (c d))) ⇒ (a b)
(cdr '((a b) (c d))) ⇒ ((c d))
```

Первый элемент списка часто называют «`car`» списка, а остальную часть списка часто называют «`cdr`» списка. `cdr` списка с одним элементом - `()`, пустой список.

Процедура `cons` составляет списки. Требуется два аргумента. Вторым аргументом обычно является список, и в этом случае `cons` возвращает список.

```
(cons 'a '()) ⇒ (a)
(cons 'a '(b c)) ⇒ (a b c)
(cons 'a (cons 'b (cons 'c '()))) ⇒ (a b c)
(cons '(a b) '(c d)) ⇒ ((a b) c d)
(car (cons 'a '(b c))) ⇒ a
(cdr (cons 'a '(b c))) ⇒ (b c)
(cons (car '(a b c))
      (cdr '(d e f))) ⇒ (a e f)
(cons (car '(a b c))
      (cdr '(a b c))) ⇒ (a b c)
```

Процедура cons создает пары. В свою очередь список - это последовательность пар; `cdr` каждой пары является следующей парой в последовательности.

В качестве примера, список `'(a, b, c, d)` модно записать как указано на рисунке 1.

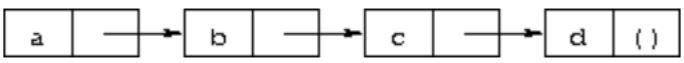


Рисунок 1

Код последней пары в правильном списке - пустой список. В противном случае последовательность пар образует неправильный список. Более формально,

пустой список является правильным списком, и любая пара, чей `cdr` является правильным списком, является правильным списком.

Неподходящий список печатается в записи с точечной парой, с точкой, предшествующей последнему элементу списка.

```
(cons 'a 'b) ⇒ (a . b)
(cdr '(a . b)) ⇒ b
(cons 'a '(b . c)) ⇒ (a b . c)
```

Из-за печатных обозначений пару, чей `cdr` не является списком, называют точечной парой. Однако даже пары, чьи `cdr` являются списками, могут быть записаны в виде записи с точечной парой, хотя принтер всегда выбирает правильные списки без точек.

```
'(a . (b . (c . ()))) ⇒ (a b c)
```

Процедура `list` похожа на `cons`, за исключением того, что она принимает произвольное количество аргументов и всегда создает правильный список.

```
(list 'a 'b 'c) ⇒ (a b c)
(list 'a) ⇒ (a)
(list) ⇒ ()
```

4.4 ПРИМЕНЕНИЕ ПРОЦЕДУР В SCHEME

рассмотрим применения процедур в форме `(procedure arg1 ... argn)`. Где `procedure` – это выражение, представляющее процедуру Scheme, а `arg1 ... argn` - это выражение, представляющее ее аргументы. Одной из возможных реализацией применения процедуры заключается в следующем.

- Найти описание *procedure*.
- Найти значение arg_1.

  …
- Найти значение arg_n.
- Применить *procedure* к значениям arg_1 ... arg_n.

Например, рассмотрим простую процедуру применения `(+ 3 4)`. Значение + - это процедура сложения, значение 3 - это число 3, а значение 4 - это число 4. Применение процедуры добавления к 3 и 4 дает 7, поэтому итоговый результат - 7.

Применяя этот процесс на каждом уровне, мы можем найти значение вложенного выражения `(* (+ 3 4) 2)`. Значение * – это процедура умножения, значение `(+ 3 4)`, которое мы можем определить как число 7, а значение 2 – это число 2. Умножив 7 на 2, мы получим 14, поэтому итоговый результат - 14.

Это правило работает для приложений процедур, но не для выражений `quote`, потому что подвыражения процедур оцениваются, а подвыражение выражений `quote` - нет. Применение выражения кавычки выполняется также, как и применение константных объектов. Значением выражения формы `(quote object)` является просто `object`.

4.5 ВЫРАЖЕНИЕ DEFINE

Для присваивания переменным значений используется спецификатор define.

```
(define sandwich "peanut-butter-and-jelly")
sandwich ⇒ "peanut-butter-and-jelly"
```

Для реализации данной процедуры, есть глобальный scope, в котором записаны все значения переменных, процедура `define` добавляет значения переменной или переназначает в случае, если переменной уже присвоено значение.

После применения `define` создается переменная.

```
(define x 10)

(define y 3)

(+ x y) ⇒ 13
```

4.6 ЛЯМБДА-ВЫРАЖЕНИЕ

Для создания новой процедуры можно использовать синтаксическую форму лямбда-выражения, в которой в качестве параметра указаны x и тело которого совпадает с выражением.

```
(lambda (x) (+ x x)) ⇒ #<procedure>
```

Общая форма лямбда-выражения

(lambda (*var* ...) $body_1$ $body_2$ ...)

Переменные var ... являются формальными параметрами процедуры, а последовательность выражений body1 body2 ... является ее телом.

После создания процедуры ее можно применить к переменным следующим образом:

```
((lambda (x) (+ x x)) (* 3 4)) ⇒ 24
```

Применение созданной процедуры не отличается от применения встроенных процедур.

Поскольку процедуры являются объектами, мы можем установить процедуру в качестве значения переменной и использовать процедуру более одного раза.

```
(define add (lambda (x y) (+ x y))  ⇒ ()
(add 4 3)  ⇒ 7
(add 2 4)  ⇒ 6
```

Процедура ожидает, что ее фактическим параметром список из двух чисел. В общем случае, фактическим параметром может быть любой вид объекта.

```
(define double-cons (lambda (x) (cons x x)) ⇒ ()
(double-cons 'a)) ⇒ (a . a)
```

В случае, если используется переменная, которая не определена в var ..., то значение переменной присваивается из области родителя объекта, аналогично тому, как это происходит в современных языках.

```
(define x 10) ⇒ ()
((lambda (y) (+ x y)) 7) ⇒ 17
```

4.7 ВЫРАЖЕНИЕ IF

Выражение `if` имеет форму `(if test confter alternative)`, где `confter` – возвращаемый объект, если `test = #t`, и `alternative` – возвращаемый объект, если `test ≠ #t`. В приведенном ниже выражении `test` равно `(<n 0)`, `confter` равно `(- 0 n)`, а `alternative` равна `n`.

```
(define abs
  (lambda (n)
    (if (>= n 0)
        n
        (- 0 n))))
```

`if` – синтаксическая форма, а не процедура. Рассмотрим следующий пример.

```
(define reciprocal
  (lambda (n)
    (if (= n 0)
        "oops!"
        (/ 1 n))))
```

Как можно заметить, если положить что `if` – процедура. Тогда перед применением процедуры должно быть выполнено вычисление всех аргументов, а это значит, что при обработке выражения `(/ 1 n)`, при условии, что n = 0, будет получена ошибка. Для того, чтобы обрабатывать данные ошибки вычисляется только нужно значение из confter и alternative.

Для работы с выражениями `if` используется процедуры возвращающие #t или #f:

```
(< -1 0) ⇒ #t
(> -1 0) ⇒ #f

(not #t) ⇒ #f
(or #f) ⇒ #f
(or #f #t) ⇒ #t

(and #f #t) ⇒ #f

(pair? '(a . c)) ⇒ #t

(boolean? #f) ⇒ #t
```

С помощью выражений `if`, `define` и `lambda` можно строить сложные программы:

```
(define slow-add
```

```
    (lambda (x y)
        (if (= x 0)
        y
        (slow-add (- x 1) (+ y 1)))))
(slow-add 6 6) ⇒ 12
```

Используя данные конструкции, можно возвращать ошибки:

```
(define reciprocal
  (lambda (n)
    (if (and (number? n) (not (= n 0)))
        (/ 1 n)
        (assertion-violation 'reciprocal
          "improper argument"
          n))))

(reciprocal .25) ⇒ 4.0
(reciprocal 0) ⇒ exception in reciprocal: improper arg
ument 0
(reciprocal 'a) ⇒ exception in reciprocal: improper ar
gument a
```

4.8 РЕКУРСИИ

Рекурсивная процедура - это процедура, которая применяется сама по себе. Вышеописанные выражения Scheme позволяют строить рекурсии:

```
(define goodbye
  (lambda ()
    (goodbye)))
(goodbye) ⇒
```

В данном случае, процедура не закончит вычисления, так как напрямую ссылается на себя же.

К примеру, для вычисления длинны списка можно воспользоваться следующей рекурсивной функцией [6]:

```
(define length
  (lambda (ls)
    (if (null? ls)
        0
        (+ (length (cdr ls)) 1))))
(length '(a b)) ⇒ 2
```

ГЛАВА 5. ОРГАНИЗАЦИЯ ИНТЕРПРЕТАЦИИ ЯЗЫКА SCHEME

5.1 ТОКЕНИЗАЦИЯ

Токенизатор принимает на вход код программы и возвращает последовательность токенов.

Токены представляют собой следующий вид:

- ConstantToken – число, может начинаться на знак +, -, а также цифры, все остальные символы цифры
- BracketToken – символ скобки (или)
- QuoteToken – символ '
- DotToken – символ “.”
- SymbolToken - последовательность символов, начинается с символов `[a-z<=>*#]` и может дополнительно содержать внутри знаки `-`, `+`, `?`. Кроме этого, есть символы-исключения `+`, `-` и `*`.

Токенизатор начинает читать символы кода слева направо и разбивать их на токены. Если текущий символ пробел, то токенизатор заканчивает читать токен и переходит к следующему токену.

Пример:

```
'(+ 4 -5) ->  QuoteToken BracketToken(open)  SymbolToken(+)
ConstantToken(4) ConstantToken(-5) BracketToken(close)
```

5.2 ПАРСИНГ

Парсер принимает последовательность токенов и строить по ним синтаксическое бинарное дерево. Для простоты будем называть детей правым сыном и левым сыном.

У каждой вершины дерева может иметь следующий из трех типов:

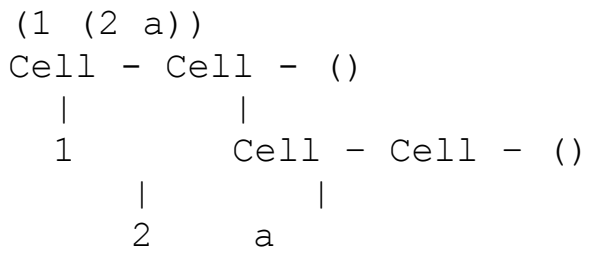
- Пустой список
- Symbol – имеет внутри себя SymbolToken
- Number – имеет внутри себя ConstantToken
- Cell – имеет внутри себя ссылки на двух сыновей.

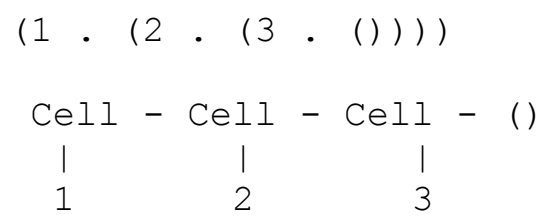
Парсер читает токены слева на право.

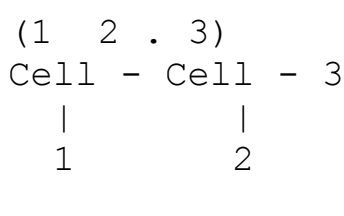
- Если он в данный момент читает SymbolToken или ConstantToken, то он создает в соответствии Symbol или Number с данным значением.
- Если парсер в данный момент читает QuoteToken, то он создает вершину типа Cell слева к которому присоединяет SymbolToken = “quote”, а справа ту вершину, которую получит при прочтении далее.
- Если парсер в данный момент читает DotToken, то он создает вершину типа Cell и присоединяет последнюю полученную вершину слева, а вершину, которую он получит при прочтении далее, справа
- Если парсер в данный момент читает BracketToken “(“, то он дочитывает токены до токена BracketToken “)“, после чего создает последовательность Cell, где данные вершины последовательно соединены через правых сыновей и присоединяет к соответствующему Cell слева значения посчитанных вершин.

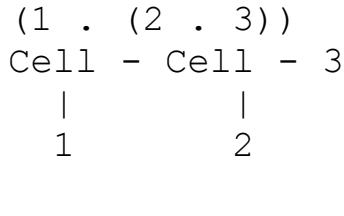
Примеры работы парсера:

```
(1  2  3)

 Cell - Cell - Cell - ()
  |       |       |
  1       2       3

(' (2  3))

 Cell - Cell - Cell - ()
  |       |       |
 quote    2       3
```


(1 (2 a))
Cell - Cell - ()
1
Cell - Cell - ()
2
a


(1 . (2 . (3 . ())))
Cell - Cell - Cell - ()
1
2
3


(1 2 . 3)
Cell - Cell - 3
1
2


(1 . (2 . 3))
Cell - Cell - 3
1
2

5.3 АЛГОРИТМ EVAL

Окружение вершины, это все заданные переменные в данной вершине с их значениями, также в окружении присутствует ссылка на родительское окружение. Процедура Get принимает строку, и отдает значение, если строка находится в окружении, иначе рекуррентно вызывает процедуру Get в родительском окружении, для поиска переменной в вершине родителя, это позволяет, использовать переменные, которые заданы в более высоком уровне.

Для интерпретации алгоритм eval рекуррентно запускается из вершины дерева.

- Если вершина является `Number`, то возвращается целое число, записанное в вершине.
- Если вершина является `Symbol`, то возвращается значение процедуры Get от значения строки `Symbol` в текущем окружении.
- В случае, если вершина типа `Cell`, то тогда левый сын вершины должен быть выражением, и в соответствии с инструкцией к выражению вычисляются процедура `eval` в соответствующих значениях из списка правого сына и после подставляются в выражение. В случае, если в инструкции к выражение требует модификации переменной, то вызывается процедура `Set`. Если исходная процедура имеет вид `define`, то создается новое окружение и в качестве родителя присваивается текущее окружение, а также подставляет переменную и ее значение в новое окружение.

Процедура Set ищет переменную в окружении, если она есть, то она модифицирует переменную, иначе вызывается рекуррентно процедура Set у родительского окружения.

В итоге, процедура eval возвращает вершину некоторого дерева, созданного этой процедурой, к которой применяется процедура Assemble. В свою очередь, Assemble выполняет обратное преобразование из дерева в возвращаемый ответ.

ГЛАВА 6. ДЕТАЛИ РЕАЛИЗАЦИИ ИНТЕРПРЕТАТОРА НА ЯЗЫКЕ C++

Для реализации интерпретатора использовалось функциональность объектно-ориентированного программирования C++.

Для реализации токенайзера использовалась структура `std::variant`, для возвращения последовательности однородных токенов:

```
typedef std::variant<QuoteToken, DotToken, BracketToken, ConstantToken, SymbolToken> Token;
```

Для парсинга и интерпретации дерева разбора использовался абстрактный класс `Object`, с виртуальными методами `Apply` и `Eval`.

Парсер возвращал класс один из следующих классов `Number`, `Symbol` и `Cell`, которые в свою очередь наследованы от `Object`.

Для процедуры eval применялся виртуальные методы `Apply` и `Eval`. В Apply была записанна инструкция для применения выражения, а метод `Eval` применялся в качестве процедуры `eval`.

Также в классе `Object` была целочисленная переменная, отвечающая за тип класса, которая определялась в наследниках. Она была для того, чтобы отличать выражения от переменных и констант.

Для решения проблема утечки памяти, был реализован алгоритм сборки мусора. Проблема была связана с природой окружения. Окружение ссылается на верхние окружение в свою очередь вершины ссылаются на окружения и наоборот, из-за этого возникал цикл из ссылок.

ЗАКЛЮЧЕНИЕ

В ходе данного курсового проекта:

- Были изучены основные концепции организации интерпретаторов;
- Проанализированы и описаны диалекты языка Lisp;
- Проанализирован синтаксис языка Scheme;
- Реализован интерпретатор диалекта Scheme на языке C++;
- Проанализированы возможности языка C++ для реализации интерпретатора.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Левин М., LISP 1.5 programmer's manual. 2-е изд., MIT, 1979 г - 107 с.
2. Абельсон Г., Structure and Interpretation of Computer Programs, MIT, 2006 г – 608 с.
3. Кеннек К., Les Langages Lisp, InterEditions, 1994 г – 480 с.
4. Холден Д. Build Your Own Lisp = Создай свой собственный LISP. 2017 – 205 с.
5. Лафоре Р. Объектно-ориентированное программирование в C++ – 4-е издание. СПб. Питер, 2004 – 924 с.
6. Дибвиг К., The Scheme Programming Language, 4-е изд., MIT, 2009 г – 210 с.

ПРИЛОЖЕНИЕ

Приложение реализованного интерпретатора выложено на GitHub в свободном доступе:

https://github.com/PaliukhA/Scheme-interpreter

Также тесты к приложению доступны в папке test:

https://github.com/PaliukhA/Scheme-interpreter/tree/master/test