Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Теоретическая информатика и компьютерные технологии»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:
«Минимальная расширяемая и встраиваемая реализация языка семейства Лисп»

Студент ИУ9-81	_____ (Подпись, дата)	В. В. Мельников
Руководитель ВКР	_____ (Подпись, дата)	А. В. Дубанов
Консультант	_____ (Подпись, дата)	_____ (И.О.Фамилия)
Консультант	_____ (Подпись, дата)	_____ (И.О.Фамилия)

2020 г.

АННОТАЦИЯ

Тема этой работы – «Минимальная расширяемая и встраиваемая реализация языка семейства Лисп».

В результате был разработан минималистичный язык с широкими возможностями в метапрограммировании, с возможностью встраивания в программы, написанные на других языках программирования, а также с большими возможностями по своему расширению.

Работа состоит из 61 страницы. В работе использовано 11 источников информации. Содержит 9 листингов, 41 таблицу и 6 рисунков.

Содержание

ВВЕДЕНИЕ

В далеком 1960 году американский математик и информатик Джон Маккарти для всего информационного мира сделал по значимости примерно то же самое, что смог сделать в свое время Евклид для геометрии – ученый показал, как при помощи примитивной скобочной нотации можно создать полноценный язык программирования, способный сам себя дополнять и изменять [1]. Семейство таких языков получило название «Lisp» (в дальнейшем будет использоваться транслитерация на русский язык – «Лисп») – от английского List Processing. Идея состоит в том, чтобы как для кода, так и для данных использовать простые структуры данных – списки.

Несмотря на свою долгую историю, Лисп до сих пор остается актуальным. Многие современные языки программирования заимствуют его особенности [2], однако все еще нет языков, которые могут с такой же свободой расширять свои возможности [3].

Целью данной работы является выявление и реализация минимального описания ядра, при котором язык остается полностью расширяемым.

Для достижения цели работы были поставлены следующие задачи:

- изучение особенностей языков семейства Лисп;
- разработка минималистичного и расширяемого языка;
- реализация его интерпретатора;
- разработка и реализация интерфейса вызова из других программ (написанных на других языках).

1. Общие теоретические сведения

1.1. S-выражения и списки

Единство кода и обрабатываемых им данных – то, что выделяет Лисп среди огромного множества языков программирования. По факту, код является теми же данными.

Первопричины такого решения лежат в процессе разработки этого языка [4]. Изначально программы на нем предполагалось записывать в виде M-выражений. M-выражения – способ записи выражений, который близок к их математической записи (примеры в таблице 1). При использовании этой нотации для различия имена функций и переменных записывались в нижнем регистре, а символы в верхнем регистре (в связи с этим в некоторых языках семейства Лисп символы записываются либо только заглавными буквами, либо заглавные буквы используются для внутреннего представления символов, что делает их регистронезависимыми). Для реализации этой нотации разработчики намеревались сконструировать компилятор, и считалось, что на его написание уйдет много лет. Его разработка началась с ручной компиляции различных функций, необходимых для обеспечения среды. В качестве нотации для описания данных, с которыми работал бы язык, был придуман синтаксис S-выражений (именно он и определяет списки, использование которых обусловлено архитектурными особенностями машины, для которой велась разработка). Чтобы показать лаконичность реализации универсальной функции eval на Лиспе, потребовалась возможность записи M-выражений в форме S-выражений. Примеры в таблице 2. Стив Рассел (один из разработчиков языка) обнаружил, что eval может служить в качестве интерпретатора Лиспа, записанного при помощи S-выражений. Довольно быстро (по сравнению с предполагаемой многолетней разработкой компилятора) закодировав эту функцию, разработчики получили интерпретируемый язык программирования.

Впоследствии сообщество программистов, которым полюбилась запись программ в виде S-выражений, сильно разрослось. В итоге проект по реализации языка, использующего M-выражения, так и остался в подвешенном состоянии: как не был доведен до конца, так и не был заброшен. Хоть сейчас и существуют реализации языков, имеющих синтаксис M-выражений, но широкого распространения эта нотация не получила.

Можно сказать, что Маккарти не просто изобрел новый язык программирования, а открыл кардинально иную точку зрения на решение многих задач.

Таблица 1 – примеры M-выражений.

Математическая запись	M-выражение
$(1, 2, 3)$	`[1;2;3]`
$f(x, y)$	`f[x;y]`
$square: x \mapsto xx$	`square[x] = x*x`
$\begin{cases} -x, \text{если } x < 0 \\ x, \text{иначе} \end{cases}$	`[x<0 -> -x; T -> x]`

Таблица 2 – соответствие M-выражений S-выражениям.

M-выражение	S-выражение
`[1;2;3]`	`(quote (1 2 3))`
`f [x;y]`	`(f x y)`
`square[x] = x*x`	`(define square (lambda (x) (* x x)))`
`[x<0 -> -x; T -> x]`	`(if (< x 0) (- x) x)`

1.2. Особенности Лиспа

Некоторые важные идеи, которые впервые появились в Лиспе и впоследствии распространились на другие языки:

- Условные выражения – Лисп был первым языком, который использовал условные высказывания в том виде, в котором они представлены почти во всех современных языках (if-else конструкции, в Фортране подобные функции требовали вычисления обоих ветвей перед возвратом [5]).
- Функции как объекты – в отличие от всех других существовавших на тот момент языков программирования, Лисп позволял работать с функциями как с остальными объектами языка, такими как числа и строки. Теперь функции могли сохраняться в переменных, передаваться как аргументы, а также быть результатом выполнения других функций.
- Рекурсия – благодаря новому виду условных высказываний, стала доступна реализация рекурсии.

1.3. Основы

В Лиспе существует два основных типа объектов – атом и точечная пара. Из атомов можно выделить символы, числа и Nil. Nil – ложь в логических выражениях. Символы – это строки, которые обозначают другие объекты (можно провести аналогию с именами переменных). Точечная пара – это кортеж из двух объектов. Элементами точечной пары могут быть как атомы, так и точечные пары [6].

Список – это особый вид точечной пары. Чтобы точечная пара являлась списком, необходимо, чтобы ее второй элемент («хвост» списка) являлся либо списком, либо Nil’ом. Nil также может интерпретироваться как пустой список.

Как правило, списки записывают в скобках, перечисляя его элементы через пробел, а обычные точечные пары – как два элемента в скобках, разделенных точкой (отсюда и название - точечные). Пример предоставлен в таблице 3.

Таблица 3 – способы записи пар.

Точечная нотация	Альтернативы
(2 . 3)	-
(2 . (4 . (5 . nil)))	(2 4 5) или (2 4 5 . nil)
((4 . (5 . nil)) . (7 . (8 . 9)))	((4 5) (7 8 . 9))

Базовый набор функций для полноценной работы со списками и парами в Лиспе [7]:

- quote – цитирование. Так как весь код записывается в виде списков, для получения самого списка без вычисления используется данная функция. Также используется для получения символа вместо его отображения в объект.
- eval – выполнение аргумента. Совершает работу, противоположную функции quote.
- car – возвращает первый элемент пары.
- cdr – возвращает второй элемент пары.
- cons – объединяет два элемента в пару.

2. Спецификация реализуемого языка LispXS

2.1. Система типов

Язык имеет динамическую неявную типизацию. Поддерживаются следующие типы:

- Number – числа;
- Symbol – лисповые символы;
- Pair – непустой список (от пар, которые не являются списками, было решено отказаться);
- Nil – пустой список, а также ложь для логических выражений.

Было решено оставить именно этот набор, так как символы позволяют работать с собой как со строками, поэтому можно обойтись без выделения отдельного типа под них, что подходит для достижения одной из целей данной работы.

2.2. Код и выражения

Как и было описано в разделе 1, все – это выражения: программа является выражением, данные являются выражениями, а также результат выполнения любого выражения – выражение. Программа – это выражение, которое состоит из выражений и возвращает результат работы последнего выражения. Если число выражений равно нулю, то возвращается Nil. Общая схема вычисления выражений (рисунок 1):

- Если выражение является символом, то оно возвращает выражение, связанное с этим символом (из ближайшей области видимости, содержащей этот символ).

- Если выражение является парой (например, (+ (- 2 3) (+ 8 9))), то сначала выполняются все (за исключением тех случаев, когда результатом работы первого выражения является одна из функций ‘quote’, ‘throw’, ‘define’, ‘if’, ‘or’, ‘and’ или макрос) элементы списка (из примера: (+ -1 17)), после чего, если первый элемент является функцией, замыканием или макросом – выполняет его, подавая на вход остальные элементы списка (из примера: 16), иначе возвращает ошибку.

- В ином случае возвращает себя же.

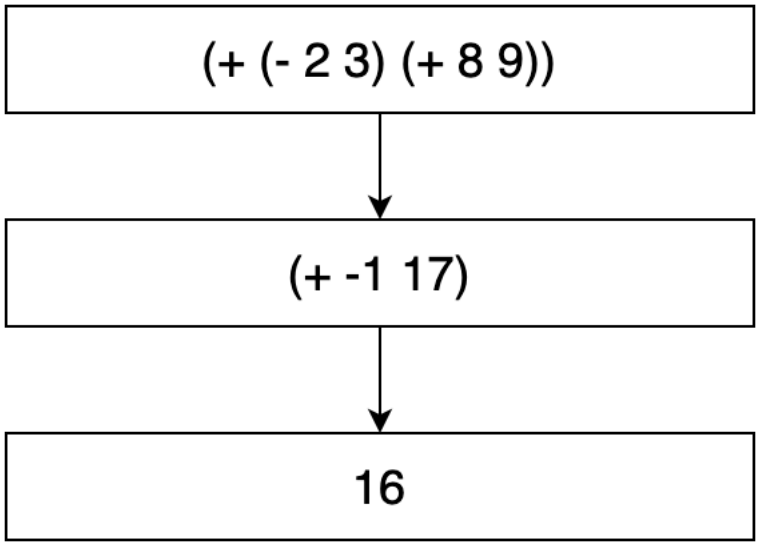

Рисунок 1 – пример вычисления выражений, являющихся парами.

2.3. Области видимости

В начале работы программы существует одна область видимости – корневая. Она содержит в себе определения всех функций, nil (связан с Nil) и T (связан с собой же). Новые области видимости создаются при вызове замыканий и макросов. Родительская область видимости определяется в момент определения замыкания/макроса. Эта область существует, пока выполняется

функция либо пока макрос генерирует код. Когда идет запрос выражения, связанного с символом, сначала идет поиск в текущей области видимости, затем в родительской и так далее. Если символа не нашлось и в корневой области, то возвращается ошибка. Функция define используется для определения переменной в текущей области, set! – для переопределения существующей переменной в ближайшей области.

Например, программа из листинга 1 вернет число 5.2, так как во внутренней области видимости доступны переменные a и c, переменная b из замыкания func1 (переменная a из func1 скрыта переменной a из внутренней области) и все переменные из корневой области (рисунок 2).

Листинг 1.

```
(define func1 (lambda (a b)
  (+ a ((lambda (a c)
    (/ a c b)) b a))))

(func1 5 3)
```

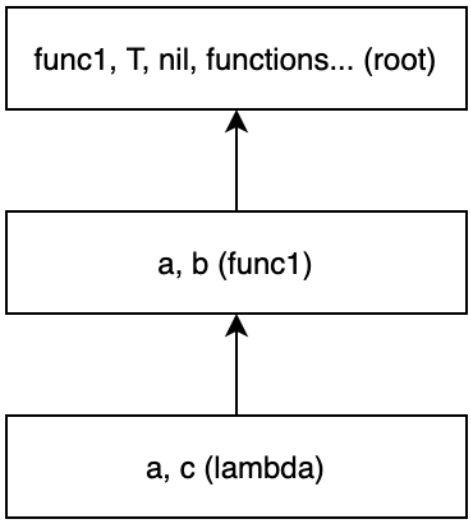

Рисунок 2 – схема областей видимости.

2.4. Макросы

Благодаря тому, что в языке нет различий между кодом и данными, то можно писать программы, которые пишут другие программы. Макросы являются одним из способов, которые позволяют это делать (второй способ – функция ‘eval’, раздел 6.2). Работа с макросами происходит в три этапа:

1. Определение макроса. Чтобы определить новый макрос, необходимо воспользоваться функцией ‘defmacro’ (раздел 6.6). В этот момент определяется родительская область видимости для второго этапа.

2. Вызов макроса и генерация кода. В момент вызова макроса происходит создание новой области видимости, передача в нее аргументов (те аргументы, перед которыми стоит запятая, перед передачей телу макроса вычисляются в текущей области видимости). Затем идет выполнение тела макроса. Результирующим кодом будет считаться результат выполнения последнего выражения.

3. Выполнение сгенерированного кода. Результат выполнения макроса выполняется в той области видимости, из которой он был вызван.

2.5. Расширение языка

Язык имеет широкие возможности по своему расширению. Чтобы изменить язык, не трогая при этом исполняемый код программы, можно воспользоваться ‘prelude’ файлом.

Перед выполнением программы интерпретатор найдет файл с названием ‘prelude’ в текущем рабочем каталоге и выполнит его содержимое.

Например, в этом файле можно реализовать такие функции, как list или map для их использования в программе.

2.6. Перехват и создание ошибок

Если в программе возникает ошибка (например, деление на ноль), то создается объект ‘Fatal’, который падает вниз по стеку вызовов и собирает информацию о местоположении ошибки. Когда достигается дно стека, вся информация выводится в поток вывода ошибок и программа завершается. Чтобы перехватить ошибку, можно воспользоваться оператором ‘catch’ (структура в листинге 2). Он ловит ошибку и ищет первый подходящий тег. Если таковой найден (тег является префиксом к тегу ошибки или это тег ‘default’), то выполняет его тело и возвращает результат. Иначе – бросает ошибку дальше вниз по стеку.

Также ошибка может быть создана при помощи функции ‘throw’ (структура на листинге 3). Если подходящий тег оператора ‘catch’ не содержит тела, то возвращает результат вычисления тела ‘throw’. Если же ‘throw’ также не содержит тела, то возвращается Nil.

Листинг 2 – структура оператора ‘catch’.

```
(catch body_that_can_throw_an_error (tag1 body) (tag2 body) ...)
```

Листинг 3 – структура оператора ‘throw’.

```
(throw 'tag body)
```

2.7. Расширяемость языка

В ядре языка LispXS нет многих привычных для языков семейства Lisp функций, однако они легко реализуются средствами самого языка. Например, в таблице 4 представлены реализации функций ‘list’, ‘apply’ и ‘map’.

Таблица 4 – реализация ‘list’, ‘apply’ и ‘map’.

Пример	Результат

`(define list (lambda args args))`
 	
`(list 2 3 (+ 1 3))`
	`(2 3 4)`

`(defmacro apply (f ,args) (cons f args))`
 	
`(apply - '(4 5 6))`
	`-7`

`(define list (lambda args args))`
 	
`(defmacro map (f1 ,args1)`
`  (define helper (lambda (f args)`
`    (if args`
`      (cons`
`        (list f (list quote (car args)))`
`        (helper f (cdr args))))))`
`  (cons list (helper f1 args1)))`
 	
`(map – ‘(-4 9 -2))`
	`(4 -9 2)`

Реализация функции импорта библиотек показана в таблице 5.

Таблица 5 – реализация ‘import’.

Пример	Результат	Файл ‘path_to_file’

(define list (lambda args args))
(defmacro map (f1 ,args1)
  (define helper (lambda (f args)
  (if args
  (cons
  (list f (list quote (car args)))
  (helper f (cdr args))))))
  (cons 'list (helper f1 args1)))
(defmacro import (path)
  (list 'map 'eval (list 'load path)))
(import 'path_to_file)
(++ 7)
	8	(define ++ (lambda (a) (+ a 1)))
		
		(define -- (lambda (a) (- a 1)))

По умолчанию в языке можно переопределить любые символы, даже те, которые в языке заняты изначально (такие, как ‘+’ или ‘eval’). Чтобы избежать переопределения определенных символов, можно воспользоваться макросами. Пример в таблице 6.

Таблица 6 – запрет на переопределение стандартной функции ‘+’.

Пример	Результат

(define list (lambda args args))
((lambda ()
  (define temp set!)
  (defmacro settemp (sym val)
  (if (= sym '+) (throw '|couldn't redefine '+' func|))
  (list temp sym (eval val)))
  (set! set! settemp)))
(set! + >)
(+ 3 2)
	ERROR

Списки являются неизменяемыми. Способ эмуляция изменяемости списков, а также их индексирование приведены в таблице 7.

Таблица 7 – индексируемые и изменяемые списки.

Пример	Результат
(define list (lambda args args)) (define <= (lambda (a b) (or (< a b) (= a b)))) (define get (lambda (l n) (if (= n 0)	new

(car l) (get (cdr l) (- n 1))))) (defmacro setl! (l pos val) (define pos (eval pos)) (define val (eval val)) (define mut (lambda (l i v) (if (<= i 0) (cons v (cdr l)) (cons (car l) (mut (cdr l) (- i 1) v))))) (list 'set! l (list mut l pos (list 'quote val)))) (define lst '(s trtrt 5 laa kooo r 4)) (setl! lst 3 'new) (get lst 3)	

При помощи предыдущего расширения можно реализовать определение изменяемых структур. Пример в таблице 8. Также в этом примере показываются макросы, которые пишут макросы.

Таблица 8 – определение определения изменяющихся структур.

Пример	Результат
(define list (lambda args args)) (defmacro apply s (define f (car s)) (define args1 (car (cdr s))) (list 'eval (list 'cons f args1))) (define list (lambda args args)) (define pow2 (lambda (x) (* x x))) (define get (lambda (l n) (if (= n 0) (car l) (get (cdr l) (- n 1))))) (define <= (lambda (a b) (or (< a b) (= a b)))) (define sqrt (lambda (x) (define findi (lambda (i) (if (<= (* i i) x) (findi (+ i 1)) (- i 1)))) (define i (findi 0)) (define p (/ (- x (* i i)) (* 2 i))) (define a (+ i p)) (- a (/ (* p p) (* 2 a))))) (defmacro setl! (l pos val) (define mut (lambda (l i v) (if (<= i 0) (cons v (cdr l)) (cons (car l) (mut (cdr l) (- i 1) v))))) (list 'set! l (list mut l pos (list 'quote val)))) (defmacro defstruct args (define structname (car args))	10

```
  (define funcname (lambda (str) (+ structname '-
str)))
  (define methods (lambda (args i)
    (if (not args)
      nil
      (cons
        (list 'define (funcname (+ 'get- (car args)))
(list 'lambda '(s) (list 'get 's i)))
        (cons
          (write (list 'defmacro (funcname (+ 'set-
(car args))) '(s v) (list 'list ''setl! 's i 'v)))
          (methods (cdr args) (+ i 1)))))))
  (cons
    'begin
    (cons
      (list 'define (funcname 'new) (list 'lambda (cdr
args) (cons 'list (cons (list 'quote structname) (cdr
args)))))
      (cons
        (list 'define (funcname '?) (list 'lambda '(s)
(list '= '(car s) (list 'quote structname))))
        (methods (cdr args) 1)))))
(defstruct point x y)
(define dist (lambda (p1 p2)
  (if (not (and (point-? p1) (point-? p2))) (throw
'|points expected|))
  (sqrt (+ (pow2 (- (point-get-x p2) (point-get-x p1)))
(pow2 (- (point-get-y p2) (point-get-y p1)))))))
(define pt1 (point-new 4 2))
(define pt2 (point-new -2 6))
(point-set-y pt1 -2)
(dist pt2 pt1)
```

2.8. Набор стандартных функций языка

Было принято решение оставить следующий набор функций:

- quote, eval – цитирование и выполнение выражений;
- define, set! – определение переменных, а также деструктивное присваивание (для возможности писать программы в императивной парадигме);
- lambda – определение замыканий;
- defmacro – определение макросов;

- if, or, and, not, =, >, < – условный оператор, а также логические операции и операции сравнения;
- catch, throw – обработка ошибок;
- write, read, load – операции ввода/вывода
- begin – объединение нескольких выражений в одно для их последовательного вычисления;
- cons, car, cdr – работа со списками;
- symbol->number, number->symbol, symbol?, number?, pair? – работа с типами (конвертация, предикаты);
- len, +, -, *, / – функции для работы с символами и числами.

Этот набор позволяет реализовать те возможности, которые есть во многих языках семейства Лисп.

Полное описание функций приведено в разделе 6 настоящей работы.

3. Лексический анализ

3.1. Описание анализатора

Перед тем, как можно будет составить абстрактное синтаксическое дерево, необходимо разделить входящий поток символов (литер) текста программы на лексемы (токены) [8].

Для описанного выше языка требуются возможность распознавать следующий набор лексем:

- Число – может быть представлено в следующих формах (число -4): ‘-4’, ‘-4.0’, ‘-0.04e2’, ‘-4000e-3’, ‘-12/3’. Схема распознавателя для чисел приведена на рисунке 3.
- Символ – последовательность кодовых единиц, окруженных знаками вертикальной черты (которые можно опустить в том случае, если внутри не содержится escape-последовательностей, пробельных символов, знаков скобок, символы не начинаются со знаков запятой, кавычки и точки с запятой, а также не являются записью числа). Поддерживаются следующие escape-последовательности: ‘\n’ – знак переноса строки, ‘\\’ – экранирование обратной косой черты, ‘\t’ – знак табуляции, ‘\|’ – экранирование вертикальной черты (без экранирования распознавание символа закончится раньше, чем необходимо). Может быть представлен в следующих форматах (показаны разные записи эквивалентных символов): ‘symbol’ и ‘|symbol|’, ‘\n’ и ‘|\\n|’. Схема распознавателя для чисел приведена на рисунке 4.
- Левая скобка
- Правая скобка

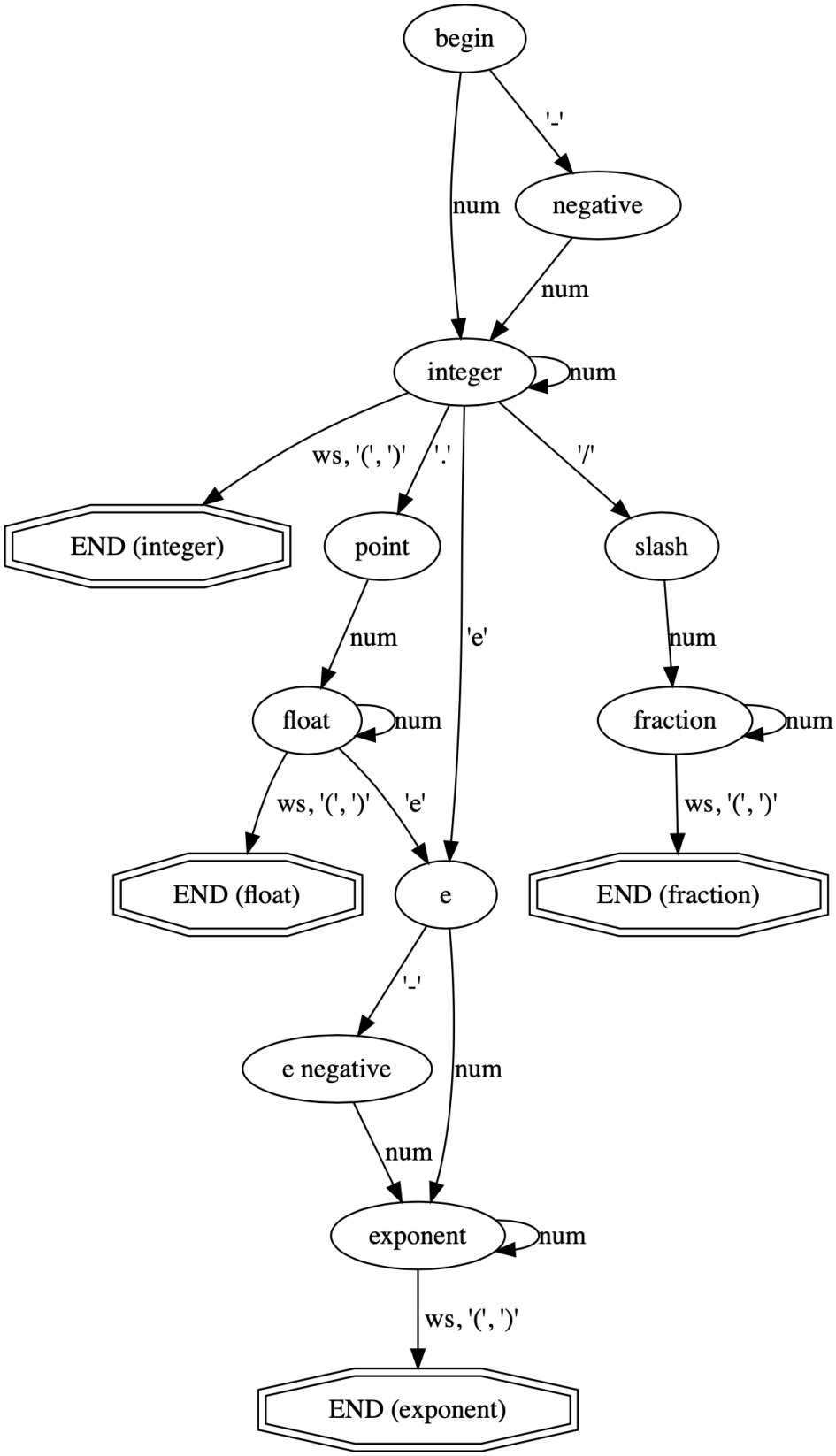

Рисунок 3 – схема лексического распознавателя для чисел. num – цифра, ws – пробельный символ. Если подается символ, который не представлен на схеме

(опущено для наглядности), то анализатор переходит в состояние symbol распознавателя символов (рисунок 4).

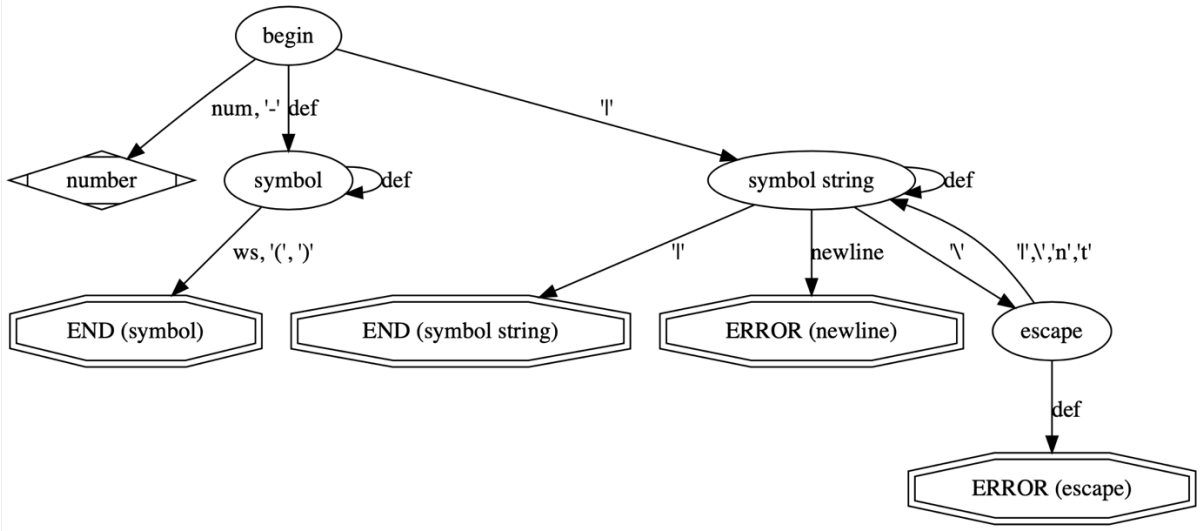

Рисунок 4 – схема лексического распознавателя для символов. num – цифра, ws – пробельный символ, newline – перенос строки, def – по умолчанию. Вершина number обозначает переход анализатора в состояние negative или integer распознавателя чисел (рисунок 3).

- Кавычка
- Запятая
- Также для возможности оставлять комментарии игнорируется весь код, начинающийся с точки с запятой (при этом эта литера не должна содержаться в какой-либо лексеме) и до знака переноса строки.

3.2. Реализация лексера

Для представления в памяти используется структура с двумя полями (листинг 4):

- text – срез из кодовых строчек текста входящей программы;
- coords – координаты последней обработанной литеры.

Листинг 4 – структура состояния лексера.

```
type Lexer struct {
    text   []rune
    coords Coords
}
```

Структура, представляющая координаты лексемы (листинг 5), содержит три поля:

- Cursor – номер литеры в тексте;
- Line – номер строчки (подстрока, не содержащая символов переноса строки и не являющаяся правильной подстрокой другой строчки), в которой содержится литера (нумерация начинается с единицы);
- Column – номер литеры в строчке (нумерация начинается с единицы).

Листинг 5 – структура координат.

```
type Lexer struct {
    Cursor, Line, Column int
}
```

Последние два поля используются для формирования сообщений о ошибках в ходе анализа.

Для начала работы с лексером необходимо вызвать функцию NewLexer – она принимает на вход исходный текст программы и возвращает указатель на структуру с лексером. В целях упрощения алгоритма лексического анализа перед сохранением текста программы к нему добавляется символ переноса строки. Далее необходимо вызывать метод NextToken – он возвращает следующий

обработанный токен и ошибку. После того, как будут обработан весь текст программы, метод будет возвращать токены, обозначающие конец файла.

Для токена была разработана структура со следующими полями (листинг 6):

- Coords – координаты последней литеры токена (фигурируют в сообщениях об ошибках, происходящих в ходе синтаксического анализа);
- Tag – тип лексемы (число, символ, левая скобка, правая скобка, кавычка, запятая или конец файла);
- String – строковое представление (только для символов);
- Number – численное представление (только для чисел).

Листинг 6 – структура токена.

```
type Token struct {
    Coords Coords
    Tag    int
    String string
    Number float64
}
```

Информация об ошибке записывается в структуру с двумя полями:

- Coords – координаты кодовой, на которой возникло ошибочное поведение;
- Message – информация об ошибке.

4. Синтаксический анализ

4.1. Грамматика

Для представления абстрактного синтаксического дерева необходимо сопоставить последовательность лексем с формальной грамматикой языка. Эту задачу выполняет синтаксический анализ [9].

```
PROGRAM    ::=  INNER eof
LIST       ::=  ( INNER )
INNER      ::=  ELEM INNER | .
ELEM       ::=  ' ELEM | , ELEM | number | symbol | LIST
```

Рисунок 5 – формальная грамматика языка.

Так как язык является членом семейства Лисп, грамматика языка является довольно простой. Она представлена на рисунке 5. В качестве терминальных символов грамматики выступают токены, которые можно получить после проведения лексического анализа. Нетерминальных же символов всего четыре:

- PROGRAM – корневой символ, обозначает код программы полностью;
- LIST – список (точечная запись списков не была реализована в связи с тем, что в языке не поддерживаются пары, которые не являются списками);
- INNER – «внутренность» списка (или, по-другому, список без скобок);
- ELEM – элемент списка.

В связи с простотой грамматики из двух основных стратегий работы синтаксического анализатора (далее парсера) – сверху-вниз и снизу-вверх выбор был сделан в пользу первой. При таком подходе построение дерева начинается с

корневого узла в сторону листьев (пример на рисунке 6). Также грамматика является LR(1), так как существует распознающий ее анализатор, который читает слева направо и по одной лексеме может определить, какому нетерминальному символу она принадлежит.

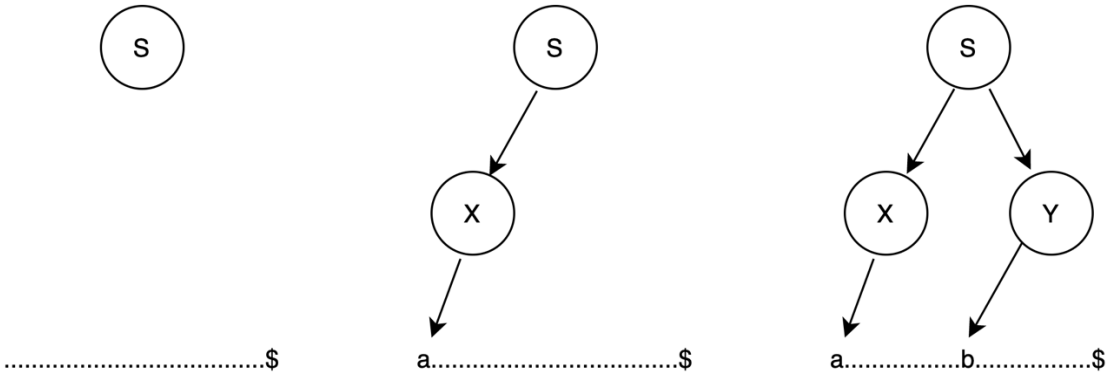

Рисунок 6 – синтаксический разбор сверху вниз.

4.2. Реализация парсера

Структура парсера состоит из следующих полей (листинг 7):

- curToken – последний полученный из лексера токен;
- lexer – лексер.

Листинг 7 – структура состояния парсера.

```
type Parser struct {
    curToken *lexer.Token
    lexer    *lexer.Lexer
}
```

Чтобы создать новый объект парсера, необходимо воспользоваться функцией NewParser - она принимает на вход текст программы и возвращает объект парсера. Метод Parse этого объекта возвращает построенное абстрактное синтаксическое дерево программы. Дерево представляет собой выражение, подробнее в разделе 5.1. Для построения этого дерева используется метод

рекурсивного спуска: для каждого нетерминального символа была разработана и реализована функция, которая разбивает данный нетерминальный символ на составляющие и генерирует соответствующее ему выражение (таблица 9) путем вызова других таких функций.

Таблица 9 – соответствие нетерминальных символов выражениям.

Нетерминальный символ	Результат работы анализатора
PROG	Результат из INNER
LIST	Результат из INNER
INNER	Nil или Pair (Результат из ELEM, Результат из INNER)
ELEM	Цитированный результат из ELEM, Number, Symbol или Результат из List

Ошибка, возникающая при синтаксическом анализе, имеет следующую структуру:

- Got – код распознанного символа;
- want – код символа, который ожидался:
- message – сообщение;
- coords – координаты лексемы.

5. Выполнение кода LispXS

5.1. Выражения

Для выражений был создан отдельный тип – Expr. Поля структуры:

- Type – тип выражения (число, символ, пара, пустой список, функция, замыкание, макрос или ошибка);
- String – строковые данные;
- Number – числовые данные;
- car – первый элемент пары;
- cdr – второй элемент пары;
- ParentVars – захваченное для замыканий и макросов окружение;

Выражение может представлять собой как данные, так и сам код. Абстрактное синтаксическое дерево также является этим выражением. Так как получение высокой производительности не является целью данной работы, то было принято решение о том, что внутреннее представление кода также будет выражением.

5.2. Интерпретатор

Объект интерпретатора представляется структурой со следующими полями (листинг 8):

- callStack – стек вызовов;
- dataStack – стек данных (в него кладутся результаты вычислений выражений);
- control – код, который выполняется в текущий момент времени;

- argsNum – количество вычисленных выражений на данный момент в списке;
- mod – модификаторы вычисления выражений, пояснения в разделе 5.4;
- varsEnvironment – текущее окружение с переменными;
- stdout – поток вывода;
- stderr – поток ошибок;
- stdin – поток ввода.

Листинг 8 – структура состояния интерпретатора.

```
type interpreter struct {
    callStack stackCall
    dataStack stackExpr
    control   *ex.Expr

    varsEnvironment *ex.Vars
    argsNum         int
    mod             *Mod

    stdout, stderr io.Writer
    stdin          io.Reader
}
```

Перед началом выполнения кода на входном языке в объекте интерпретатора определяются исполняемая программа (control), потоки ввода, ошибок и вывода, а также загружаются стандартные функции, T и nil в текущую область видимости. Также в начало программы подставляется функция begin, так как программа является списком и должна возвращать результат вычисления

последнего выражения, а первым элементом списка должна быть функция (иначе вернется ошибка).

При старте интерпретатора запускается бесконечный цикл обработки выражений. На каждой итерации проверяется, является ли код, выполняемый в текущий момент времени парой или нет.

Если это пара – то выполняется голова списка в зависимости от типа выражения:

- символ – на стек с данными кладется выражение, соответствующее этому символу из таблицы символов;
- пара – в стеке вызовов сохраняется указатель на выполняемый в данный момент код, после чего управление передается первому элементу пары;
- в ином случае – на стек кладется само выражение;

а указатель на выполняемый код перемещается к хвосту.

Если же это пустой список (Nil) – то со стека снимаются все результаты выражений списка, после чего проверяется тип первого:

- функция – выполняется вычисление функции и результат кладется на стек, затем со стека вызовов снимается элемент и ему передается управление;
- замыкание или макрос – создается новая область видимости, а старая сохраняется в верхнем элементе стека вызовов; управление передается телу замыкания (макроса);
- в ином случае – на вершину стека кладется ошибка с сообщением о том, что результат вычисления первого выражения не является функцией, замыканием или макросом, со стека вызовов снимается элемент и ему передается управление.

5.3. Ошибки

Перед каждой итерацией проверяется, не лежит ли на вершине стека ошибка. Если это так, то со стека вызовов снимаются значения до тех пор, пока не встретится функция catch или стек не опустеет.

Если встречается catch, то происходит поиск подходящего тега, и если он найден – то управление передается его телу, в ином случае идет поиск следующего catch.

Если catch так и не встретился, то программа завершает свою работу, а в поток ошибок выводится трассировка стека и описание ошибки.

5.4. Модификаторы вычисления выражений

Некоторые функции (такие, как if или and) не требуют вычисления всех аргументов. После вычисления первого элемента списка проверяется, является ли его результат одной из таких функций. Если это так – то меняется схема вычисления выражений этого списка. Для этого в поле объекта интерпретатора mod ставится соответствующий идентификатор, а на каждой итерации интерпретатора он применяется. Если в поле mod лежит нулевой указатель – то никаких модификаторов к текущей итерации не применяется. Список модификаторов:

- ModOr – интерпретатор вычисляет выражения лишь до тех пор, пока не встретится выражение, результатом которого не является Nil. После этого вместо вычисления кладет на стек символ T как результат. Применяется в тех случаях, когда первый элемент является функцией or.
- ModAnd – интерпретатор вычисляет выражения лишь до тех пор, пока не встретится выражение, результатом которого является Nil.

После этого вместо вычисления кладет на стек пустой список как результат. Применяется в тех случаях, когда первый элемент является функцией and.

- ModIf – интерпретатор вычисляет условие, а затем проверяет его результат. Если это истина, то вычисляется третье выражение в списке, если ложь – то четвертое (при его наличии). Применяется в тех случаях, когда первый элемент является функцией if.
- ModExec – вычисляются только указанные аргументы. Остальные кладутся на стек без вычислений. Применяется в тех случаях, когда первый элемент является одной из следующих функций: quote (аргументы не вычисляются), define (вычисляется второй аргумент), defmacro (аргументы не вычисляются), lambda (аргументы не вычисляются), а также если первый аргумент является макросом (вычисляются только те аргументы, которые помечаются запятой в месте определения макроса).
- ModTry – вычисляется только первый аргумент, а для остальных на стек кладется пустой список как результат. Применяется в тех случаях, когда первый элемент является функцией catch.

6. Описание стандартных функций

6.1. Цитирование, универсальная функция, последовательное выполнение выражений

Функция quote возвращает выражение, не вычисляя его. Ожидается один аргумент. Следующие записи эквивалентны: ‘(quote {EXPR})’ и ‘’{EXPR}’, где {EXPR} – любое выражение. Примеры в таблице 10.

Таблица 10 – примеры использования ‘quote’.

Пример	Результат
`(quote (one two))`	`(one two)`
`(quote 23)`	`23`

Функция eval выполняет результат выражения. Ожидается один аргумент. Следующие записи эквивалентны: ‘(eval (quote {EXPR}))’ и ‘{EXPR}’. Примеры в таблице 11.

Таблица 11 – примеры использования ‘eval’.

Пример	Результат
`(eval '(+ 23 32))`	`55`
`(eval 23)`	`23`

Функция begin возвращает результат последнего выражения, либо Nil в случае нулевого количества аргументов. Примеры использования в таблице 12.

Таблица 12 – примеры использования ‘begin’.

Пример	Результат
`(begin)`	`nil`

`(begin 4 (+ 4 5) 'qwerqwe (/ 3 (- 1 2)))`	`-3`

6.2. Определение переменных

Функция define определяет переменную в текущей области видимости. Ожидается два аргумента: первый – символ, который является именем переменной, второй – выражение, результат которого будет сохранен и возвращен. Примеры в таблице 13.

Таблица 13 – примеры использования ‘define’.

Пример	Результат
`(define a 2)`	`2`
`(define a '|some string|)`	`some string`
`(define a (+ 20 3))` `(+ a 32)`	`55`

Функция set! переопределяет переменную в ближайшей области видимости, где она определена (деструктивное присваивание). Ожидается два аргумента: первый – символ, который является именем переменной, второй – выражение, результат которого будет сохранен и возвращен. Примеры в таблице 14.

Таблица 14 – примеры использования ‘set!.

Пример	Результат
`(define a 2)` `(set! a 3)`	`3`
`(define a 5)` `((lambda (b) (set! a (+ a b))`	`55`

50) a	

6.3. Определение анонимной функции

Функция lambda возвращает новое замыкание с текущей областью видимости в качестве родительской. Когда это замыкание будет вызвано, создастся новая область видимости и будет существовать до тех пор, пока оно вычисляется. Ожидается хотя бы два аргумента: первый – список с символами, которые обозначают аргументы или символ, который обозначает список аргументов, второй и последующие – тело замыкания. При вызове замыкание вычисляет все выражения из тела и возвращает результат последнего. Примеры в таблице 15.

Таблица 15 – примеры использования ‘lambda’.

Пример	Результат
(define a (lambda (b) (+ b b))) (a 3)	6
(define list (lambda args args)) (list 3 (+ 5 4) 0)	(3 9 0)
(define list (lambda args args)) (defmacro apply s (define f (car s)) (define args1 (car (cdr s))) (list 'eval (list 'cons f args1))) (apply + '(1 2 3)) (define 100+ (lambda args (if args (cons (+ 100 (car args)) (apply 100+ (cdr args))) nil))) (100+ 1 4 7)	(101 104 107)

(define a 5) ((lambda (b) (set! a (+ a b)) 50) a	55

6.4. Макроопределения

Функция defmacro определяет макрос в текущей области видимости (работа с макросами описана в разделе 7). Ожидается хотя бы три аргумента: первый - символ, с которым будет связан макрос, второй - список с символами, которые обозначают аргументы или символ, который обозначает список аргументов, третий и последующие – тело макроса. Выполняет результат выполнения последнего выражения в теле. Примеры в таблице 16.

Таблица 16 – примеры использования ‘defmacro’.

Пример	Результат
define list (lambda args args)) (define a 2) (defmacro set10! (b) (list 'set! b 10)) (set10! a) a	10
(defmacro apply s (define f (car s)) (define args1 (car (cdr s))) (list 'eval (list 'cons f args1))) (apply + '(3 4 5))	12

(define list (lambda args
args))
(defmacro map (f1 ,args1)
  (define helper (lambda (f
args)
    (if args
      (cons (list f (car args))
(helper f (cdr args)))
      nil)))
  (cons list (helper f1
	(-1 -2 -3)

args1))) (map - '(1 2 3))	

6.5. Условный оператор и логические операции

Функция if ожидается два или три аргумента: первый – условие, второй – выражение, которое будет вычислено и возвращено в том случае, если результат выполнения условия не является Nil, третий – в ином случае (если третий аргумент отсутствует, то возвращается Nil). Примеры в таблице 17.

Таблица 17 – примеры использования ‘if’.

Пример	Результат
(if (> 2 3) 'two 'three)	three
(if (> 2 1) 'two)	two
(if (> 2 3) 'two)	nil

Функция or (логическое ИЛИ) выполняет выражения до тех пор, пока не встретится выражение, результатом которого не является Nil и возвращает его результат. Если таковых нет, то возвращает Nil. Примеры в таблице 18.

Таблица 18 – примеры использования ‘or’.

Пример	Результат
(or 2 3)	2
(define a 3) (or nil (define a 5) (define a 10)) a	5
(or (cdr '(2)) nil)	nil
(or)	nil

Функция and (логическое И) выполняет выражения до тех пор, пока не встретится выражение, результатом является Nil. Если таковых нет, то возвращает результат последнего выражения, Nil - иначе. Если на вход подано нулевое количество аргументов, то возвращает символ ‘T’. Примеры в таблице 19.

Таблица 19 – примеры использования ‘and’.

Пример	Результат
`(and 2 3)`	`3`

`(define a 3)`
`(and nil (define a 5) (define a 10))`
`a`
	`3`

`(and (cdr '(3 3)) 'YY)`	`YY`
`(and)`	`T`

6.6. Работа с ошибками

Функция catch выполняет перехват ошибок. Если в теле выражения происходит ошибка, то выполнение передается блоку с подходящим тегом. Подробнее в разделе 2.6. Примеры использования в таблице 20.

Таблица 20 – примеры использования ‘catch’.

Пример	Результат
`(catch (/ 2 0) (/ 123123))`	`123123`

Функция throw создает ошибку с определенным тегом. Подробнее в разделе 2.6. Примеры использования в таблице 21.

Таблица 21 – примеры использования ‘throw’.

Пример	Результат
`(catch (throw 'error) (err 123123))`	`123123`

6.7. Функции ввода и вывода

Функция write выводит строковое представление результата выполнения выражения в поток вывода и возвращает его (результат). Ожидается один аргумент. Примеры использования в таблице 22.

Таблица 22 – примеры использования ‘write’.

Пример	Результат	Вывод
`(write '(2))`	`(2)`	`(2)`
`(write (if T 'ss 3))`	`ss`	`ss`

Функция read читает строковое представление выражений из потока ввода и возвращает его список этих выражений. Ожидается нулевое число аргументов. Примеры использования в таблице 23.

Таблица 23 – примеры использования ‘read’.

Пример	Результат	Ввод
`(read)`	`((2) 3)`	`(2) 3`

Функция load читает строковое представление выражений из файла и возвращает его список этих выражений. Ожидается один аргумент – путь к файлу. Примеры использования в таблице 24.

Таблица 24 – примеры использования ‘load’.

Пример	Результат	Ввод

`(load)`	`(45 (+ 6 7))`	`45` `(+ 6 7)`

6.8. Операции с точечными парами

Функция cons возвращает пару. Ожидается два аргумента: «голова» и «хвост». «Хвост» должен быть парой или Nil. Примеры использования в таблице 25.

Таблица 25 – примеры использования ‘cons’.

Пример	Результат
`(cons 2 nil)`	`(2)`
`(cons (+ 5 6) '(12 13 14))`	`(11 12 13 14)`

Функция car возвращает «голову» пары. Ожидается один аргумент, который должен быть парой. Примеры использования в таблице 26.

Таблица 26 – примеры использования ‘car’.

Пример	Результат
`(car '(2))`	`2`
`(car '((4 5) 6 7))`	`(4 5)`

Функция cdr возвращает «хвост» пары. Ожидается один аргумент, который должен быть парой. Примеры использования в таблице 27.

Таблица 27 – примеры использования ‘cdr’.

Пример	Результат
`(cdr '(2))`	`nil`
`(cdr '((4 5) 6 7))`	`(6 7)`

6.9. Преобразования типов

Функция symbol->number преобразует символ в число. Ожидает один аргумент, который должен быть символом и являться строковым представлением какого-либо числа. Примеры использования в таблице 28.

Таблица 28 – примеры использования ‘symbol->number’.

Пример	Результат
`(symbol->number '|  23 |)`	`23`
`(symbol->number '|6/3|)`	`2`
`(symbol->number '|1234.56e-2|)`	`12.3456`
`(symbol->number '|  -23.4|)`	`-23.4`

Функция number->symbol преобразует число в символ, который является строковым представлением числа. Ожидается один аргумент, который должен быть числом. Примеры использования в таблице 29.

Таблица 29 – примеры использования ‘number->symbol’.

Пример	Результат
`(define |234| (lambda (a) (+ a 1)))` `((eval (number->symbol 234)) 5)`	`6`

6.10. Предикаты типов

Функция symbol? возвращает символ ‘T’, если аргумент является символом и Nil – в ином случае. Ожидается один аргумент. Примеры использования в таблице 30.

Таблица 30 – примеры использования ‘symbol?’.

Пример	Результат
`(symbol? '|2|)`	`T`
`(symbol? 2)`	`nil`

Функция number? возвращает символ ‘T’, если аргумент является числом и Nil – в ином случае. Ожидается один аргумент. Примеры использования в таблице 31.

Таблица 31 – примеры использования ‘number?’.

Пример	Результат
`(number? 2)`	`T`
`(number? '|2|)`	`nil`

Функция pair? возвращает символ ‘T’, если аргумент является парой и Nil – в ином случае. Ожидается один аргумент. Примеры использования в таблице 32.

Таблица 32 – примеры использования ‘pair?’.

Пример	Результат
`(pair? '(2 3 4 5))`	`T`
`(pair? nil)`	`Nil`

Функция not возвращает символ ‘T’, если аргумент является парой и Nil – в ином случае. Ожидается один аргумент. Примеры использования в таблице 33.

Таблица 33 – примеры использования ‘not’.

Пример	Результат
`(not nil)`	`T`

`(not 1234)`	`nil`

6.11. Функции сравнения

Функция = возвращает символ ‘T’, если аргументы эквивалентны и Nil – в ином случае. Ожидается хотя бы два аргумента. Примеры использования в таблице 34.

Таблица 34 – примеры использования ‘=’.

Пример	Результат
`(= 's2 (begin 's2) (if nil 2 's2) (+ 's '|2|))`	`T`
`(= '(2 3) (cons 2 '(3)))`	`T`
`(= 2 '|2|)`	`nil`
`(= 2 2 2 2 2 3)`	`nil`

Функция > возвращает символ ‘T’, если первый аргумент больше второго и Nil – в ином случае. Ожидается два числа. Примеры использования в таблице 35.

Таблица 35 – примеры использования ‘>’.

Пример	Результат
`(> 3 2)`	`T`
`(> 3 3)`	`nil`

Функция < возвращает символ ‘T’, если первый аргумент меньше второго и Nil – в ином случае. Ожидается два числа. Примеры использования в таблице 36.

Таблица 36 – примеры использования ‘<’.

Пример	Результат
`(< -7 2)`	`T`
`(< 4 2)`	`nil`

6.12. Функции для работы с числами и символами

Функция len возвращает длину символа. Ожидается один символ. Примеры использования в таблице 37.

Таблица 37 – примеры использования ‘len’.

Пример	Результат
`(len '12345)`	`5`
`(len '漢字!)`	`3`

Функция + возвращает сумму чисел или конкатенацию символов. Ожидается любое количество чисел или хотя бы один символ. Примеры использования в таблице 38.

Таблица 38 – примеры использования ‘+’.

Пример	Результат
`(+ 2 3 4)`	`9`
`(+ 'hello, '| world!|)`	`hello, world!`
`(+)`	`0`

Функция - Возвращает разность чисел или символ, который является подстрокой другого символа. Ожидается любое количество чисел или один символ и два числа. Примеры использования в таблице 39.

Таблица 39 – примеры использования ‘-’.

Пример	Результат
`(- )`	`-2`
`(- 2 3 4)`	`-5`
`(- 'thisstringishuge 4 10)`	`string`
`(-)`	`0`

Функция * возвращает произведение чисел. Ожидается любое количество чисел. Примеры использования в таблице 40.

Таблица 40 – примеры использования ‘*’.

Пример	Результат
`(* 2 3)`	`6`
`(*)`	`1`

Функция / производит деление чисел. Ожидается хотя бы одно число. Примеры использования в таблице 41.

Таблица 41 – примеры использования ‘/’.

Пример	Результат
`(/ 6 2 4)`	`0.75`
`(/ 7)`	`7`

7. Интерфейс интерпретатора

7.1. REPL

REPL (от английского “read-eval-print-loop”) – интерактивная терминальная среда разработки, которая принимает от пользователя выражение, вычисляет его и выводит результат в стандартный поток вывода [10]. Программа, написанная в этой среде, будет выполняться по частям, и пользователю будут видны промежуточные результаты работы программы, что позволяет быстро протестировать отдельные функции, а также ускоряет знакомство с языком.

Реализовывать REPL оболочку было принято на самом языке LispXS. Реализация предоставлена в листинге 4. Во избежание переопределения пользователем функций, которые используются во время работы оболочки, создается новая область видимости, в которую они и переносятся. Функция repl1 печатает приглашение для ввода выражения, затем исполняет все, что вводит пользователь и выводит на экран. Если при выполнении введенных пользователем выражений происходит ошибка, то она перехватывается, также выводится на экран, а программа интерпретатора работает дальше, не завершаясь.

Листинг 9.

```
(define repl nil)
((lambda ()
  (define define define) (define lambda lambda) (define defmacro
defmacro)
  (define if if) (define cons cons) (define car car) (define cdr
cdr) (define nil nil)
  (define write write) (define begin begin) (define eval eval)
(define read read)
  (define catch catch)
  (define list (lambda args args))
  (defmacro map (f1 ,args1)
    (define helper (lambda (f args)
      (if args
        (cons (list f (list quote (car args))) (helper f (cdr
args))))))
```

```
    (cons list (helper f1 args1)))
  (define writeln (lambda (sym) (write sym) (write '|\n|) sym))
  (defmacro repl1 ()
    (list begin
      (list write ''|> |)
      (list map writeln
        (list catch
          (list map eval (list read))
          '(default (list (+ '|ERROR, | error_description)))))
      (list repl1)))
  (set! repl repl1)))
(repl)
```

7.2. Golang вызовы

Для работы с интерпретатором посредством вызовов функций из языка Golang был разработан следующий интерфейс:

- Execute – выполнение программы на языке LispXS. Поток вывода и ошибок возвращает в виде строк.
- ExecuteStdout – также выполнение программы, однако в качестве потоков вывода и ошибок используются os.Stdin и os.Stderr соответственно.
- ExecuteTo – потоки, которые используются в качестве потоков вывода и ошибок подаются в аргументах функции.
- LoadLibrary – загружает и возвращает библиотеку на языке LispXS для последующего вызова определенных в ней функций при помощи метода Call.
- Call – вызов функций из библиотеки с необходимыми аргументами.
- obj_free – очищает память из-под объекта.

7.3. FFI вызовы

FFI (от английского «foreign function interface») – интерфейс вызова внешних функций. Данный механизм позволяет вызывать программы, написанные на одном языке программами, написанными на другом [11].

Интерфейс для вызова из других языков при помощи технологии FFI:

- execute – то же, что и одноименная функция из раздела 7.2.
- execute_stdout – то же, что и одноименная функция из раздела 7.2.
- library_load – то же, что и одноименная функция из раздела 7.2.
- library_call – то же, что и функция Call из раздела 7.2.
- call_new – создание нового объекта ‘call’ для вызова функции из LispXS-библиотеки.
- call_add_number – добавление числа к вызову.
- call_add_symbol – добавление символа к вызову.
- call_add_list – добавление списка к вызову (для списка используется объект ‘call’).
- expr_is_pair – предикат для проверки выражения на то, является ли оно списком.
- expr_length – длина списка.
- expr_index – индексация списка.
- expr_atom – возвращает информацию об атоме (тип и данные).

ЗАКЛЮЧЕНИЕ

Благодаря своим возможностям, связанными с метапрограммированием, Лисп и по сей день используется разработчикам программного обеспечения, особенно в областях, связанных с развитием искусственного интеллекта.

В результате выполнения данной работы были выполнены следующие задачи:

- рассмотрены основные особенности языков семейства Лисп;
- разработан минималистичный Лисп, который является полностью расширяемым, а также реализован интерпретатор для выполнения написанного на нем кода;
- добавлена возможность встраивания в программы, написанные на других языках.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. P. Graham. The Roots of Lisp, 2002.
2. P. Graham. ANSI Common Lisp – Prentice Hall, 1996.
3. M.V. Zelkowitz. Advanced in Computers – Elsevier Inc., 2009.
4. J. McCarthy. History of Lisp, 1979.
5. T.D. Brown. C for Fortran Programmers – Silicon Press, 1990.
6. H.W. Loidi, R. Pena. Trends in Functional Programming – Springer Heidelberg New York Dordrecht London, 1998.
7. John McCarthy. LISP 1.5 Programmer's Manual, 1965.
8. А. Ю. Молчанов. Системное программное обеспечение – Питер, 2010.
9. М. Фаулер. Предметно-ориентированные языки программирования – Вильямс, 2017.
10. P. Seibel. Practical Common Lisp – Apress, 2005.
11. B. Cureton. Software Engineering on Sun Workstations – Springer-Verlag, 1993.

ПРИЛОЖЕНИЕ А

Руководство пользователя

1. Установка и использование как standalone приложения

Для сборки проекта требуется установленный на устройстве компилятор языка Golang версии 1.13 и выше. Команды для загрузки и сборки предоставлены на листинге 1.

Листинг 1 – загрузка репозитория и сборка проекта.

```
$ git clone https://github.com/batrSens/LispXS.git
$ cd ./LispXS
$ go build
```

Для тестирования приложения необходимо выполнить команды из листинга 2.

Листинг 2 – запуск тестов ({lispx_directory} – директория с загруженным репозиторием).

```
$ cd {lispxs_directory}
$ go test ./…
```

Чтобы запустить приложение, нужно воспользоваться командами, представленными в листинге 3.

Листинг 3 – запуск интерпретатора.

```
$ cd {lispxs_directory}
$ ./LispXS [-n -e -r]
```

Флаги запуска:

- -n – ожидание двойного переноса строки (“\n\n”) в конце входящей программы;

- -e – ожидание EOF символа в конце входящей программы;
- -r – REPL-режим. Интерпретатор работает в этом режиме по умолчанию. В этом режиме приложение ожидает на вход выражение, после чего исполняет его, выводит результат в стандартный поток вывода и затем снова ожидает выражение.

2. Использование как Golang-библиотеки

Для установки зависимости необходимо набрать команду из листинга 4 и добавить наименование библиотеки в import-блок (как в листингах 5 и 7).

Листинг 4 – установка зависимости.

```
$ go get go.mongodb.org/mongo-driver/mongo
```

Описание методов:

- Execute(program string) (*Output, error) – принимает на вход программу на языке LispXS. Исполняет и возвращает результат работы программы, поток вывода и ошибок (в виде строк) в структуре Output.
- ExecuteStdout(program string) (*ex.Expr, error) – принимает на вход программу на языке LispXS. Исполняет и возвращает результат работы программы. В качестве потоков вывода и ошибок используются os.Stdin и os.Stderr соответственно.
- ExecuteTo(program string, ioout, ioerr io.Writer, ioin io.Reader) (*ex.Expr, error) – принимает на вход программу на языке LispXS, а также потки, которые будут использоваться в качестве потоков ввода, вывода и ошибок. Исполняет и возвращает результат работы программы.

- LoadLibrary(path string) (*Library, error) – загружает и возвращает библиотеку на языке LispXS для последующего вызова определенных в ней функций при помощи метода Call.

- (lib *Library) Call(symbol string, args …interface{}) (*ex.Expr, error) – вызов функций из библиотеки. Первом аргументом идет наименование, последующие аргументы являются аргументами функции. Каждый аргумент может быть одним из следующих типов: int, float64, string и []interface{}. Срез также должен содержать аргументы перечисленных типов.

Примеры использования представлены на листингах 5-7.

Листинг 5 – выполнение LispXS-кода из Golang-программы.

```
package main

import (
    “fmt”

    lispxs “github.com/batrSens/LispXS/interpreter”
)

func main() {
    res, err := lispxs.Execute(“(+ ‘Hello, ‘| World!|)”)
    if err != nil {
        panic(err)
    }

    fmt.Println(res.Output.String) // Выведет “Hello, World!”
}
```

Листинг 6 – LispXS-библиотека.

```
(define hello (lambda (name)
  (+ ‘|Hello, | name ‘!)))

(define ++ (lambda (a) (+ a 1)))
```

Листинг 7 – вызов функций из LispXS-библиотеки.

```
package main

import (
     «fmt»

     lispxs «github.com/batrSens/LispXS/interpreter»
)

func main() {
    lib, err := lispxs.LoadLibrary(«path_to_file») // Загрузка
библиотеки
    if err != nil {
        panic(err)
    }

    res, err := lib.Call(«hello», «World») // Вызов функции
'hello'
    if err != nil {
        panic(err)
    }

    fmt.Println(res.ToString()) // Выведет «Hello, World!»

    res, err = lib.Call(«++», 2019) // Вызов функции '++'
    if err != nil {
        panic(err)
    }

    fmt.Println(res.ToString()) // Выведет 2020
}
```

3. Использование как FFI-библиотеки

Для сборки необходимо выполнить команды из листинга 8. Создадутся файл с библиотекой ‘lispxs.so’ и хедер-файл ‘lispxs.h’.

Листинг 8 – сборка FFI-библиотеки

```
$ cd {lispxs_directory}

$ go build -o lispxs.so -buildmode=c-shared ./ffi/main/main.go
```

FF интерфейс (описание первых четырех функций находится в разделе 2):

- execute(prog *C.char) (expr unsafe.Pointer, stdout, stderr, error *C.char)
- execute_stdout(prog *C.char) (expr unsafe.Pointer, error *C.char)
- library_load(path *C.char) (library unsafe.Pointer, str *C.char)
- library_call(lib, call unsafe.Pointer) (expr unsafe.Pointer, error *C.char)
- call_new(fn *C.char) unsafe.Pointer – создание нового объекта ‘call’ для вызова функции из LispXS-библиотеки.
- call_add_number(c unsafe.Pointer, number float64) unsafe.Pointer – добавление числа к вызову.
- call_add_symbol(c unsafe.Pointer, symbol *C.char) unsafe.Pointer – добавление символа к вызову.
- call_add_list(c, list unsafe.Pointer) unsafe.Pointer – добавление списка к вызову (для списка используется объект ‘call’).
- expr_is_pair(expr unsafe.Pointer) bool – предикат для проверки выражения на то, является ли оно списком.
- expr_length(expr unsafe.Pointer) int – длина списка.
- expr_index(expr unsafe.Pointer, i int) unsafe.Pointer – индексация списка.
- expr_atom(expr unsafe.Pointer) (typ int, number float64, str *C.char) – возвращает информацию об атоме (тип и данные).

Примеры использования FFI библиотеки из C-кода предоставлены в листингах 9-11.

Листинг 9 – компиляция и запуск C-программы.

```
$ gcc -o lispxsc ./main.c ./lispxs.so

$ ./lispxsc
```

Листинг 10 - выполнение LispXS-кода из C-программы.

```
#include <stdio.h>
#include "lispxs.h"

int main() {
    struct execute_stdout_return res = execute_stdout("(define
list (lambda args args)) (list 'Hello, 'C-lang! (+ 2000 20))");

    GoInt len = expr_length(res.r0);

    // Выведет "Hello, C-lang! 2020 "
    for (int i = 0; i < len; i++) {
        struct expr_atom_return atom =
expr_atom(expr_index(res.r0, i));

        switch (atom.r0) {
        case 1:
            printf("%.f ", atom.r1);
            break;
        case 2:
            printf("%s ", atom.r2);
            break;
        default:
            printf("undefined ");
        }
    }

    printf("\n");

    return 0;
}
```

Листинг 11 – вызов функций из LispXS-библиотеки.

```
#include <stdio.h>
#include "lispxs.h"

int main() {
    struct library_load_return lib =
library_load("path_to_file"); // Загрузка библиотеки
```

```
    void *call = call_new("hello");
    call = call_add_symbol(call, "C-lang");

     struct library_call_return res = library_call(lib.r0, call);
// Вызов функции 'hello' с аргументом 'C-lang
     printf("%s\n", expr_atom(res.r0).r2); // Выведет "Hello, C-
lang!"

     call = call_new("++");
     call = call_add_number(call, 2019);

     res = library_call(lib.r0, call); // Вызов функции '++' с
аргументом 2019
     printf("%.f\n", expr_atom(res.r0).r1); // Выведет 2020

    return 0;
}
```

4. Система типов

Язык имеет динамическую неявную типизацию. Поддерживаются следующие типы:

- Number – числа;
- Symbol – лисповые символы;
- Pair – непустой список;
- Nil – пустой список, а также ложь для логических выражений.

5. Код и выражения

Все, что есть в коде, включая саму программу – это выражения. Результатом выражений также является выражения. Общая схема вычисления выражений (рисунок 1): если выражение является символом – то результат берется из таблицы символов, если выражение является парой – то сначала вычисляются все элементы списка, после чего результату первого элемента (если это функция (под функциями подразумеваются все стандартные процедуры языка), замыкание или макрос, в ином случае возвращается ошибка) подаются на вход все остальные и результат возвращается как вычисление этой пары. В

иных случаях выражение возвращает само себя (например, число 7 вернет число 7, а пустой список вернет пустой список).

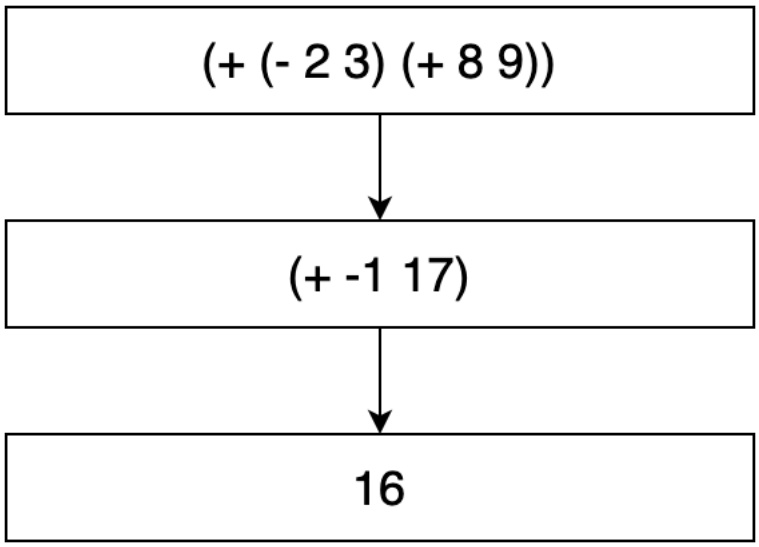

Рисунок 1 – пример вычисления выражений, являющихся парами.

6. Области видимости

На старте программы существует только одна область видимости переменных, в которой содержатся только стандартные функции языка, символ T, содержащий себя же и символ nil с пустым списком. Новые области видимости создаются при вызове функций и макросов (на этапе генерации кода).

Функции define и set! выполняют похожие действия: define определяет переменную в текущей области видимости, set! переопределяет переменную в ближайшей.

7. Макросы

Работа с макросами происходит в три этапа:

1. При помощи функции defmacro происходит определение макроса.
2. В момент вызова макроса происходит генерация кода.

3. Выполнение сгенерированного кода.

8. Prelude

Чтобы расширить язык, не трогая при этом исполняемый код, можно воспользоваться ‘prelude’ файлом – перед выполнением программы интерпретатор найдет файл с названием ‘prelude’ в текущем рабочем каталоге и выполнит его содержимое.

9. Перехват и создание ошибок

Для перехвата ошибок необходимо воспользоваться функцией catch (структура в листинге 12), подав первым аргументом код, который может вернуть ошибку. Тогда в случае возникновения ошибки управление передастся обработчику с подходящим тегом (тег является префиксом к тегу ошибки или это тег ‘default’).

Листинг 12 – структура ‘catch’.

```
(catch body_that_can_throw_an_error (tag1 body) (tag2 body) ...)
```

Для генерации собственных ошибок можно воспользоваться функцией catch.

10. Набор стандартных функций языка

Язык содержит следующий набор стандартных функций: quote, eval - цитирование и выполнение выражений; define, set! – определение переменных; lambda, defmacro – замыкания и макросы; if, or, and – условный оператор и логические выражения; catch, throw – работа с ошибками; write, read, load – операции ввода-вывода; begin – последовательное вычисление выражений; cons, car, cdr – работа со списками; symbol->number, number->symbol, symbol?, number?, pair?, not – конвертация и предикаты типов; =, >, < - операции

сравнения; len, +, -, *, / - арифметические операции и работа со строками (символами).