1 Интерпретатор SBCL

Запуск интрепретатора осуществляется командой `sbcl`.

```
C:\Users\contest>sbcl
This is SBCL 1.2.1, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.

WARNING: the Windows port is fragile, particularly for multithreaded
code.  Unfortunately, the development team currently lacks the time
and resources this platform demands.
*
```

Приглашением интерпретатора для ввода команды является `*` .

Команда считается завершенной если Enter нажат после введения команды со сбалансированными круглыми скобками. После ввода команды система её анализирует, компилирует, выполняет и выводит результат выполнения на терминал.

Образец диалога:

```
* (+ 3 2)

5
```

Здесь пользователь вводит фразу `(+ 3 2)`, что означает «Сложить 3 и 2». Система вычисляет выражение и выдает сообщение о результате: `5`.

В качестве команды интерпретатору может быть передано имя переменной. В таком случае результатом выполнения такой команды будет значение переменной. Также, командой может быть любая системная константа. Тогда результатом выполнения будет значение этой константы. Если же командой интерпретатора является вызов фукнкции (макроса, метода), то результатом выполнения команды будет значение данного вызова.

Если в качестве команды интерпретатору передается имя неинициализированной переменной или вызов необъявленной функции, то интерпретатор выдает сообщение об ошибке вместо результата.

Все «команды» языка LISP можно воспринимать функциями — любая из команд возвращает результат и может быть использована в выражении.

В процессе диалога могут возникать различные виды ошибок. В основном они распадаются на три категории: синтаксические ошибки, ошибки в согласовании типов и ошибки времени исполнения. При возникновении ошибки система инициирует диалог обработки ошибки — вам предлагается выбрать предложенный вариант ее обработки. Обычно, когда при возникновении ошибки необходимо просто прервать работу программы, нужно нажать на Enter, ввести abort (или число, соответствующее этому действию в сообщении интерпретатора) и снова нажать Enter.

Пример того, что происходит, когда введена синтаксически неправильная фраза:

```
* (let (x 3) x)
; in: LET (X 3)
;     (LET (X 3)
;       X)
;
; caught ERROR:
;   3 is not a symbol, and cannot be used as a variable.
;
; compilation unit finished
;   caught 1 ERROR condition

debugger invoked on a SB-INT:COMPILED-PROGRAM-ERROR in thread
#<THREAD "main thread" RUNNING {24086A01}>:
  Execution of a form compiled with errors.
Form:
  (LET (X 3)
  X)
```

```
Compile-time error:
  3 is not a symbol, and cannot be used as a variable.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.
```

Ошибки времени исполнения (как, например, деление на 0) являются одним из видов *исключительных событий*. Вот пример того, что можно получить при возникновении ошибки времени исполнения:

```
* (/ 3 0)

debugger invoked on a DIVISION-BY-ZERO in thread
#<THREAD "main thread" RUNNING {24086A01}>:
  arithmetic error DIVISION-BY-ZERO signalled
Operation was SB-KERNEL::DIVISION, operands (3 0).

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.
```

Ошибки в согласовании типов возникают при некорректном использовании значений, например, при попытке прибавить 34 к NIL:

```
* (+ 34 NIL)

debugger invoked on a SIMPLE-TYPE-ERROR in thread
#<THREAD "main thread" RUNNING {24086A01}>:
  Argument Y is not a NUMBER: NIL

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.
```

Одной из ошибок, не распознаваемых системой, является бесконечный цикл. Если вы подозреваете, что ваша программа зациклилась, вы можете прекратить ее выполнение, нажав Ctrl-C. Система выдаст варианты обработки останова, после которого выполнение программы можно будет прервать или продолжить. В следующем примере определяется функция, вызывающая сама себя бесконечное число раз. При запуске этой функции от любого аргумента происходит зацикливание. Ниже показана реакция системы на нажатие Ctrl-C во время обработки вызова этой функции.

```
* (defun f (n) (f n))

F
* (f 4)

debugger invoked on a SB-SYS:INTERACTIVE-INTERRUPT in thread
#<THREAD "main thread" RUNNING {24086A01}>:
  Interactive interrupt at #x241E776C.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE] Return from SB-WIN32::SIGINT.
  1: [ABORT   ] Exit debugger, returning to top level.
```

Исполнение файла с программой на языке LISP осуществляется с помощью команды `load`

```
* (load "C:\\temp\\streams.lsp")
```

Выход из интерпретатора происходит после команды `(exit)`

```
* (exit)
```

Для включения в тело программы на языке LISP однострочных комментариев используется символ `;`. Текст, расположенный между ним и концом строки, считается комментарием.

2 Основные конструкции языка

2.1 S-выражения

Основу языка LISP составляют символьные выражения, которые называются S-выражениями и образуют область определения для функциональных программ.

S-выражение — это либо атом, либо списочная структура. Примеры S-выражений:

```
250
ABA
"John Smith"
(A 1 2 3 4)
(B (DBA FG) AB B 15)
(B (DBA FG) . 15)
```

Атом — объект, представляющий единое неделимое целое. Из атомов строятся другие лисповские структуры. Так из перечисленных выше объектов атомами являются первые три, остальные же являются списочными структурами.

Отличают символьные и числовые атомы. Символьные атомы — элементы данных типа SYMBOL. Обычно их можно представить как последовательность цифр и букв латинского алфавита не являющуюся числом. Например, `Ab1`, `1f1`, `a`. Числовым атомом называют элемент любого числового типа. Например, `25`, `2/7`, `2.6E16`.

Но числовыми и символьными атомами набор атомов не ограничивается. Атомом является, к примеру, любая стока (элемент типа STRING), любой символ (элемент типа CHARACTER) и т. п.

Символьный атом `NIL` играет важную роль в языке LISP. Прежде всего, этот атом обозначает пустой список — список, в котором нет ни одного элемента. Еще, в контексте логических выражений, этот атом обозначает обозначает ложь. Истиной в логических выражениях выступает любое значение отличное от `NIL`, но кроме этого, истину обозначает специальный символьный атом `T`.

Программы и данные имеют одинаковую списочную структуру. Конкретно это выражается в том, что вызов любой функции представляет собой список — последовательность разделенных пробелами элементов, заключенную в круглые скобки. Первый элемент списка-вызова функции — описание функциональности, а остальные — аргументы вызова функциональности. Простейшее описание функциональности — указание имени существующей функции. Так например вызовами функций могут являться списки

```
(+ 1 2 3 4)
(list AB CB 15)
(cons (DBA FG) NIL)
```

С любым симольным атомом, кроме `T` и `NIL`, может быть связано некоторе значение, т. е. символьный атом может выступать в роли переменной. С символьными атомом `NIL` связано значение `NIL`, а символьным атомом `T` — значение `T`.

LISP-система воспринимает по умолчанию любой непустой список как вызов функции, а любой символьный атом как имя переменной. Чтобы указать системе на то, что объект должен восприниматься как элемент данных, необходимо применить специальную функцию `quote`. Функция `quote` возвращает свой аргумент без вычислений. Так например

```
* (quote (+ 1 2 3 4 5))

(+ 1 2 3 4 5)
* (quote (A B C D))

(A B C D)
* (quote AB)

AB
```

Функция `quote` имеет более короткую запись: вместо `(quote expr)` можно писать `'expr`.

Таким образом, когда пишут `(A B C D)`, то имеют ввиду, что нужно вызвать функцию `A` с параметрами `B`, `C` и `D` (где `B`, `C` и `D` — имена переменных), а когда пишут `'(A B C D)`, то обращаются к значению — списку из четырех элементов `A`, `B`, `C` и `D` (где `A`, `B`, `C` и `D` — не имена переменных, а просто символьные атомы). Когда пишут `A`, то подразумевают обращение к значению переменной с именем `A`, а когда пишут `'A`, то обращаются к значению — символьному атому `A`.

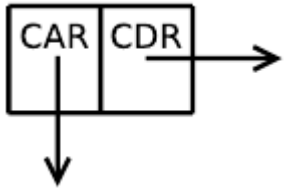
2.2 Списочные структуры

В дальнейшем будем употреблять термины «внешнее представление» и «внутреннее представление» объекта. Под внешним представлением объекта будем понимать форму, в которой объект представляется в программе на LISP или в которой система выдает объект как результат выполнения функции. Внутреннее представление объекта — описание совокупности ячеек памяти представляющих объект.

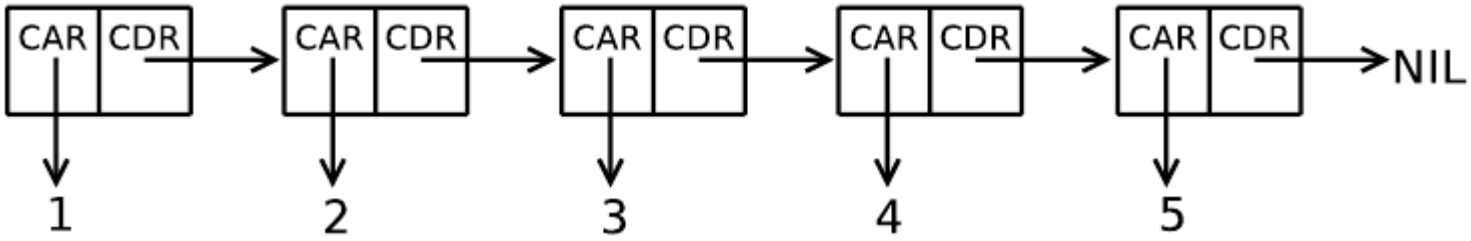
Два основных типа данных для списочных структур — тип LIST для списков и тип CONS для точечных пар.

Во внешнем представлении список является последовательностью элементов данных. Во внутреннем представлении список является последовательностью так называемых CONSов — ячеек памяти, содержащих по паре указателей.

CONS — элемент типа данных CONS, ячейка памяти состоящая из двух частей, которые называются называются CAR и CDR соответственно. Каждая из частей представляет собой указатель и может ссылаться на любую структуру данных LISPa, в том числе и на другой CONS.



Каждый представитель типа LIST — список. В списках CONSы выстраиваются в ряд, ссылаясь друг на друга с помощью ссылки CDR, а ссылка CDR последнего в списке CONSa указывает на `NIL` — специальный элемент типа SYMBOL. Каждый CONS отвечает за один элемент списка, а часть CAR CONSa указывает на содержимое соответствующего элемента.



`NIL` — выступает в роли пустого списка, т. е. списка, в котором нет ни одного элемента, а значит и ни одного CONSa.

Внешнее представление CONSa, называемое точечной парой, выглядит следующим образом

(CAR . CDR)

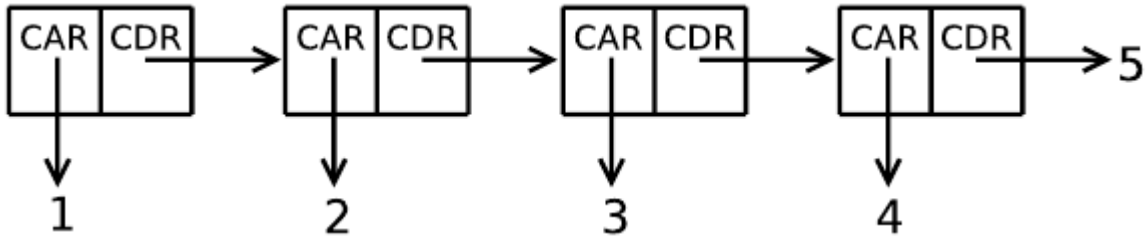
где точка окружена обязательными пробелами, а CAR и CDR представляют значения, на которые ссылаются соответствующие части.

Таким образом, внешнее представление списка представленного на рисунке может быть следующим

```
(1 . (2 . (3 . (4 . (5 . NIL)))))
```

Не всякая последовательнось CONSов является списком, а только та, которая заканчивается элементом `NIL`. Так например следующая структура — не список.

```
(1 . (2 . (3 . (4 . 5))))
```



Представленные структуры могут иметь другое, более простое, внешнее представление: внутри круглых скобок через пробелы перечисляются значения, на которые ссылаются части CAR CONSов, после которых через точку представляется значение, на которое ссылается часть CDR последнего CONSa. Так, последняя структура может быть представлена в виде

```
(1 2 3 4 . 5)
```

а упомянутый ранее список может быть представлен в виде

```
(1 2 3 4 5 . NIL)
```

Более того, если последний CONS ссылается на `NIL`, т. е. структура является списком, точку и `NIL` в конце внешнего представления обычно не пишут. Т. е. последний список можно представить как

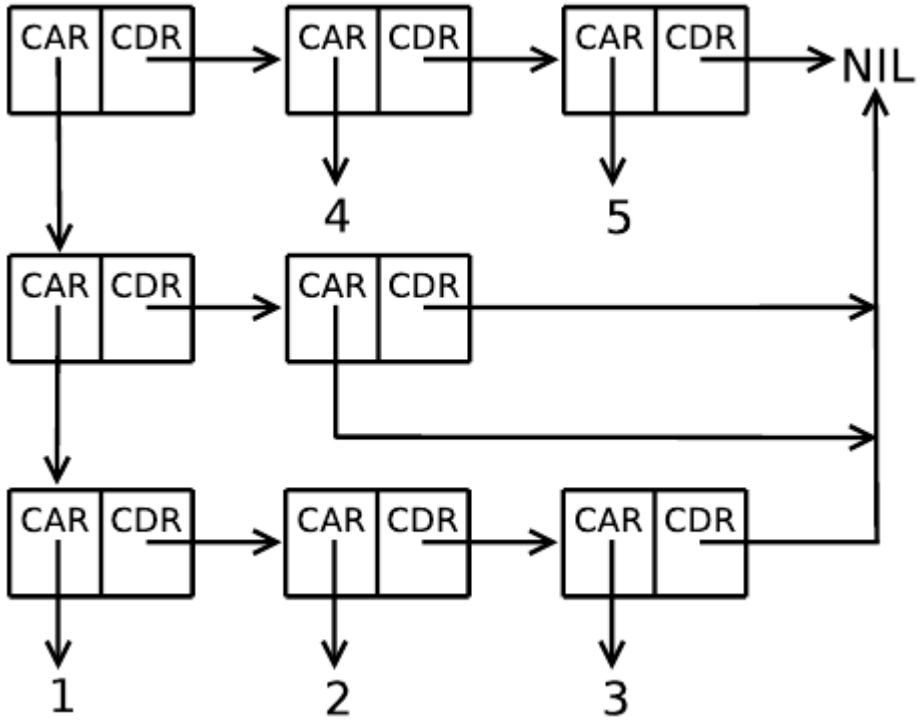
```
(1 2 3 4 5)
```

Внешними представлениями пустого списка являются `NIL` и `()`.

Элементами списка могут выступать другие списки. Например

```
(((1 2 3) ()) 4 5)
```

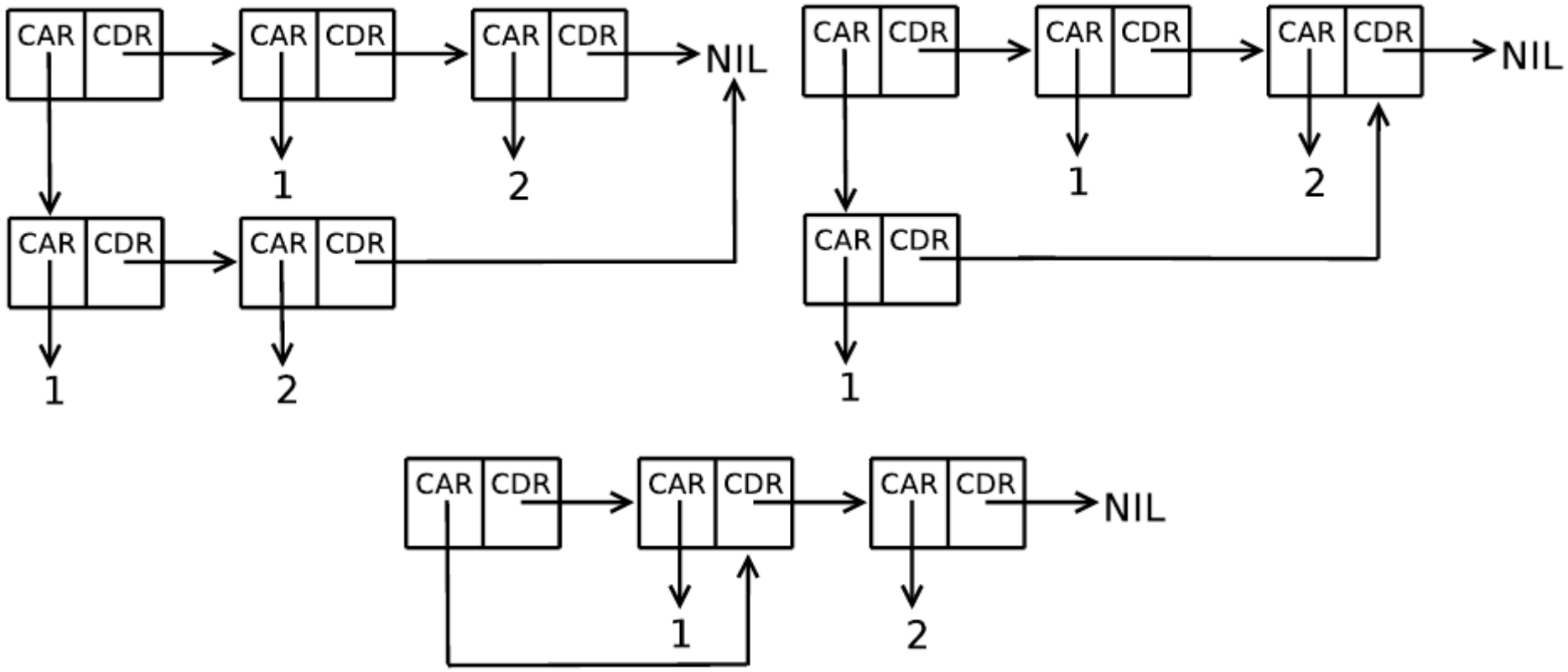
Здесь в списке три элемента. Первый из них сам является списком, в котором 2 элемента. Если рассмотреть внутреннюю структуру этого списка, то она может выглядеть следующим образом.



Для одного и того же внешнего представления некоторых списком возможны несколько внутренних представлений. Рассмотрим например список

```
((1 2) 1 2)
```

Для этого списка возможны следующие внутренние представления (и любое из них можно получить средствами LISP):



Список может быть задан явно во внешнем представлении или сконструирован с помощью специальных функций.

2.3 Базовые функции LISP

В языке LISP все функции (конструкции, методы, макросы, операции) оформляются в префиксной форме и представляют собой список, первый элемент которого задает действие, а остальные элементы — параметры этого действия.

2.3.1 Основные предикаты

Функция `atom` возвращает `T` если аргумент является атомом и `NIL` в противном случае.

```
* (atom '(1 2 3 4))

NIL
* (atom 25)

T
* (atom 'A)

T
```

Функция `listp` возвращает `T` если аргумент является списком. В противном случае функция выдает `NIL`.

```
* (listp '(1 2 3 4))

T
* (listp 25)

NIL
```

Функция `null` возвращает `T` если аргумент является пустым списком и `NIL` в противном случае.

```
* (null '(1 2 3 4))

NIL
* (null NIL)

T
* (null ())

T
* (null 25)

NIL
```

2.3.2 Работа со списками

Функция `list` может иметь произвольное число аргументов и в качестве результата выдает список составленный из них.

```
* (list 1 2 3 4 5)

(1 2 3 4 5)
* (list)

NIL
```

Функция `cons` принимает два аргумента, создает и выдает в качестве результата CONS, в котором часть CAR указывает на первый аргумент, а часть CDR — на второй аргумент. Например

```
* (cons 1 NIL)

(1)
* (cons 1 (cons 2 NIL))

(1 2)
* (cons 1 (cons 2 3))

(1 2 . 3)
```

Функция `cons` может быть применена для конструирования произвольных точечных пар.

Функции `car` и `cdr` являются обратными функциями для функции `cons`. Обе эти функции получают CONS в качестве аргумента. Функция `car` выдает значение, на которое ссылается часть CAR данного CONSa, а функция `cdr` выдает значение, на которое ссылается часть CDR.

```
* (car (cons 1 2))

1
* (cdr (cons 1 2))

2
```

Когда в качестве аргументов эти функции получают список, про функцию `car` говорят, что она извлекает голову списка, а про функцию `cdr` — извлекает хвост списка. Т. о. голова списка — его первый элемент, а хвост — исходный список без первого элемента.

Функция `append` объединяет свои аргументы—списки в один список, который выдает в качестве результата.

```
* (append '(1 2 3 4) '(5 6 7 8) '(9 10 11))

(1 2 3 4 5 6 7 8 9 10 11)
```

Последний аргумент функции `append` может быть не списком, а точечной парой. В этом случае результатом вызова функции является точечная пара.

Композицию из четырех или менее вызовов функций `car` и `cdr` можно заменить одним вызовом специальной функции `c...r` где между символами `c` и `r` размещаются символы `a` (для `car`) и `d` (для `cdr`) в том же порядке, в каком вызовы `car` и `cdr` располагаются в композиции. Например вызовы

```
(car (cdr (car (cdr L))))
(car (car (car (cdr L))))
(car (cdr L))
(cdr (car (car L)))
```

можно заменить вызовами

```
(cadadr L)
(caaadr L)
(cadr L)
(cdaar L)
```

2.3.3 Функции сравнения

Функция `eq` принимает два аргумента и возвращает `T` если оба аргумента представляют из себя один и тот же объект (ссылаются на один и тот же объект).

```
* (eq '(1 2 3) '(1 2 3))

NIL
* (eq 23 23)

T
* (eq 'asd (car '(asd ert)))

T
```

Функция `equal` принимает два аргумента и возвращает `T` если аргументы равны в своем внешнем представлении.

```
* (equal '(1 2 3) '(1 2 3))

T
* (equal 23 23)

T
* (equal '(1 2) (cons  1 (cons 2 NIL)))

T
```

2.3.4 Математические функции

Арифметические действия выполняются с помощью функций `+`, `*`, `-`, `/`, `mod`, выполняющих сложение, умножение, вычитание, деление и поиск остатка от деления. Первые две из этих функций принимают произвольное количество аргументов, вторые две — от одного и более.

```
* (* 45
     36
     (/ 42 7)
     (mod (+ 64 17)
          13))

29160
```

Функции `=`, `/=`, `<=`, `>=`, `<`, `>` используется для сравнения числовых данных.

```
* (= 13 13)

T
* (/= 13 23)

T
* (= 12 (+ 5 7))

T
```

Предикат `zerop` возвращает `T` если аргумент равен нулю.

2.3.5 Условные конструкции

Конструкция `cond` является основной управляющей конструкцией языка. Конструкция принимает любое количество аргументов. Каждый аргумент — условная пара должен представлять собой списочную пару (условие результат). Функция вычисляет первый элемент каждой списочной пары в порядке следования аргументов до тех пор, пока не найдет пару у которой условие не равно `NIL`. Значение второго элемента этой пары выдается в качестве своего результата. Если условие всех переданных пар равно `NIL`, то `NIL` выдается в качестве результата.

```
* (cond
    ((= 13 23)            "Сравнение неудачное")
    ((equal 12 'A)        "Сравнение тоже неудачное")
    ((eq 'A (car '(A B))) "Успешное сравнение. Результат - эта фраза")
    (T "Условие истинно, но проверяться не будет, т.к. результат получен выше"))

"Успешное сравнение. Результат - эта фраза"
```

Конструкция `if` представляет реализацию условного выражения. Принимает три аргумента, первый из которых — условное выражение. Если результат вычисления условного выражения не `NIL` то вычисляется второй аргумент и его значение выдается как результат всего выражения. Если первый аргумент равен `NIL`, то второй аргумент не вычисляется, но вычисляется третий аргумент и его значение выдается как результат. Если третий аргумент отсутствует, то его значение предполагается равным `NIL`.

```
* (if (> 13 23)
    "Эта фраза выдана не будет"
    "Результат - эта фраза")

"Результат - эта фраза"
```

2.4 Определение функций

2.4.1 Определение неименованной функции

Для определения неименованной функции используется конструкция `lambda`. λ-выражение соответствует используемому в других языках определению процедуры или функции, а λ-вызов — вызову процедуры или функции. Например выражение

```
(lambda (x n) (/ x (+ x n)))
```

определяет неименованную функцию с двумя параметрами `x` и `n`. А вызов этой функции может выглядеть так

```
* ((lambda (x n) (/ x (+ x n))) -8 4)

2
```

2.4.2 Определение именованной функции

Для определения именованной функции можно использовать макрос `defun`.

```
* (defun func (x n) (/ x (+ x n)))

FUNC
```

Этот макрос имеет те же параметры, что и `lambda`, но к ним добавился первый параметр — имя функции, который не вычисляется. В качестве результата `defun` выдает имя определенной функции.

```
* (func -8 4)

2
```

2.4.3 λ-список

λ-списком называется список формальных параметров. При определении функции среди параметров в λ-списке можно указывать так-называемые ключевые слова, которые позволяют в дальнейшем трактовать параметры функции по разному. Таких ключевых слов четыре: `&optional`, `&key`, `&rest`, `&aux`. Упоминаться в λ-списке эти слова могут только по одному разу. Если ключевое слово упоминается в λ-списке, то оно относится ко всем параметрам расположенным в списке после него до следующего ключевого слова или до конца списка, если такое отсутствует.

Параметры, расположенные в λ-списке до первого ключевого слова являются обязательными параметрами функции.

Ключевое слово `&optional` указывает на то, что относящиеся к нему параметры являются необязательными и при вызове для них можно не указывать фактических параметров. В таком случае параметр связывается со значением по умолчанию, или, если оно отсутствует, со значением `NIL`.

```
* (defun func1 (&optional a b c) (list a b c))

FUNC1
* (func1)

(NIL NIL NIL)
* (func1 1)

(1 NIL NIL)
* (func1 1 2)

(1 2 NIL)
* (func1 1 2 3)

(1 2 3)
```

При указании фактических параметров связывание происходит в порядке следования формальных параметров в λ-списке.

Для указания значения по умолчанию пара формальный параметр — значение по умолчанию объединяется в список. В качестве значения по умолчанию может быть использовано любое S-выражение, в том числе в которых используются параметры описываемой функции, упомянутые

ранее в λ-списке.

```
* (defun func2 (x &optional (y (* x 2)) (z (+ y 5)))
    (list x y z))

FUNC2
* (func2 1)

(1 2 7)
* (func2 1 2)

(1 2 7)
* (func2 1 3)

(1 3 8)
* (func2 1 3 9)

(1 3 9)
```

Ключевое слово `&rest` говорит о том, что количество фактических параметров описываемой функции может превосходить количество формальных параметров. С формальным параметром расположенным в λ-списке после слова `&rest` связывается список всех значений фактических параметров, для которых не предусмотрено формального параметра.

```
* (defun func3 (x &optional (y 2) (z 3) &rest r)
    (list x y z r))

FUNC3
* (func3 1)

(1 2 3 NIL)
* (func3 1 3 5 7 9 11 13)

(1 3 5 (7 9 11 13))
```

Естественно предположить, что ключевому слову `&rest` может соответствовать только один формальный параметр. кроме того, ключевое слово `&optional` может встретиться в λ-списке только до слова `&rest`.

Ключевое слово `&key` служит для определения ключевых параметров. Ключевым параметром функции в LISP называется необязательный параметр для передачи значения которому требуется указание соответствующего символьного ключа (ключевого слова). Символьный ключ для каждого ключевого параметра представляет имя формального параметра перед которым поставлено двоеточие. Например при определении

```
* (defun func4 (&key x y z) (list x y z))

FUNC4
* (func4)

(NIL NIL NIL)
* (func4 :y 6)

(NIL 6 NIL)
* (func4 :y 6 :x 8)

(8 6 NIL)
* (func4 :z 4 :x 8)

(8 NIL 4)
```

Если при вызове функции ключевой параметр не указан, то он связывается со значением по умолчанию или, если оно отстутствует, с NIL.

Значения ключевых параметров при вызове функции можно указывать в произвольном порядке.

Для исключения недоразумений желательно не использовать в одном λ-списке ключевые слова `&optional` и `&key`. Если все же в функции ключевые параметры используются наряду с опциональными, то ключевые параметры должны стоять в λ-списке после опциональных. При вызове такой функции указание ключевых параметров возможно только если указаны все необязательные параметры.

Нельзя использовать в одном λ-списке ключевые слова `&rest` и `&key`.

Ключевое слово &aux говорит, что перечисленные далее символьные имена не являются на самом деле параметрами функции, а являются её вспомогательными локальными переменными. Этим именам не могут быть сопоставлены фактические параметры. Но для них можно, так же как и для необязательных параметров, указать значения по умолчанию (начальные значения).

```
* (defun func5 (x &optional (y 2) &aux (z (+ x y)))
    (list x y z))

FUNC5
* (func5 4)

(4 2 6)
* (func5 4 6)

(4 6 10)
* (func5 4 6 8)

debugger invoked on a SB-INT:SIMPLE-PROGRAM-ERROR in thread
#<THREAD "main thread" RUNNING {24086CA1}>:
  FUN6 called with invalid number of arguments: 3
```

При определении функции ключевое слово &aux может использоваться в сочетании с любыми другими ключевыми словами. Параметры описываемые с помощью &aux должны являться последними элементами λ-списка.

2.5 Локальные имена

2.5.1 Конструкции `let` и `let*`

Специальные конструкции `let` и `let*` позволяют создавать локальную область с набором символьных имен с соответствующими им значениями.

```
* (let (a
        (b 171)
        c)
    (list a b c))

(NIL 171 NIL)
* (let* ((a 25)
         (b (+ a 354)))
    (list b a))

(379 25)
```

Первый аргумент этих конструкций — список инициализаций локальных имен. Инициализация каждого имени выражается либо в его упоминании (тогда его начальным значением становится `NIL`) или в представлении списка — пары, в которой первый элемент — описываемое имя, а второй — выражение, значение которого связывается с этим именем.

Второй аргумент — выражение, значение которого становится значением всей конструкции.

Отличие `let*` от `let` в том, что инициализация локальных имен в `let` проводится параллельно, а в `let*` — последовательно. В выражениях, используемых для инициализации локальных имен в конструкции `let*`, могут использоваться имена, определенные ранее в этом же списке инициализации.

```
* (let ((a 5)
        (b 6))
    (let ((a 7)
          (b (+ 1 a))
          (c (+ b a)))
      (list a b c)))

(7 6 11)
* (let ((a 5)
        (b 6))
    (let* ((a 7)
           (b (+ 1 a))
           (c (+ b a)))
      (list a b c)))

(7 8 15)
```

При определении значения первого выражения во внутреннем вызове конструкции `let` три параметра определяются параллельно, и в выражениях для `b` и `c` используются значения `a` и `b`, определенные в списке инициализации локальных имен внешней конструкции `let` (5 и 6, соответственно). Во втором выражении в конструкции `let*` инициализация имен происходит после-

довательно, и при подсчете значения для `b` и `c` используется значения `a` и `b`, определенные ранее в этом же списке инициализации (7 и 8).

2.5.2 Конструкция `labels`

Конструкция `labels` как и конструкция `let` определяет область, в которой определены локальные функции, описанные в блоке инициализации. Каждый элемент списка — первого аргумента labels является функцией. Каждое описание такой функции представляет собой тройку — имя функции, λсписок, тело функции (как вызов конструкции `defun`, но без слова `defun`).

Второй аргумент в `labels` — произвольное S-выражение, в котором могут использоваться описанные в списке инициализации функции.

```
* (labels ((z (&optional x y)
             (list x y)))
    (z (z 5) (z 2 (z 3))))

((5 NIL) (2 (3 NIL)))
```